Real-Time Software Transactional Memory: Contention Managers, Time Bounds, and Implementations

Mohammed El-Shambakey

Dissertation Submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

September 13, 2013
Blacksburg, Virginia

Keywords: Software Transactional Memory, Embedded Systems, Contention Managers

Real-Time Software Transactional Memory: Contention Managers, Time Bounds, and Implementations

Mohammed El-Shambakey

(ABSTRACT)

Lock-based concurrency control suffers from programmability, scalability, and composability challenges. These challenges are exacerbated in emerging multicore architectures, on which improved software performance must be achieved by exposing greater concurrency. Transactional memory (TM) is an emerging alternative synchronization model for shared memory objects that promises to alleviate these difficulties.

In this dissertation, we consider software transactional memory (STM) for concurrency control in multicore real-time software, and present a suite of real-time STM contention managers for resolving transactional conflicts. The contention managers are called ECM, RCM, LCM, PNF, and FBLT. RCM and ECM resolve conflicts using fixed and dynamic priorities of real-time tasks, respectively, and are naturally intended to be used with the fixed priority (e.g., G-RMA) and dynamic priority (e.g., G-EDF) multicore real-time schedulers, respectively. LCM resolves conflicts based on task priorities as well as atomic section lengths, and can be used with G-EDF or G-RMA schedulers. Transactions under ECM, RCM, and LCM may retry due to conflicts with higher priority tasks even when there are no shared objects, i.e., transitive retry. PNF avoids transitive retry and optimizes processor usage by lowering the priority of retrying transactions, thereby enabling other non-conflicting transactions to proceed. PNF, however, requires a priori knowledge of all requested objects for each atomic section, which is inconsistent with the semantics of dynamic STM. Moreover, its centralized design increases overhead. FBLT avoids transitive retry, do not require a priori knowledge of requested objects, and has a decentralized design.

We establish upper bounds on transactional retry costs and task response times under the contention managers through schedulability analysis. Since ECM and RCM preserve the semantics of the underlying real-time scheduler, their maximum transactional retry cost is double the maximum atomic section length. This is improved in the design of LCM, which achieves shorter retry costs and tighter upper bounds. As PNF avoids transitive retry and improves processor usage, it yields shorter retry costs and tighter upper bounds than ECM, RCM, and LCM. FBLT's upper bounds are similarly tight because it combines the advantages of PNF and LCM.

We formally compare the proposed contention managers with each other, with lock-free synchronization, and with multiprocessor real-time locking protocols. Our analysis reveals that, for most cases, ECM, RCM, and LCM achieve higher schedulability than lock-free synchronization only when the atomic section length does not exceed half of lock-free synchronization's retry loop length. With equal periods and greater access times for shared objects, atomic section length under ECM, RCM, and LCM can be much larger than the retry loop length while still achieving better schedulability. With proper values for LCM's design parameters, atomic section length can be larger than the retry loop length for better schedulability. Under PNF, atomic section length can exceed lock-free's retry loop length and still achieve better schedulability in certain cases. FBLT achieves equal or better schedulability than lock-free with appropriate values for design parameters. The schedulability advantage of the contention managers over multiprocessor real-time locking protocols such as Global OMLP and RNLP depends upon the value of s_{max}/L_{max}, the ratio of the maximum transaction length to the maximum critical section length. FBLT's schedulability is equal or better than Global OMLP and RNLP if $s_{max}/L_{max} \leq 2$.

Checkpointing enables partial roll-back of transactions by recording transaction execution states (i.e., checkpoints) during execution, allowing roll-back to a previous checkpoint instead of transaction start, improving task response time. We extend FBLT with checkpointing and develop CP-FBLT, and identify the conditions under which CP-FBLT achieves equal or better schedulability than FBLT.

We implement the contention managers in the Rochester STM framework and conduct experimental studies using a multicore real-time Linux kernel. Our studies reveal that among the contention managers, CP-FBLT has the best average-case performance. CP-FBLT's higher performance is due to the fact that PNF's and LCM's advantages are combined into the design of FBLT, which is the base of CP-FBLT. Moreover, checkpointing improves task response time. The contention managers were also found to have equal or better average-case performance than lock-free synchronization: more jobs meet their deadlines using CP-FBLT, FBLT, and PNF than lock-free synchronization by 34.6%, 28.5%, and 32.4% (on average), respectively. The superiority of the contention managers is directly due to their better conflict resolution policies.

Locking protocols such as OMLP and RNLP were found to perform better: more jobs meet their deadlines under OMLP and RNLP than any contention manager by 12.4% and 13.7% (on average), respectively. However, the proposed contention managers have numerous qualitative advantages over locking protocols. Locks do not compose, whereas STM transactions do. To allow multiple objects to be accessed in a critical section, OMLP assigns objects to non-conflicting groups, where each group is protected by a distinct lock. RNLP assumes that objects are accessed in a specific order to prevent deadlocks. In contrast, STM allows multiple objects to be accessed in a transaction in any order, while guaranteeing deadlock-freedom, which significantly increases programmability. Moreover, STM offers platform independence: the proposed contention managers can be entirely implemented in the user-space as a library. In contrast, real-time locking protocols such as OMLP and RNLP must be supported by the underlying platform (i.e., operating system or virtual machine).

Dedication

To my parents, my wife, my daughter, and all my family

Acknowledgments

I would like to gratefully acknowledge the supervision of my advisor, Dr. Binoy Ravindran during my Ph.D. study, for his enthusiasm, and his great efforts to guide my research from the start.

Many thanks to the rest of my committee: Dr. Robert P. Broadwater, Dr. Cameron D. Patterson, Dr. Mohamed R. Rizk, and Dr. Anil Vullikanti for their invaluable advice and comments during my preliminary and defence exams. It is a great honour to have them serving in my committee.

In addition, I would like to thank all my previous and current colleagues in Systems Software Research Group, who provided my great environment for collaboration and discussion. Their warm suggestions and help made me never feel alone in this long journey.

Last but not least, thank all my family members for their love and support. I am grateful to my parents, who always did their best in supporting my education from the childhood, and suggested me the correct direction to make my dream come true. Also, I thank my dear wife who devoted her love and support to me through the ups and downs over the past years. It is difficult to overstate my gratitude to her for being such a wonderful wife.

This dissertation is dedicated to all the people who helped me and are helping me all the way.

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385. Any opinions, findings, and conclusions or recommendations expressed in this site are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Embedded systems sense physical processes and control their behavior, typically through feedback loops. Since physical processes are concurrent, computations that control them must also be concurrent, enabling them to process multiple streams of sensor input and control multiple actuators, all concurrently. Often, such computations need to concurrently read/write shared data objects. Typically, they must also process sensor input and react, satisfying application-level time constraints.

The de facto standard for programming concurrency is the threads abstraction, and the de facto synchronization abstraction is locks. Lock-based concurrency control has significant programmability, scalability, and composability challenges [73]. Coarse-grained locking (e.g., a single lock guarding a critical section) is simple to use, but permits no concurrency: the single lock forces concurrent threads to execute the critical section sequentially, in a one-at-a-time order. This is a significant limitation, especially with the emergence of multicore architectures, on which improved software performance must be achieved by exposing greater concurrency.

With fine-grained locking, a single critical section is broken down into several critical sections – e.g., each bucket of a hash table is guarded by a unique lock. Thus, threads that need to access different buckets can do so concurrently, permitting greater parallelism. However, this approach has low programmability: programmers must acquire only necessary and sufficient locks to obtain maximum concurrency without compromising safety, and must avoid deadlocks when acquiring multiple locks. Moreover, locks can lead to livelocks, lock-convoying, and priority inversion.

Perhaps, the most significant limitation of lock-based code is its non-composability. For example, atomically moving an element from one hash table to another using those tables' (lock-based) atomic methods is not possible in a straightforward manner: if the methods internally use locks, a thread cannot simultaneously acquire and hold the locks of the methods (of the two tables); if the methods were to export their locks, that will compromise safety.

Lock-free synchronization [72], which uses atomic hardware synchronization primitives (e.g., Compare And Swap [83, 84], Load-Linked/Store-Conditional [134]), also permits greater concurrency, but has even lower programmability: lock-free algorithms must be custom-designed for each situation (e.g., a data structure [30, 63, 71, 77, 112]). Most importantly, reasoning about the correctness of lock-free algorithms is significantly difficult [72].

1.1 Transactional Memory

Transactional memory (TM) is an alternative synchronization model for shared memory data objects that promises to alleviate these difficulties. With TM, programmers write concurrent code using threads, but organize code that read/write shared memory objects as *memory transactions*, which speculatively execute, while logging changes made to objects–e.g., using an undo-log or a write-buffer. Objects read and written by transactions are also monitored, in read-sets and write-sets, respectively. Two transactions conflict if they access the same object and one access is a write. (Conflicts are usually detected by detecting non-empty read- and write-set intersections.) When that happens, a contention manager (CM) resolves the conflict by aborting one and committing the other, yielding (the illusion of) atomicity. Aborted transactions are re-started, after rolling-back the changes–e.g., undoing object changes using the undo-log (eager), or discarding the write buffer (lazy).

In addition to a simple programming model (locks are excluded from the programming interface), TM provides performance comparable to lock-based synchronization [124], especially for high contention and read-dominated workloads, and is composable. TM's first implementation was proposed in hardware, called hardware transactional memory (or HTM) [76]. HTM has the lowest overhead, but HTM transactions are usually limited in space and time. Examples of HTMs include TCC [70], UTM [4], Oklahoma [136], ASF [44], and Bulk [35]. TM implementation in software, called software transactional memory (or STM) was proposed later [132]. STM transactions do not need any special hardware, are not limited in size or time, and are more flexible. However, STM has a higher overhead, and thus lower performance, than HTM. Examples of STMs include RSTM [144], TinySTM [122], Deuce [90], and AtomJava [78].

Listing 1.1: STM example

```
BEGIN_TRANSACTION;
        stm::wr_ptr<Counter> wr(m_counter);
        wr->set_value(wr->get_value(wr) + 1, wr);
END_TRANSACTION;
```

Listing 1.1 shows an example STM code written against RSTM [133]'s interface. RSTM's `BEGIN_TRANSACTION` and `END_TRANSACTION` keywords are used to enclose a critical section, which creates a transaction for the enclosed code block and guarantees its atomic execution. The first line inside the transaction creates a write pointer to a variable "m_counter" of type "Counter". The second line reads the current value of the counter variable through "wr−>get_value". The counter value is incremented through the "wr−>set_value" operation.

Hybrid TM (or HyTM) was subsequently proposed in [103], which combines HTM with STM, and avoids their limitations. Examples of HyTMs include SpHT [100], VTM [120], HyTM [45], LogTM [113], and LogTM-SE [151].

TM is increasingly gaining traction: Intel has released a C++ compiler with STM support [85]. Oracle [36, 139], AMD [1, 44], and Intel [86] have released experimental or commodity hardware with HTM support. GCC has released language extensions to support STM [138].

1.2 STM for Real-Time Software

Given the hardware-independence and flexibility of STM, which are significant advantages, we focus on STM in this dissertation. STM's programmability, scalability, and composability advantages are also compelling for concurrency control in multicore embedded real-time software. However, this will require bounding transactional retries, as real-time threads, which subsume transactions, must satisfy application-level time constraints. Transactional retry bounds in STM are dependent on the CM policy at hand (analogous to the way thread response time bounds are OS scheduler-dependent).

Despite the large body of work on STM contention managers, relatively few results are known on real-time contention management. STM concurrency control for real-time systems has been previously studied, but in a limited way. For example, [109] proposes a restricted version of STM for uniprocessors. Uniprocessors do not need contention management. [61] bounds response times in distributed multicore systems with STM synchronization. They consider Pfair scheduling [82], which is largely only of theoretical interest[1], limit to small atomic regions with fixed size, and limit transaction execution to span at most two quanta. [125] presents real-time scheduling of transactions and serializes transactions based on transactional deadlines. However, the work does not bound transactional retries and response times.

[128] proposes real-time HTM, which of course, requires hardware with TM support. The retry bound developed in [128] assumes that the worst-case conflict between atomic sections of different tasks occurs when the sections are released at the same time. We show that, this assumption does not cover the worst-case scenario (see Chapter 4). [60] presents a contention manager that resolves conflicts using task deadlines. The work also establishes upper bounds on transactional retries and task response times. However, similar to [128], [60] also assumes that the worst-case conflict between atomic sections occurs when the sections are released

[1]This is due to Pfair class of algorithm's time quantum-driven nature of scheduling and consequent high run-time overheads.

simultaneously. Besides, [60] assumes that all transactions have equal lengths. The ideas in [60] are extended in [13], which presents three real-time CM designs. But no retry bounds or schedulability analysis techniques are presented for those CMs.

Thus, past efforts on real-time STM are limited, and do not answer important fundamental questions:

(1) How to design "general purpose" real-time STM contention managers for multicore architectures? By general purpose, we mean those that do not impose any restrictions on transactional properties (e.g., transaction lengths, number of transactional objects, levels of transactional nestings), which are key limitations of past work.

(2) What tight upper bounds exist for transactional retries and task response times under such real-time CMs?

(3) How does the schedulability of real-time CMs compare with that of lock-free synchronization and real-time locking protocols? i.e., are there upper bounds or lower bounds for transactional lengths below or above which is STM superior to lock-free and locking protocols?

(4) How does transactional retry costs and task response times of real-time CMs compare with that of lock-free and real-time locking protocols in practice (i.e., on average)?

(5) How does real-time CMs qualitatively compare with lock-free synchronization and real-time locking? i.e., how does the techniques compare from an application and systems programmer's perspective?

1.3 Research Contributions

In this dissertation, we answer these questions. We present a suite of real-time STM contention managers, called RCM [54], ECM [54], LCM [53], PNF [57], and FBLT [56]. The contention managers progressively improve transactional retry and task response time upper bounds (and consequently improve STM's schedulability advantages) and also relax the underlying task models.

RCM and ECM resolve conflicts using fixed and dynamic priorities of real-time tasks, respectively, and are naturally intended to be used with the fixed priority (e.g., G-RMA [31]) and dynamic priority (e.g., G-EDF [31]) multicore real-time schedulers, respectively. LCM resolves conflicts based on task priorities as well as atomic section lengths, and can be used with G-EDF or G-RMA schedulers. Transactions under ECM, RCM, and LCM can restart because of other transactions that share no objects with them. This is called *transitive retry*. Transitive retry causes a transaction to abort and retry due to another non-conflicting transaction, which increases retry costs. Besides, under LCM, higher priority transactions can be blocked by lower priority transactions in certain conditions.

PNF avoids transitive retry, and also optimizes processor usage by lowering the priority of retrying transactions, and thereby allows other non-conflicting transactions to proceed. Thus, PNF reduces retry costs. However, PNF is a centralized contention manager and needs a priori knowledge of all requested objects of each transaction, which is not consistent with the semantics of dynamic STM [74]. Being a centralized contention manager, PNF induces significant overhead due to the contention between transactions (on the centralized contention manager). Under PNF, transactions may be non-conflicting, yet they delay each other as they wait to be served by the centralized contention manager. Also, under PNF, higher priority transactions and higher priority real-time tasks can be blocked by lower priority non-preemptive transactions.

FBLT combines the benefits of PNF and LCM, and avoids PNF's problems. Under FBLT, each newly released transaction is aborted for a specified number of times. Afterwards, similar to PNF, a transaction becomes non-preemptive and therefore cannot be aborted by any other preemptive or non-atomic real-time task. Also, FBLT is a decentralized contention manager that defaults to LCM when conflicting non-preemptive transactions have not reached their maximum abort numbers yet.

We establish upper bounds on transactional retry costs and task response times under the contention managers through schedulability analysis. Since ECM and RCM preserve the semantics of the underlying real-time scheduler, their maximum transactional retry cost is double the maximum atomic section length. This is improved in the design of LCM, which achieves shorter retry costs and upper bounds. However, ECM, RCM, and LCM are affected by transitive retries when transactions access multiple objects, which is eliminated in the design of PNF, yielding further reduced retry costs and upper bounds. FBLT combines the advantages of PNF and LCM by resolving conflicts using timestamps instead of transaction's original priority. This design yields further reduced retry costs and upper bounds.

We formally compare the proposed contention managers with each other, with lock-free synchronization [49], and with multiprocessor real-time locking protocols such as OMLP [22,29] and RNLP [149]. Our analysis reveals that, for most cases, ECM, RCM, and LCM (with G-EDF or G-RMA scheduler) achieve higher schedulability than lock-free synchronization only when the atomic section length does not exceed one half of the length of lock-free synchronization's "retry loop."[2] With equal periods and longer access times to shared objects, atomic section length under ECM, RCM, and LCM (with G-EDF or G-RMA) can be much larger than lock-free synchronization's retry loop length, while still achieving better schedulability. LCM (with G-EDF or G-RMA) achieves equal or better schedulability than lock-free for atomic section lengths larger than the lock-free retry loop length with appropri-

[2] Lock-free synchronization optimistically attempts to modify a shared data structure, and retries the attempt upon failure. This logic is often enclosed in a loop, which is repeated until the modification attempt succeeds [73].

ate values for design parameters. LCM/G-EDF has equal or better schedulability than ECM, whereas LCM/G-RMA's schedulability is equal or better than RCM's schedulability depending upon design parameters. Under PNF, atomic section length can exceed lock-free's retry loop length and still achieve better schedulability depending upon design parameters. FBLT also achieves equal or better schedulability than lock-free synchronization with appropriate values for design parameters.

We also establish the conditions under which the proposed contention managers achieve equal or better schedulability than the OMLP and RNLP locking protocols. The schedulability advantage of the contention managers depends upon the value of s_{max}/L_{max}, the ratio of the maximum transaction length to the maximum critical section length. With high number of processors and low number of tasks (besides other conditions), schedulability of the contention managers is equal or better than that of OMLP and RNLP while $s_{max}/L_{max} \geq 1$. FBLT's schedulability is equal or better than OMLP and RNLP if $s_{max}/L_{max} \leq 2$.

Checkpointing is a technique that enables partial roll-back of transactions [91]. The method involves saving the transaction execution state, i.e., taking a checkpoint, using special API calls, throughout the transaction execution. Thus, when a transaction is aborted, it can be rolled back to a previous checkpoint instead of the start of the transaction, improving task response time. Checkpoints can be recorded, for example, at each newly accessed object. Taking a checkpoint, however, induces overhead. To exploit checkpointing for real-time STM, we extend FBLT with checkpointing and develop CP-FBLT (i.e., FBLT augmented with checkpointing). We develop upper bounds for transaction retry costs and task response times for CP-FBLT. We also identify the conditions under which CP-FBLT achieves equal or better schedulability than FBLT. Our main result is that the overhead of taking checkpoints should not exceed the reduction in task response time. Otherwise, checkpointing is not effective.

Why are we concerned about expanding STM's schedulability advantage? When STM's schedulability advantage holds, programmers can reap STM's significant programmability and composability benefits in multicore real-time software. Thus, by expanding STM's schedulability advantage, we increase the range of real-time software for which those benefits can be tapped. Our results, for the first time, thus provides a fundamental understanding of when to use, and not use, STM concurrency control in multicore real-time software.

We also implement the proposed contention managers in the Rochester STM framework [133]. Using this implementation, running atop the ChronOS multicore real-time Linux kernel [48], which supports a range of multicore real-time schedulers, we conducted a comprehensive suite of experimental studies. The studies used a synthetic benchmark suite that was derived from the Baker's taskset [12], which is widely used in the real-time systems literature for similar experimental studies (e.g., [15, 25, 26, 33, 34, 47]).

Our studies reveal that among the contention managers, CP-FBLT has the best average-case performance. CP-FBLT's higher performance is due to the fact that PNF's and LCM's advantages are combined into the design of FBLT, which is the base of CP-FBLT. Moreover,

Table 1.1: Qualitative comparison between locking, STM and lock-free synchronization

	Locking	STM	Lock-free
Compositionality	No	Yes	No
Nesting	OMLP: No	Yes	No
	RNLP: Yes		
Deadlock	OMLP: No	No	No
	RNLP: Programmer dependent		
Implementation complexity	Hard	Simple	Hard
Transparency	OMLP: No	Yes	Yes
	RNLP: Yes		
Platform dependence	Dependent	Independent	Independent

checkpointing improves task response time.

The contention managers show equal or better average-case performance than lock-free: more jobs meet their deadlines using CP-FBLT, FBLT, and PNF than lock-free synchronization by 34.6%, 28.5%, and 32.4% (on average), respectively. The superiority of the contention managers over lock-free is directly due to their better conflict resolution policies. The comparison between the contention managers and the locking protocols, however, revealed the superiority of the latter: more jobs meet their deadlines under OMLP and RNLP than any contention manager by 12.4% and 13.7% on average, respectively. However, the contention managers have numerous *qualitative* advantages over locking protocols. Locks do not compose, whereas STM transactions do. Support for nested critical sections is generally complicated for locking protocols, whereas it is trivial with STM. To allow multiple objects to be accessed in a critical section, OMLP assigns objects to non-conflicting groups, where each group is protected by a distinct lock. RNLP assumes that objects are accessed in a specific order to prevent deadlocks. In contrast, STM allows multiple objects to be accessed in a transaction in any order, while guaranteeing deadlock-freedom, which significantly increases programmability. From a systems programmer's perspective, OMLP and RNLP are relatively difficult to implement, whereas proposed contention managers are easy to implement. From an application programmer's perspective, OMLP is not transparent as it requires the description of additional information (i.e., what objects will be needed in each critical section). For RNLP to avoid order on object access, RNLP needs to know required objects for each critical section a priori. In contrast, no such extra information is needed for using proposed contention managers (except for PNF), which significantly increases programmability. STM offers platform independence: the proposed contention managers can be entirely implemented in the user-space as a library. In contrast, OMLP and RNLP must be supported by the underlying platform (i.e., operating system or virtual machine). Table 1.1 qualitatively compares locking, STM, and lock-free synchronization.

1.4 Organization

The rest of this dissertation is organized as follows.

Chapter 2 overviews past and related work on real-time concurrency control. Chapter 3 describes our task and system models, and other assumptions.

Chapter 4 describes the ECM and RCM contention managers, derives upper bounds for their retry costs and response times, and compares their schedulability with lock-free synchronization and real-time locking protocols. Chapters 5, 6, and 7 similarly describe the LCM, PNF, and FBLT contention managers, respectively.

Chapter 8 extends FBLT (Chapter 7) with checkpointing to develop CP-FBLT. We upper bound retry costs and response times under CP-FBLT, and derive conditions under which CP-FBLT has equal or better schedulability than FBLT in Chapter 8.

Chapter 9 describes our implementation and reports our experimental studies. Appendix A presents properties of all tasksets used in experiments. Appendecies B and C presents results in full.

Chapter 10 qualitatively compares proposed contention managers against lock-free and locking protocols.

We conclude the dissertation and outline potential future research directions in Chapter 11.

Chapter 2

Past and Related Work

Many mechanisms appeared for concurrency control for real-time systems. These methods include locking [32, 101], lock-free [5, 7, 8, 39, 42, 49, 58, 59, 81, 95] and wait-free [6, 16, 37, 39, 40, 43, 59, 79, 121, 137, 140, 141]. In general, real-time locking protocols have disadvantages like: 1) serialized access to shared object, resulting in reduced concurrency and reduced utilization. 2) increased overhead due to context switches. 3) possibility of deadlock when lock holder crashes. 3) some protocols requires apriori knowledge of ceiling priorities of locks. This is not always available. 4) Operating system data structures must be updates with this knowledge which reduces flexibility. For real-time lock-free, the most important problem is to bound number of failed retries and reduce cost of a single loop. The general technique to access lock-free objects is "retry-loop". Retry-loop uses atomic primitives (e.g., CAS) which is repeated until success. To access a specific data structure efficiently, lock-free technique is customized to that data structure. This increases difficulty of response time analysis. Primitive operations do not access multiple objects concurrently. Although some attempts made to enable multi-word CAS [7], but it is not available in commodity hardware [111]. For real-time wait-free protocols. It has a space problem due to use of multiple buffers. This is inefficient in some applications like small-memory real-time embedded systems. Wait-free has the same problem of lock-free in handling multiple objects.

The rest of this Chapter is organized as follows, Section 2.1 summarizes previous work on real-time locking protocols. In Section 2.2, we preview related work on lock-free and wait-free methods for real-time systems. Section 2.3 provides concurrency control under real-time database systems as a predecessor and inspiration for real-time STM. Section 2.4 previews related work on contention management. Contention management policy affects response time analysis of real-time STM.

2.1 Real-Time Locking Protocols

A lot of work has been done on real-time locking protocols. Locks in real-time systems can lead to priority inversion [32, 101]. Under priority inversion, a higher priority job is not allowed to run because it needs a resource locked by a lower priority job. Different locking protocols appeared to solve this problem, but exposing other problems. Most of real-time blocking protocols are based on *Priority Inheritance Protocol (PIP)* [32, 51, 130], *Priority Ceiling Protocol (PCP)* [32, 38, 51, 89, 96, 118, 119, 130] and *Stack Resource Protocol (SRP)* [11, 32, 65].

In PIP [32, 130], resource access is done in FIFO order. A resource holder inherits highest priority of jobs blocked on that resource. When resource holder releases the resource and it holds no other resources, its priority is returned to its normal priority. If it holds other resources, its priority is returned to highest priority job blocked on other resources. Under PIP, a high priority job can be blocked by lower priority jobs for at most the minimum of number of lower priority jobs and number of shared resources. PIP suffers from chained blocking, in which a higher priority task is blocked for each accessed resource. Besides, PIP suffers from deadlock where each of two jobs needs resources held by the other. So, each job is blocked because of the other. [51] provides response time analysis for PIP when used with fixed-priority preemptive scheduling on multiprocessor system.

PCP [32, 119, 130] provides concept of priority ceiling. Priority ceiling of a resource is the highest priority of any job that can access that resource. For any job to enter a critical section, its priority should be higher the priority ceiling of any currently accessed resource. Otherwise, the resource holder inherits the highest priority of any blocked job. Under PCP, a job can be blocked for at most one critical section. PCP prevents deadlocks. [38] extends PCP to dynamically scheduled systems.

Two protocols extend PCP to multiprocessor systems: 1) *Multiprcoessor PCP (M-PCP)* [96, 118, 119] discriminates between global resources and local resources. Local resources are accessed by PCP. A global resource has a base priority greater than any task normal priority. Priority ceiling of a global resource equals sum of its base priority and highest priority of any job that can access it. A job uses a global resource at the priority ceiling of that resource. Requests for global resources are enqueued in a priority queue according to normal priority of requesting job. 2) *Parallel-PCP (P-PCP)* [51] extends PCP to deal with fixed priority preemptive multiprocessor scheduling. P-PCP, in contrast to PCP, allows lower priority jobs to allocate resources when higher priority jobs already access resources. Thus, increasing parallelism. Under P-PCP, a higher priority job can be blocked multiple times by a lower priority job. With reasonable priority assignment, blocking time by lower priority jobs is small. P-PCP uses α_i parameter to specify permitted number of jobs with basic priority lower than i and effective priority higher than i. When α_i is small, parallelism is reduced, so as well blocking from lower priority tasks. Reverse is true. [51] provides response time analysis for P-PCP.

[106] extends P-PCP to provide *Limited-Blocking PCP (LB-PCP)*. LB-PCP provides more control on indirect blocking from lower priority tasks. LB-PCP specify additional counters that control number of times higher priority jobs can be indirectly blocked without the need of reasonable priority assignment as in P-PCP. [106] analyzes response time of LB-PCP and experimentally compares it to P-PCP. Results show that LB-PCP is appropriate for task sets with medium utilization.

PCP can be unfair from blocking point of view. PCP can cause unnecessary and long blocking for tasks that do not need any resources. Thus, [89] provides Intelligent PCP (IPCP) to increase fairness and to work in dynamically configured system (i.e., no a priori information about number of tasks, priorities and accessed resources). IPCP initially optimizes priorities of tasks and resources through learning. Then, IPCP tunes priorities according to system wide parameters to achieve fairness. During the tuning phase, penalties are assigned to tasks according to number of higher priority tasks that can be blocked.

SRP [11, 32, 65] extends PCP to allow multiunit resources and dynamic priority scheduling and sharing runtime stack-based resources. SRP uses *preemption level* as a static parameter assigned to each task despite its dynamic priority. Resource ceiling is modified to include number of available resources and preemption levels. System ceiling is the highest resource ceiling. A task is not allowed to preempt unless it is the highest priority ready one, and its preemption level is higher than the system ceiling. Under SRP, a job can be blocked at most for one critical section. SRP prevents deadlocks. *Multiprocessor SRT (M-SRP)* [64] extends SRP to multiprocessor systems. M-SRP, as M-PCP, discriminates between local and global resources. Local resources are accessed by SRP. Request for global resource are enqueued in a FIFO queue for that resource. Tasks with pending requests busy-wait until their requests are granted.

Another set of protocols appeared for PFair scheduling [20]. [80] provide initial attempts to synchronize tasks with short and long resources under PFair. In Pfair scheduling, each task receives a weight that corresponds to its share in system resources. Tasks are scheduled in quanta, where each quantum has a specific job on a specific processor. Each lock has a FIFO queue. Requesting tasks are ordered in this FIFO queue. If a task is preempted during critical section, then other tasks can be blocked for additional time known as *frozen time*. Critical sections requesting short resources execute at most in two quanta. By early lock-request, critical section can finish in one quanta, avoiding the additional blocking time. [80] proposes two protocols to deal with short resources: 1) *Skip Protocol (SP)* leaves any lock request in the FIFO queue during frozen interval until requesting task is scheduled again. 2) *Rollback Protocol (RP)* discards any request in the FIFO queue for the lock during frozen time. For long resources, [80] uses *Static Weight Server Protocol (SWSP)* where requests for each resource l is issued to a corresponding server S. S orders requests in a FIFO queue and has a static specific weight.

Flexible Multiprocessor Locking Protocol (FMLP) [20] is the most famous synchronization protocol for PFair scheduling. The FMLP allows non-nested and nested resources access without constraints. FMLP is used under global and partitioned deadline scheduling. Short or long resource is user defined. Resources can be grouped if they are nested by some task and have the same type. Request to a specific resource is issued to its containing group. Short groups are protected by non-preemptive FIFO queue locks, while long groups are protected by FIFO semaphore queues. Tasks busy-wait for short resources and suspend on long resources. Short request execute non-preemptively. Requests for long resources cannot be contained within requests for short resources. A job executing a long request inherits highest priority of blocked jobs on that resource's group. FMLP is deadlock free.

[22] is concerned with suspension protocols. Schedulability analysis for suspension protocols can be suspension-oblivious or suspension-aware. In suspension-oblivious, suspension time is added to task execution. While in suspension-aware, it is not. [22] provides *Optimal Multiprocessor Locking Protocol (OMLP)*. Under OMLP, each resource has a FIFO queue of length at most m, and a priority queue. Requests for each resource are enqueued in the corresponding FIFO queue. If FIFO queue is full, requests are added to the priority queue according to the requesting job's priority. The head of the FIFO queue is the resource holding task. Other queued requests are suspended until their turn come. OMLP achieves $O(m)$ priority inversion (pi) blocking per job under suspension oblivious analysis. This is why OMLP is asymptotically optimal under suspension oblivious analysis. Under suspension aware analysis, FMLP is asymptotically optimal. [23] extends work in [22] to clustered-based scheduled multiprocessor system. [23] provides concept of *priority donation* to ensure that each job is preempted at most once. In priority donation, a resource holder priority can be unconditionally increased. Thus, a resource holder can preempt another task. The preempted task is predetermined such that each job is preempted at most once. OMLP with priority donation can be integrated with k-exclusion locks (K-OMLP). Under K-exclusion locks, there are k instances of the same resource than can be allocated concurrently. K-OMLP has the same structure of OMLP except that there are K FIFO queues for each resource. Each FIFO queue corresponds to one of the k instances. K-OMLP has $O(m/k)$ bound for pi-blocking under s-oblivious analysis. [55] extends the K-OMLP in [23] to global scheduled multiprocessor systems. The new protocol is *Optimal K-Exclusion Global Locking Protocol (O-KGLP)*. Despite global scheduling is a special case of clustering, K-OMLP provides additional cost to tasks requesting no resources if K-OMLP is used with global scheduling. O-KGLP avoids this problem.

[149] presents RNLP protocol to access nested resources. FMLP [20], FMLP+ [26] and OMLP family protocols [22, 29] use "group locking" to support nesting. "Group Locking" combines all resources that can be accessed by one atomic section in any task within a single group. Each group is protected by a single lock. Any task acquiring a resource must hold the group lock first. "Group locking" reduces parallelism if two or more tasks are not accessing the same resources, yet these tasks execute serially because resources belong to the same group. "Group locking" requires static assignment of resources to groups before execution. While "Group Locking" supports coarse-grained resource nesting, RNLP supports fine-grained resource nesting. RNLP is used under s-oblivious and s-aware analysis.

Progress is guaranteed via priority inheritance, priority boosting or priority donation. RNLP worst case analysis approximates worst case analysis of FMLP, FMLP+ and OMLP family, but RNLP increases parallelism through fine-grained resource nesting. Thus, average case analysis for RNLP is better than other protocols. RNLP requires a strict partial order on resources (which can be avoided in case of global scheduling). Resources should be acquired in their order to prevent deadlocks and improve pi-blocking bounds. RNLP is composed of two phases: *Token Lock* and *Request Satisfaction Mechanism (RSM)*. These two stages differ according to the scheduler and waiting mechanism. *Token Lock* bounds number of jobs with incomplete resource requests to k. k is usually n or m. RSM serializes access to shared resources. In RSM, each resource θ_a has a queue of k length. For a job τ_i^x to access θ_a, τ_i^x must first obtain a token. $ts(\tau_i^x)$ is the timestamp of token acquisition. τ_i^x is inserted in θ_a's queue according to increasing timestamp order. τ_i^x acquires θ_a when τ_i^x is the head of θ_a's queue, and no other task τ_j^y accesses a resource $\theta_b < \theta_a$ and $ts(\tau_j^y) < ts(\tau_i^x)$. τ_i^x can access nested resources using the same rules. τ_i^x releases its token when it finishes its outer most request. If requested resources in any critical section for τ_i^x are known a priori, this relaxes RNLP's rules and improves concurrency [148]. If requested resources are known a priori for each atomic section, then τ_i^x can access resources in any order without deadlock. This is because requests are spotted in the corresponding resources' queues at the beginning of each critical section in increasing timestamp order. Design of *Token Lock* differs according to scheduler and waiting mechanism. For global schedulers, RNLP makes use of a k-exclusion locking protocol named O-KGLP [55] to implement the *Token Lock*. [149] suggests an improved k-exclusion locking protocol (I-KGLP) for globally scheduled systems that reduces pi-blocking bounds (I-KGLP exists in the longer version of [149] and introduced as *Replica Request Donation Global Locking Protocol* (R^2DGLP) later in [150]). Under globally scheduled systems when I-KGLP is used with priority inheritance RSM (I-RSM), pi-blocking is bounded by $(2m-1)L_{max}$ per outermost request, where L_{max} is the maximum length of any outermost critical section.

[148] extends RNLP [149]. Atomic resource request is done using *Dynamic Group Locks (DGL)*. The timestamp of token acquisition is recorded for each requested resource in a DGL in the corresponding resource queue. A job is ready when it is the head of all requested resource queues. DGL requests can be nested themselves. DGL reduces progress blocking when certain jobs are uselessly blocked. This reduction is done by splitting resources into groups that are not going to nest together. Thus, different instances of RNLP can be instantiated for different resource groups with different progress and request blocking bounds. Worst-case blocking bounds using DGL are the same under original RNLP [149]. [148] discriminates between short and long resources. Short resource requests do not have to wait on long resources. Each type of resources has their own tokens. Short requests can be nested with long requests, but versa is not true. A request of nested long and short resources competes for a long token, while a request with only short resources competes for a short token. Long requests cannot reserve slots in short resources queues, even if the short resources will be nested in the future. [148] supports concurrent access to replicas of multiple resources. If DGLs are used, then atomic resource requests are placed in the shortest queues

of corresponding resources.

2.2 Real-Time Lock-Free and Wait-Free Synchronization

Due to locking problems (e,g,. priority inversion, high overhead and deadlock), research has been done on non-blocking synchronization using lock-free [5, 7, 8, 39, 42, 58, 59, 81, 95] and wait-free algorithms [6,16,37,39,40,43,59,79,121,137,140,141]. Lock-free iterates an atomic primitive (e.g., CAS) inside a retry loop until successfully accessing object. When used with real-time systems, number of failed retries must be bounded [5,7]. Otherwise, tasks are highly likely to miss their deadlines. Wait-free algorithms, on the other hand, bound number of object access by any operation due to use of sized buffers. Synchronization under wait-free is concerned with: 1) single-writer/multi-readers where a number of reading operations may conflict with one writer. 2) multi-writer/multi-reader where a number of reading operations may conflict with number of writers. The problem with wait-free algorithms is its space cost. As embedded real-time systems are concerned with both time and space complexity, some work appeared trying to combine benefits of locking and wait-free.

[5] considers lock-free synchronization for hard-real time, periodic, uniprocessor systems. [5] upper bounds retry loop failures and derives schedulability conditions with Rate Monotonic (RM), and Earliest Deadline First (EDF). [5] compares, formally and experimentally, lock-free objects against locking protocols. [5] concludes that lock-free objects often require less overhead than locking-protocols. They require no information about tasks and allow addition of new tasks simply. Besides, lock-free object do not induce excessive context switches nor priority inversion. On the other hand, locking protocols allow nesting. Besides, performance of lock-free depends on the cost of "retry-loops". [7] extends [5] to generate a general framework for implementing lock-free objects in uniprcocessr real-time systems. The framework tackles the problem of multi-objects lock-free operations and transactions through multi-word compare and swap (MWCAS) implementation. [7] provides a general approach to calculate cost of operation interference based on linear programming. [7] compares the proposed framework with real-time locking protocols. Lock-free objects are prefered if cost of retry-loop is less than cost of lock-access-unlock sequence. [8] extends [5,7] to use lock-free objects in building memory-resident transactions for uniprocessor real-time systems. Lock-free transactions, in contrast to lock-based transactions, do not suffer from priority inversion, deadlocks, complicated data-logging and rolling back. Lock-free transaction do not require kernel support.

[49] presents two synchronization methods under G-EDF scheduled real-time multiprocessor systems for simple objects. The first synchronization technique uses queue-based spin locks, while the other uses lock-free. The queue lock is FIFO ordered. Each task appends an entry at the end of the queue, and spins on it. While the task is spinning, it is non-

preemptive. The queue could have been priority-based but this complicates design and does not enhance worst case response time analysis. Spinning is suitable for short critical sections. Disabling preemption requires kernel support. So, second synchronization method uses lock-free objects. [49] bounds number of retries. [49] , analytically and experimentally, evaluates both synchronization techniques for soft and hard real-time analysis. [49] concludes that queue locks have a little overhead. They are suitable for small number of shared object operations per task. Queue locks are not generally suitable for nesting. Lock-free have high overhead compared with queue locks. Lock-free is suitable for small number of processors and object calls in the absence of kernel support.

[81] uses lock-free objects under PFair scheduling for multiprocessor system. [81] provides concept of *supertasking* to reduce contention and number of failed retries. This is done by collecting jobs that need a common resource into the same supertask. Members of the same supertask run on the same processor. Thus, they cannot content together. [81] upper bounds worst case duration for lock-free object access with and without supertasking. [81] optimizes, not replaces, locks by lock-free objects. Locks are still used in situations like sharing external devices and accessing complex objects.

Lock-free objects are used with time utility models where importance and criticality of tasks are separated [42,95]. [95] presents *MK-Lock-Free Utility Accrual (MK-LFUA)* algorithm that minimizes system level energy consumption with lock-free synchronization. [42] uses lock-free synchronization for dynamic embedded real-time systems with resource overloads and arbitrary activity arrivals. Arbitrary activity arrivals are modelled with Universal Arrival Model (UAM). Lock-free retries are upper bounded. [42] identifies the conditions under which lock-free is better than lock-based sharing. [58] builds a lock-free linked-list queue on a multi-core ARM processor.

Wait-free protocols use multiple buffers for readers and writers. For single-writer/multiple-readers, each object has a number of buffers proportional to maximum number of reader's preemptions by the writer. This bounds number of reader's preemptions. Readers and writers can use different buffers without interfering each other.

[43] presents wait-free protocol for single-writer/multiple-readers in small memory embedded real-time systems. [43] proves space optimality of the proposed protocol, as it required the minimum number of buffers. The protocol is safe and orderly. [43] also proves, analytically and experimentally, that the protocol requires less space than other wait-free protocols. [40] extends [43] to present wait-free utility accrual real-time scheduling algorithms (RUA and DASA) for real-time embedded systems. [40] derives lower bounds on accrued utility compared with lock-based counterparts while minimizing additional space cost. Wait-free algorithms experimentally exhibit optimal utility for step time utility functions during underload, and higher utility than locks for non-step utility functions. [121] uses wait-free to build three types of concurrent objects for real-time systems. Built objects has persistent states even if they crash. [141] provides wait-free queue implementation for real-time Java specifications.

A number of wait-free protocols were developed to solve multi-writer/multi-reader problem in real-time systems. [140] provides m-writer/n-reader non-blocking synchronization protocol for real-time multiprocessor system. The protocol needs $n + m + 1$ slots. [140] provides schedulability analysis of the protocol. [6] presents wait-free methods for multi-writer/multi-reader in real-time multiprocessor system. The proposed algorithms are used for both priority and quantum based scheduling. For a B word buffer, the proposed algorithms exhibit $O(B)$ time complexity for reading and writing, and $\Theta(B)$ space complexity. [137] provides a space-efficient wait-free implementation for n-writer/n-reader synchronization in real-time multiprocessor system. The proposed algorithm uses timestamps to implement the shared buffer. [137] uses real-time properties to bound timestamps. [37] presents wait-free implementation of the multi-writer/multi-reader problem for real-time multiprocessor synchronization. The proposed mechanism replicates single-writer/multi-reader to solve the multi-writer/multi-reader problem. [37], as [137], uses real-time properties to ensure data coherence through timestamps.

Each synchronization technique has its benefits. So, a lot of work compares between locking, lock-free and wait-free algorithms. [59] compares building snapshot tool for real-time system using locking, lock-free and wait-free. [59] analytically and experimentally compares the three methods. [59] concludes that wait-free is better than its competitors. [39] presents synchronization techniques under LNREF [41] (an optimal real-time multiprocessor scheduler) for simple data structures. Synchronization mechanisms include lock-based, lock-free and wait-free. [39] derives minimum space cost for wait-free synchronization. [39] compares, analytically and experimentally, between lock-free and lock-based synchronization under LNREF.

Some work tried to combine different synchronization techniques to combine their benefits. [79] uses combination of lock-free and wait-free to build real-time systems. Lock-free is used only when CAS suffices. The proposed design aims at allowing good real-time properties of the system, thus better schedulability. The design also aims at reducing synchronization overhead on uni and multiprocessor systems. The proposed mechanism is used to implement a micro-kernel interface for a uni-processor system. [16] combines locking and wait-free for real-time multiprocessor synchronization. This combination aims to reduce required space cost compared to pure wait-free algorithms, and blocking time compared to pure locking algorithms. The proposed scheme is jsut an idea. No formal analysis nor implementation is provided.

2.3 Real-Time Database Concurrency Control

Real-time database systems (RTDBS) is not a synchronization technique. It is a predecessor and inspiration for real-time transactional memory. RTDBS itself uses synchronization techniques when transactions conflict together. RTDBS is concerned not only with logical data consistency, but also with temporal time constraints imposed on transactions. Temporal time constraints require transactions finish before their deadlines. External constraints require updating temporal data periodically to keep freshness of database. RTDBS allow mixed types of transactions. But a whole transaction is of one type. In real-time TM, a single task may contain atomic and non-atomic sections.

High-Priority two Phase Locking (HP-2PL) protocol [97, 98, 116, 153] and *Real-Time Optimistic Concurrency (RT-OCC)* protocol [46, 62, 97–99, 153] are the most two common protocols for RTDBS concurrency . HP-2PL works like 2PL except that when a higher priority transaction request a lock held by a lower priority transaction, lower priority transaction releases the lock in favor of the higher priority one. Then, lower priority transaction restarts. RT-OCC delays conflict resolution till transaction validation. If validating transaction cannot be serialized with conflicting transactions, a priority scheme is used to determine which transaction to restart. In *Optimistic Concurrency Control with Broadcast Commit (OCC-BC)*, all conflicting transactions with the validating one are restarted. HP-2PL may encounter deadlock and long blocking times, while transactions under RT-OCC suffer from restart time at validation point.

Other protocols were developed based on HP-2PL [97, 98, 116] and RT-OCC [10, 62, 97, 99]. HP-2PL, and its derivatives, are similar to locking protocols in real-time systems. They have the same problems in real-time locking protocols like priority inversion. So, the same solutions exist for the RTDBS locking protocols. Despite RT-OCC, and its derivatives, use locks in their implementation, their behaviour is closer to abort and retry semantics in TM. Some work integrates different protocols to handle different situations [116, 152].

[97] presents *Reduced Ceiling Protocol (RCP)* which is a combination of *Priority Ceiling Protocol (PCP)* and *Optimistic Concurrency Protocol (OCC)*. RCP targets database systems with mixed hard and soft real-time transactions (RTDBS). RCP aims at guarantee of schedulabiltiy of hard real-time transactions, and minimizing deadline miss of soft real-time transactions. Soft real-time transactions are blocked in favor of conflicting hard real-time transactions. While hard real-time transactions use PCP to synchrnonize among themselves, soft real-time transactions use OCC. Hard real-time transactions access locks in a *tow phase locking (2PL)* fashion. Seized locks are released as soon as hard real-time transaction no longer need them. This reduces blocking time of soft real-time transactions. [97] derives analytical and experimental evaluation of RCP against other synchronization protocols.

[152], like [97], deals with mixed transaction. [152] classifies mixed transactions into hard (HRT), soft (SRT) and non (NRT) real-time transactions. HRT has higher priority than SRT. SRT has higher priority than NRT. [152] aims at guranting deadlines of HRTs, minimizing miss rate of SRTs and reducing response time of NRTs. So, [152] deals with inter and intra-transaction concurrency. HRTs use PCP for concurrency control among themselves. SRTs use WAIT-50, and NRTs use 2PL. SRT and NRT are blocked or aborted in favor of HRT. If NRT requests a lock held by SRT, then NRT is blocked. If SRT requests a lock held by NRT, WAIT-50 is applied. Experimental evaluation showed effective improvement in overall system performance. Performance objectives of each transaction type was met.

[62] is concerned with semantic lock concurrency control. The semantic lock technique

allows negotiation between logical and temporal constraints of data and transactions. It also controls imprecision resulting from negotiation. Thus, the semantic lock considers scheduling and concurrency of transactions. Semantic lock uses a compatibility function to determine if the release transaction is allowed to proceed or not.

Time Interval OCC protocols try to reduce number of transaction restarts by dynamic adjustment of serialization timestamps. Time interval OCC may encounter unnecessary restarts. [10] presents Timestamp Vector based OCC to resolve these unnecessary restarts. Timestamp Vector base OCC uses a timestamp vector instead of a single timestamp as in Time Interval OCC protocols. Experimental comparison between Timestamp Vector OCC and previous Time Interval OCC shows higher performance of Timestamp Vector OCC.

[46] aims to investigate performance improvement of priority congnizant OCC over incognizant counterparts. In OCC-BC, all conflicting transactions with the validating transaction are restarted. [46] wonders if it is really worthy to sacrifice all other transactions in favor of one transaction. [46] proposes *Optimistic Concurrency Control- Adaptive PRiority (OCC-APR)* to answer this question. A validating transaction is restarted if it has sufficient time to its deadline if restarted, and higher priority transactions cannot be serialized with the conflicting transaction. Sufficient time estimate is adapted according to system feedback. System feedback is affected by contention level. [46] experimentally concludes that integrating priority into concurrency control management is not very useful. Time Interval OCC showed better performance.

WAIT-X [46, 99] is one of the optimistic concurrency control (OCC) protocols. WAIT-X is a prospective (forward validation) OCC. Prospective means it detects conflicts between a validating transaction and conflicting transaction that may commit in the future. In retrospective (backward validation) protocols, conflicts are detected between a validating transaction and already committed transactions. Retrospective validation aborts validating transaction if it cannot be serialized with already committed conflicting transactions. When WAIT-X detects a conflict, it can either abort validating transaction, or commit validating transaction and abort other conflicting transactions, or it can dealy validating a transction slightly hoping that conflicts resolve themsleves someway. Which action to take is a function of priorities of vlaidating and conflicting transactions. WAIT-X can delay validating transaction until percetage of higher priority transactions in the conflict set is lower than X%. WAIT-50 is a common implementation of WAIT-X.

[92] is concerned with concurrency control for multiprocessor RTDBS. [92] uses priority cap to modify *Reader/Write Prirority Ceiling Protocol (RWPCP)* [131] to work on multiprocessor systems. The proposed protocol, named *One Priority Inversion RWPCP (1PI-RWPCP)*, is deadlock-free and bounds number of priority inversions for any transaction to one. [92] derives feasiblity condition for any transaction under 1PI-RWPCP. [92] experimentally compares performance of 1PI-RWPCP against RWPCP.

[116] combines locking, multi-version and valid confirmation concurrency control mechanisms. The proposed method adopts different concurrency control mechanism according to

idiographic situation. Experiments show lower rate of transactional restart of the proposed mechanis compared to 2PL-HP.

[98] is concerned with RTDBS containing periodically updated data and one time transactactions. [98] provides two new concurrency control protocols to balance freshness of data and transaction performance. [98] proposes *HP-2PL with Delayed Restart (HP-2PL-DR)* and *HP-2PL with Delayed Restart and Pre-declaration (HP-2PL-DRP)* based on HP-2PL. Before a transaction T restarts in HP-2PL-DR, next update time of each temporal data accessed by T is checked. If next update time starts before currently re-executing T, then T's restart time is delayed until the next udpate. Otherwise, T is restarted immediately. If T_r and T_n are two transactions under HP-2PL-DRT. T_r is requesting a lock held by T_n. If priority of T_r is greater than priority of T_n, then T_n releases the lock in favor of T_r. Othewise, T_r fails. If T_n releases the lock and T_n is a one time transaction, then T_n restarts immediately. Otherwise, T_n lock waiting time is updated. Experiments show improved performance of HP-2PL-DR and HP-2PL-DRP over HP-2PL.

2.4 Real-Time TM Concurrency Control

Concurrency control in TM is done through contention managers. Contention managers are used to ensure progress of transactions. If one or more transactions conflict on an object, contention manager decides which transaction to commit. Other transactions abort or wait. Mostly, contention managers are *distributed* or *decentralized* [68, 69, 126, 127], in the sense that each transaction maintains its own contention manager. Contention managers may not know which objects will be needed by transactions and their duration. Past work on contention managers can be classified into two classes: 1) Contention management policy that decides which transaction commits and which do other actions [67–69, 126, 127, 135]. 2) Implementation of contention management policy in practice [19, 50, 66, 108, 126, 135]. The two classes are orthogonal. The second class tries to increase the benefit of the the contention management policy in reality by considering different aspects in TM design (e.g., lazy versus eager, visible versus invisible readers). Second class suggests contention managers should be proactive instead of reactive. This can prevent conflicts before they happen. Contention managers can be supported a lot if they are integrated into system schedulers. This provides a global view of the system (due to applications feedback) and reduces overhead of the implementation of contention manager.

Contention management policy ranges from never aborting enemies to always aborting them [126, 127]. These two extremes can lead to deadlock, starvation, livelock and major loss of performance. Contention manager policy lies in between. Depending on heuristics, contention manager balances between decisions complexity against quality and overhead.

Different types of contention management policies can be found in [67–69, 126, 127, 135] like:

1. Passive and Aggressive: Passive contention manager aborts current transaction, while aggressive aborts enemy.

2. Polite: When conflicting on an object, a transaction spins exponensially for average of $2^{(n+k)}\,ns$, where n is number of times to access the object, and k is a tuning parameter. Spinning times is bounded by m. Afterwards, any enemy is aborted.

3. Karma: It assigns priorities to transaction based on the amount of work done so far. Amount of work is measured by number of opened objects by current transaction. Higher priority transaction aborts lower priority one. If lower priority transaction tries to access an object for a number of times greater than priority difference between itself and higher priority transaction, enemy is aborted.

4. Eruption: It works like Karma except it adds priority of blocked transaction to the transaction blocking it. This way, enemy is sped-up, allowing blocked transactions to complete faster.

5. Kindergarten: A transaction maintains a hit list (initially empty) of enemies who previously caused current thread to abort. When a new enemy is encountered, current transaction backs off for a limited amount of time. The new enemy is recorded in the hit list. If the enemy is already in the hit list, it is aborted. If current transaction is still blocked afterwards, then it is aborted.

6. Timestamp: It is a fair contention manager. Each transaction gets a timestamp when it begins. Transaction with newer timestamp is aborted in favour of the older. Otherwise, transaction waits for a fixed intervals, marking the enemy flag as defunct. If the enemy is not done afterwards, it is killed. Active transaction clear their flag when they notice it is set.

7. Greedy: Each transaction is given a timestamp when it starts. The earlier the timestamp of a transaction, the higher its priority. If transaction A conflicts with transaction B, and B is of lower priority or is waiting for another transaction, then A aborts B. Otherwise, A waits for B to commit, abort or starts waiting.

8. Randomized: It aborts current transaction with some probability p, and waits with probability $1 - p$.

9. PublishedTimestamp: It works like Timestamp contention manager except it has a new definition for an “inactive” transaction. Each transaction maintains a “recency” flag. Recency flag is updated every time the transaction makes a request. Each transaction maintains its own “inactivity” threshold parameter that is doubled every time it is aborted up to a specific limit. If the enemy “recency” flag is behind the system global time by amount exceeding its “inactivity” threshold, then enemy is aborted.

10. Polka: It is a combination of Polite and Karma contention managers. Like Karma, it assigns priorities based on amount of job done so far. A transaction backs off for a number of intervals equals difference in priority between itself and its enemy. Unlike Karma, back-off length increases exponentially.

11. Prioritized version of some of the previous contention managers appeared. Prioritized contention managers include base priority of the thread holding the transaction into contention manager policy. This way, higher priority threads are more favoured.

[9] compares performance of different contention managers against an optimal, clairvoyant contention manager. The optimal contention manager knows all resources needed by each transaction, as well as its release time and duration. Comparison is based on the "makespan" concept which is amount of time needed to finish a specific set of transactions. The ratio between makespan of analyzed contention manager and the makespan of the optimal contention manager is known as competitive ratio. [9] proves that any contention manager can be of $O(s)$ competitive ratio if the contention manager is work conserving (i.e., always lets the maximal set of non-conflicting transactions run), and satisfies pending property [68]. The paper proves that this result is asymptotically tight as no on-line work conserving contention manager can achieve better result. [9] also proves that the makespan of greedy contention manager is $O(s)$ instead of $O(s^2)$ [68]. This allows transactions of arbitrary release time and durations in contrast to what is assumed in [68]. For randomized contention managers, a lower bound of $\Omega(s)$ if transaction can modify their resource needs when they are reinvoked.

[67] analyzes different contention managers under different situations. [67] concludes that no single contention manager is suitable for all cases. Thus, [67] proposes a polymorphic contention manager that changes contention managers on the fly throughout different loads, concurrent threads of single load and even different phases of a single thread. To implement polymorphic contention manager, it is important to resolve conflicts resulting from different contention managers in the same application by different methods. The easiest way is to abort the enemy contention manager if it is of different type. [67] uses generic priorities for each transaction regardless of the transaction's contention manager. Upon conflict between different classes of contention manager, highest priority transaction is committed.

[135] provides a comprehensive contention manager attempting to achieve low overhead for low contention, and good throughput and fairness in case of high contention. The main components of comprehensive contention manager are lazy acquisition, extendable timestamp-based conflict detection, and efficient method for capturing conflicts and priorities.

[108] is concerned with implementation issues. [108] considers problems resulting from previous contention management policies like backing off and waiting for time intervals. These strategies make transactions suffer from many aborts that may lead to livelocks, and increased vulnerability to abort because of transactional preemption due to higher priority tasks. Imprecise information and unpredictable benefits resulting from handling long transactions make it difficult to make correct conflict resolution decisions. [108] discriminates be-

tween decisions for long and short transactions, as well as, number of threads larger or lower than number of cores. [108] suggests a number of user and kernel level support mechanisms for contention managers, attempting to reduce overhead in current contention managers' implementations. Instead of spin-locks and system calls, the paper uses shared memory segments for communication between kernel and STM library. It also proposes reducing priority of loser threads instead of aborting them. [108] increases time slices for transactions before they are preempted by higher priority threads. This way, long transactions can commit quickly before they are suspended, reducing abort numbers.

For high number of cores, back-off strategies perform poorly. This is due to hot spots created by small set of conflicts. These hotspots repeat in predictable manner. [19] introduces proactive contention manger that uses history to predict these hotspots and scheduler transactions around them without programmer's input. Proactive contention manager is useful in high contention, but has high cost for low contention. So, [19] uses a hybrid contention managers that begins with back-off strategy for low contention. After a specific threshold for contention level, hybrid contention manager switches to proactive manager.

Contention managers concentrate on preventing starvation through fair policies. They are not suitable for specific systems like real-time systems where stronger behavioural guarantees are required. [66] proposes user-defined priority transactions to make contention management suitable for these specific systems. It investigates the correlation between consistency checking (i.e., finding memory conflicts) and user-defined priority transactions. Transaction priority can be static or dynamic. Dynamic priority increases as abort numbers of transaction increases.

Contention managers are limited in: 1) they are reactive, and suitable only for imminent conflicts. They do not specify when aborted transaction should restart, making them conflict again easily. 2) Contention managers are decentralized because they consume a large part of traffic during high contention. Decentralization prevents global view of the system and limit contention management policy to heuristics. 3) As contention managers are user-level modules, it is difficult to integrate them in HTM. [126] tackles the previous problems by *adaptive transaction scheduling* (ATS). ATS uses contention intensity feedback from the application to adaptively decide number of concurrent transactions running within critical sections. ATS is called only when transaction starts in high contention. Thus, resulting traffic is low and scheduler can be centralized. ATS is integrated into HTM and STM.

[50] presents CAR-STM, a scheduling-based mechanism for STM collision avoidance and resolution. CAR-STM maintains a transaction queue per each core. Each transaction is assigned to a queue by a dispatcher. At the beginning of the transaction, dispatcher uses a conflict probability method to determine the suitable queue for the transaction. The queue with high contention for the current transaction is the most suitable one. All transactions in the same queue are executed by the same thread, thus they are serialized and cannot collide together. CAR-STM uses a serializing contention manager. If one transaction conflicts with another transaction, the former transaction is moved to the queue of the latter. This

prevents further collision between them unless the second transaction is moved to a third queue. Thus, CAR-STM uses another serialization strategy in which the two transactions are moved to the third queue. This guarantees conflict between transactions for at most once.

[111] uses HTM to build single and double linked queue, and limited capacity queue. HTM is used as an alternative synchronization operation to CAS and locks. [111] provides worst case time analysis for the implemented data structures. It experimentally compares the implemented data structures with CAS and lock. [111] reverses the role of TM. Transactions are used to build the data structure, instead of accessing data structures inside transactions. [129] presents an implementation for HTM in a Java chip multiprocessor system (CMP). The used processor is JOP, where worst case execution time analysis is supported.

[14] presents two steps to minimize and limit number of transactional aborts in real-time multiprocessor embedded systems. [14] assumes tasks are scheduled under partitioned EDF. Each task contains at most one transaction. [14] uses multi-versioned STM. In this method, read-only transactions use recent and consistent snapshot of their read sets. Thus, they do not conflict with other transactions and commit on first try. This reduction in abort number comes at the cost of increased memory storage for different versions. [14] uses real-time characteristics to bound maximum number of required versions for each object. Thus, required space is bounded. [14] serializes conflicting transaction in a chronological order. Ties are broken using least laxity and processor identification. [14] does not provide experimental evaluation of its work.

[17] studies the effect of eager versus lazy conflict detection on real-time schedulability. In eager validation, conflicts are detected as soon as they occur. One of the conflicting transactions should be aborted immediately. In lazy validation, conflict detection is delayed to commit time. [17] assumes each task is a complete transaction. [17] proves that synchronous release of tasks does not necessarily lead to worst case response time of tasks. [17] also proves that lazy validation will always result in a longer or equal response time than eager validation. Experiments show that this gap is quite high if higher priority tasks interfere with lower priority ones.

[107]proposes an adaptive scheme to meet deadlines of transactions. This adaptive scheme collects statistical information about execution length of transactions. A transaction can execute in any of three modes depending on its closeness to deadline. These modes are optimistic, visible read and irrevocable. The optimistic mode defers conflict detection to commit time. In visible read, other transactions are informed that a particular location has been read and subject to conflict. Irrevocable mode prevents transaction from aborting. As a transaction gets closer to its deadline, it moves from optimistic to visible read to irrevocable mode. Deadline transactions are supported by the underlying scheduler by disabling preemption for them. Experimental evaluation shows improvement in number of committed transactions without noticeable degradation in transactional throughput.

Previous CMs try to enhance response time of real-time tasks using different policies for conflict resolution. Checkpointing does not require aborted transaction to restart from beginning. Thus, Checkpointing can be plugged into different CMs to further improve response time. [91] introduces checkpointing as an alternative to closed nesting transactions [142]. [91] uses boosted transactions [75] instead of closed nesting [88,117,142] to implement checkpointing. Booseted transactions are based on linearizable objects with abstract states and concrete implementation. Methods under boosted transaction have well defined semantics to transit objects from one state to another. Inverse methods are used to restore objects to previous states. Upon a conflict, a transaction does not need to revert to its beginning, but rather to a point where the conflict can be avoided. Thus, checkpointing enables partial abort. [143] applies checkpointing in distributed transactional memory using Hyflow [123].

Chapter 3

Models and Assumptions

We consider a multicore system with m identical processors and n sporadic tasks $\tau_1, \tau_2, \ldots, \tau_n$. The k^{th} instance (or job) of a task τ_i is denoted τ_i^k. Each task τ_i is specified by its worst case execution time (WCET) c_i, its minimum period T_i between any two consecutive instances, and its relative deadline D_i. We assume implicit deadline systems (i.e., $D_i = T_i$). Job τ_i^j is released at time r_i^j and must finish no later than its absolute deadline $d_i^j = r_i^j + D_i$. The time interval between start and end of τ_i^j is the response time of τ_i^j. Maximum response time of any job of τ_i is upper bounded by R_i^{up}. Under a fixed priority scheduler such as G-RMA, p_i determines τ_i's (fixed) priority and it is constant for all instances of τ_i. Under a dynamic priority scheduler such as G-EDF, τ_i^j's priority, p_i^j, is determined by its absolute deadline. Any job of task τ_j may interfere with a single job of τ_i for a number of times, $g_{ij}^A(L)$, during a duration $L \leq T_i$ under scheduler A. If τ_i and τ_j are independent from each other, then $I_{ij}(L)$ is the amount of time any job of τ_j increases response time of a single job of τ_i during an interval L.

Shared objects. A task may need to access (i.e., read, write) shared, in-memory objects while it is executing any of its atomic sections. Synchronization between different tasks accessing shared objects is done by Software Transactional Memory(STM). Terms "Atomic Section", "Critical Section" and "Transaction" are used interchangeably throughout this document. The set of atomic sections of task τ_i is denoted s_i. $|s_i|$ is number of transactions in τ_i. s_i^k is the k^{th} atomic section of τ_i. s_i^k starts at $S(s_i^k)$. Each object, θ, can be accessed by multiple tasks. The set of distinct objects accessed by any job of τ_i is Θ_i. Θ_i^k is the set of distinct objects accessed by s_i^k. $s_i^k(\Theta)$ states that the s_i^k accesses set of objects $\Theta \subseteq \Theta_i^k$. If more than one atomic section are trying to access the same object(s) at the same time, and at least one access is a "write" operation, then these atomic sections are said to "conflict" together. If s_i^k conflicts with s_j^l and s_i^k starts before s_j^l, then s_i^k is called "interfered" atomic section, and s_j^l is called "interfering" atomic section. The time length of s_i^k is $len(s_i^k)$ if s_i^k runs alone (i.e., s_i^k does not conflict with any other atomic section, nor s_i^k is preempted by any real-time task). To simplify notations, $len(c1.s_{a1}^{b1} + ... + cz.s_{az}^{bz})$ denotes summation of product of constant c_i

by length of atomic section s_{ai}^{bi} where $1 \leq i \leq z$. The maximum-length atomic section in any job of τ_i is $s_{i_{max}}$, whereas the maximum length atomic section among all tasks is s_{max}. The maximum length atomic section in all tasks that accesses any object in Θ is $s_{max}(\Theta)$.

γ_i is the set of tasks that share any object with τ_i. Whereas γ_i^k is the set of tasks sharing objects with s_i^k. Atomic sections are non-nested. When STM is compared against lock-free, each atomic section is assumed to access only one object to allow a head-to-head comparison with lock-free synchronization [49]. Due to "Transitive Retry" (Section 4.1.2), an atomic section s_i^k can conflict with another atomic section s_j^l with no shared objects between s_i^k and s_j^l. While Θ_i contains distinct objects accessed directly by any atomic section in τ_i, Θ_i^* is the set of distinct objects not directly accessed by any atomic section in τ_i. Objects in Θ_i^* can cause one or more atomic sections in τ_i to abort and retry due to transitive retry. Thus, objects in Θ_i^* are indirectly(transitively) accessed by atomic sections in τ_i. $\Theta_i \cap \Theta_i^* = \emptyset$. γ_i^* is the set of tasks, other than τ_i, that access any object in Θ_i^*. Θ_i^{ex} is the extended set of distinct objects accessed directly or indirectly by any atomic section in τ_i. Thus, Θ_i^{ex} is the union of Θ_i and Θ_i^*. γ_i^{ex} is the set of tasks, other than τ_i, that access any object in Θ_i^{ex}. $\Theta_i^{k^{ex}}$ is the subset of objects in Θ_i^{ex} accessed directly or indirectly by s_i^k. $\gamma_i^{k^{ex}}$ is the set of tasks, other than τ_i, that access any object in $\Theta_i^{k^{ex}}$.

STM retry cost. If two or more atomic sections conflict, then a contention manager(CM) resolves the conflict. CM will commit one section and abort and retry the others, increasing the time to execute the aborted sections. Each time s_i^k aborts and retries, $S(s_i^k)$ is updated to the new restart time. The maximum increased time that an atomic section s_i^k will take to execute due to conflict with another section s_j^l, is $W_i^k(s_j^l)$. The maximum time that a task τ_i's atomic sections have to retry due to conflict with any atomic section in tasks other than τ_i over an interval L is $RC_i(L)$. If L is omitted, then $L = T_i$. If i is omitted, then $RC_i(L)$ refers to retry cost of any task τ_i, $1 \leq i \leq n$. $RC_{i_{re}}(L)$, $RC_{i_{re}}$ and RC_{re} are similar to $RC_i(L)$, RC_i and RC, respectively. While RC results from conflict between atomic sections, RC_{re} results from preemption of a job- executing an atomic section- by a higher priority job. The maximum total retry cost by all atomic sections in τ_i during an interval L is the sum of $RC_i(L)$ and $RC_{i_{re}}(L)$. i is omitted if it is known. Priority of atomic sections is one of the used parameters by CMs to resolve conflicts. $p_o(s_i^k)$ is the original priority of atomic section s_i^k. Throughout this document, if s_i^k belongs to job τ_i^j, then $p_o(s_i^k) = p_j^l$. Effective priority (or for simplicity just "priority") of s_i^k is $p(s_i^k)$. $p(s_i^k) = p_o(s_i^k)$ unless $p(s_i^k)$ changes by the underlying synchronization technique. $max\_s_{ik}^{jl}(\theta)$ is the maximum length atomic section in all tasks that accesses θ and its priority is lower than $p(s_j^l)$ and higher than $p(s_i^k)$. Whereas $max\_s_{ik}^{jl}(\Theta)$ is the maximum length $max\_s_{ik}^{jl}(\theta)$ that access any object $\theta \in \Theta$. Previous notations are summarized in Table 3.1.

Table 3.1: Notations

General notations	
TM	Transactional Memory.
CM	Contention Manager.
STM	Software Transactional Memory.
HTM	Hardware Transactional Memory.
ECM	Earliest Deadline Contention Manager.
RCM	Ratemonotonic Contention Manager.
LCM	Length-based Contention Manager.
PNF	Priority with Negative value and First access contention manager.
FBLT	First Bounded, Last Timestamp contention manager.
CPFBLT	Checkpointing First Bounded, Last Timestamp contention manager.
G-EDF	Global Earliest Deadline First scheduler.
RMS	Rate-Monotonic Scheduling.
G-RMA	Global Rate-Monotonic Scheduler.
Real-time task's notations	
τ_i	i^{th} task in the task set.
τ_i^j	j^{th} instance (job) of the i^{th} task. τ_i is used to generally represent any instance τ_i^j (j is indeterminate).
c_i	Worst case execution time (WCET) of any instance of τ_i.
T_i	Minimum period between any two consequitive instances of τ_i.
D_i	Relative deadline of any instance of τ_i. In case of implicit deadline system, $D_i = T_i$.
r_i^j	Release time of job τ_i^j.
d_i^j	Absolute deadline of job τ_i^j ($d_i^j = r_i^j + D_i$). d_i^j is also the absolute deadline of any atomic section (transaction) in τ_i^j.
p_i^j, p_i	p_i^j is priority of job τ_i^j. If p_i^j is fixed for all jobs of τ_i, then $p_i^j = p_i$, $\forall j$.
$g_{ij}^A(L)$	Maximum number of interferences made by any job τ_j^y to only one job τ_i^x during an interval L, where $L \leq T_i$, under scheduling algorithm A.
m	Number of processors in a multiprocessor systems.
n	Number of tasks in a set of sporadic tasks.
Atomic section's (transaction's) notations	
θ, Θ	θ is one object that can be accessed within an atomic section (transaction) of any task. Whereas Θ is a set of objects.
s_i	Set of atomic sections (transactions) in any job of τ_i.
s_i^k	The k^{th} atomic section (transaction) in any job of τ_i.

$p_o(s_i^k)$, $p(s_i^k)$	$p_o(s_i^k)$ is the orginal priority of atomic section (transaction) s_i^k. Throughout this document, if s_i^k belongs to job τ_i^j, then $p_o(s_i^k) = p_j^l$. $p(s_i^k) = p_o(s_i^k)$ unless priority of s_i^k changes due to behaviour of underlying synchronization technique.
Θ_i	Set of distinct objects accessed directly by any job of τ_i.
Θ_i^k	The set of distinct objects accessed directly by s_i^k. $\Theta_i^k \subseteq \Theta_i$.
$s_i^k(\Theta)$	The same as s_i^k. $s_i^k(\Theta)$ states that s_i^k accesses a set of objects $\Theta \subseteq \Theta_i^k$.
$\Theta_i^k(\theta_a)$	Set of distinct objects accessed by s_i^k for the first time after s_i^k's first access to θ_a. $\Theta_i^k(\theta_a) \subseteq \Theta_i^k$.
$len\left(s_i^k\right)$	Time length of s_i^k in absence of conflict with any other atomic section and preemption by any real-time job.
$len\left(\sum_{i=1}^{z} c_i s_{ai}^{bi}\right)$	Summation of the product of constant c_i by the length of transaction s_{ai}^{bi} where $1 \leq i \leq z$.
γ_i	Set of tasks other than τ_i that access at least one object in Θ_i.
γ_i^k	Set of tasks that share any object with s_i^k. Each task in γ_i has direct access to at least one object in Θ_i^k.
$s_{i_{max}}$	The maximum length atomic section (transaction) in any job of τ_i.
$s_{max}(\Theta)$	The maximum length atomic section (transaction) in all tasks that accesses any object $\theta \in \Theta$.
s_{max}	Length of the longest atomic section (transaction) in all tasks.
$S(s_i^k)$	Start time of s_i^k. $S(s_i^k)$ is updated each time s_i^k aborts and retries.
$max\_s_{ik}^{jl}(\theta)$	The maximum length atomic section (transaction) in all tasks that accesses θ and its priority is lower than $p(s_j^l)$ and higher than $p(s_i^k)$.
$max\_s_{ik}^{jl}(\Theta)$	$max\{max\_s_{ik}^{jl}(\theta) : \forall \theta \in \Theta\}$.
$W_i^k(s_j^l)$	The maximum time s_i^k aborts and retries due to a conflict with s_j^l. If s_i^k was executing before s_j^l was released, then s_i^k is called "interfered transaction", whereas "s_j^l" is called "interfering transaction".
Θ_i^*	Set of distincit objects not accessed directly by transactions in τ_i, but can cause transactions in τ_i to retry. While Θ_i respresent "direct objects" accessed by τ_i, Θ_i^* represent "indirect objects" accessed by transactions in τ_i. $\Theta_i \cap \Theta_i^* = \emptyset$.
Θ_i^{ex}	Extended set of objects accessed "directly" or "indirectly" by τ_i (i.e., $\Theta_i^{ex} = \Theta_i \cup \Theta_i^*$).
$\Theta_i^{k^{ex}}$	Subset of objects in Θ_i^{ex} that are accessed directly or indirectly by s_i^k.
γ_i^*	Set of tasks, other than τ_i, that access at least one object in Θ_i^*.
γ_i^{ex}	Set of tasks, other than τ_i, that access at least one object in Θ_i^{ex}.

$\gamma_i^{k^{ex}}$	Set of tasks, other than τ_i, that access objects in $\Theta_i^{k^{ex}}$.
$RC_i(L)$, RC_i, RC	$RC_i(L)$ is maximum retry cost of any job in τ_i due to conflict between transactions in τ_i and transactions in other tasks during an interval L. If L is omitted, then $L = T_i$. If i is omitted, then i can be any task (i.e., $i \leq n$).
$RC_{i_{re}}(L)$, $RC_{i_{re}}$, $RC_{re}(L)$	$RC_{i_{re}}(L)$ is maximum retry cost of any job in τ_i due to release of higher priority jobs of tasks other than τ_i during an interval L. If L is omitted, then $L = T_i$. If i is omitted, then i can be any task (i.e., $i \leq n$).
$RC_{i_{to}}(L)$, $RC_{to}(L)$	Maximum total retry cost of any job in τ_i during an interval L (i.e., $RC_{i_{to}}(L) = RC_i(L) + RC_{i_{re}}(L)$). If i is known, then i is omitted.
$I_{ij}(L)$	In absence of retry cost, $W_{ij}(L)$ is the amount of time any job of τ_j increases response time of any job of τ_i.
$cp_i^k(\theta)$	Recorded checkpoint in s_i^k for the newly accessed object θ.
∇_{i*}^k	Time interval between start of s_i^k and the first access to the first shared object between s_i^k and any other transaction.
t_{cp}^c, t_{cp}^r	Time cost to construct and remove a single checkpoint.
r_{max}	Maximum execution cost of a single iteration of any retry-loop lock-free of any task.

Chapter 4

The ECM and RCM Contention Managers

We consider software transactional memory (STM) for concurrency control in multicore embedded real-time software. We investigate real-time contention managers (CMs) for resolving transactional conflicts, including those based on dynamic and fixed priorities, and establish upper bounds on transactional retries and task response times. We identify the conditions under which STM (with the proposed CMs) is superior to lock-free synchronization [49] and real-time locking protocols (i.e., OMLP [22, 29] and RNLP [149]).

The rest of this Chapter is organized as follows, Section 4.1 investigates Earliest Deadline Contention Manager under G-EDF scheduling (ECM) and illustrates its behaviour. We provide computations of workload interference and retry cost analysis under ECM. Section 4.2 presents Rate Monotonic Contention Manager under G-RMA scheduling (RCM). It also includes retry cost and response time analysis under RCM. Section 4.3 compares performance between any two synchronization techniques in terms of total utilization. Total utilization of ECM and RCM is compared against total utilization of lock-free in Section 4.4 and real-time locking protocols in Section 4.5. We conclude the Chapter in Section 4.6.

4.1 ECM

Since only one atomic section among many that share the same object can commit at any time under STM, those atomic sections execute in sequential order. A task τ_i's atomic sections are interfered by other tasks that share the same objects with τ_i. Hereafter, we will use *ECM* to refer to a multicore system scheduled by G-EDF and resolves STM conflicts using the EDF CM. ECM was originally introduced in [60] and analyzed in more details in [54]. ECM will abort and retry an atomic section of τ_i^x, s_i^k due to a conflicting atomic section of τ_j^y, s_j^l, if the absolute deadline of τ_j^y is less than or equal to the absolute deadline of τ_i^x. ECM behaviour is shown in Algorithm 1. [60] assumes the worst case scenario for transactional retry occurs when conflicting transactions are released simultaneously. [60] also assumes all transactions have the same length. Here, we extend the analysis in [60] to a more worse conflicting scenario and consider distinct-length transactions.

Algorithm 1: ECM

```
Data: s_i^k → interfered atomic section. s_j^l → interfering atomic section
Result: which atomic section aborts
if d_i^k < d_j^l then
    s_j^l aborts;
else
    s_i^k aborts;
end
```

4.1.1 Illustrative Example

Behaviour of ECM can be illustrated by the following example:

- Transaction $s_i^k \in \tau_i^x$ begins execution. Currently, s_i^k does not conflict with any other transaction.
- Transaction $s_j^l \in \tau_j^y$ is released while s_i^k is still running. $\Theta_i^k \cap \Theta_j^l \neq \emptyset$. $d_j^y < d_i^x$. So, $p_j^y > p_i^x$. Hence, ECM will abort and restart s_i^k in favour of s_j^l.
- Transaction $s_h^v \in \tau_h^u$ is released while s_j^l is still running. $d_h^u < d_j^y < d_i^x$. So, $p_h^u > p_j^y > p_i^x$. s_j^l and s_i^k will abort and retry until s_h^v commits.
- s_h^v commits. s_j^l executes while s_i^k aborts and retries.
- s_j^l commits. s_i^k executes.

4.1.2 Transitive Retry

With multiple objects per transaction, ECM will face transitive retry, which we illustrate with an example.

Example 1. Consider three atomic sections s_1^x, s_2^y, and s_3^z belonging to jobs τ_1^x,τ_2^y, and τ_3^z, with priorities $p_3^z > p_2^y > p_1^x$, respectively. Assume that s_1^x and s_2^y share objects, s_2^y and s_3^z share objects. s_1^x and s_3^z do not share objects. s_3^z can cause s_2^y to retry, which in turn will cause s_1^x to retry. This means that s_1^x may retry transitively because of s_3^z, which will increase the retry cost of s_1^x.

Assume another atomic section s_4^f is introduced. Priority of s_4^f is higher than priority of s_3^z. s_4^f shares objects only with s_3^z. Thus, s_4^f can make s_3^z to retry, which in turn will make s_2^y to retry, and finally, s_1^x to retry. Thus, transitive retry will move from s_4^f to s_1^x, increasing the retry cost of s_1^x. The situation gets worse as more tasks of higher priorities are added, where each task shares objects with its immediate lower priority task. τ_3^z may have atomic sections that share objects with τ_1^x, but this will not prevent the effect of transitive retry due to s_1^x.

Definition 1. ***Transitive(indirect) Retry:*** *A transaction s_i^k suffers from transitive retry when s_i^k retries due to a higher priority transaction s_z^h, and $\Theta_z^h \cap \Theta_i^k = \emptyset$.*

Claim 1. *ECM suffers from transitive retry for multi-object transactions.*

Proof. ECM depends on priorities to resolve conflicts between transactions. Thus, lower priority transaction must always be aborted for a conflicting higher priority transaction. Claim follows. □

Because of transitive retry, Θ_i for any τ_i is extended to include any object $\theta \notin \Theta_i$, but θ can make at least one transaction $s_i^k \in \tau_i$ retry transitively. The new set of objects that can cause direct or indirect retry of at least one transaction in τ_i is denoted as Θ_i^{ex}. Θ_i^{ex} is obtained by being initialized to Θ_i (i.e., the set of objects that are already accessed by any transaction $s_i^k \in \tau_i$). We then cycle through all transactions belonging to all other higher priority tasks. Each transaction s_j^l that accesses at least one of the objects in Θ_i^{ex} adds all other objects accessed by s_j^l to Θ_i^{ex}. The loop over all higher priority tasks is repeated, each time with the new Θ_i^{ex}, until there are no more transactions accessing any object in Θ_i^{ex}. However, this solution may over-extend the set of conflicting objects, and may even contain all objects accessed by all tasks. Θ_i^* represent the set of objects not accessed directly by any transaction in τ_i, but any $\theta \in \Theta_i^*$ can make at least one transaction in τ_i retry transitively. Thus, $\Theta_i^{ex} = \Theta_i + \Theta_i^*$. Similarly, the distinct set of objects that can make s_i^k retry directly or indirectly(transitively) is denoted as $\Theta_i^{k^{ex}}$. γ_i is the extended to γ_i^{ex}. While γ_i is the set of tasks- other than τ_i- that access at least one object $\theta \in \Theta_i$, γ_i^{ex} is the set of tasks- other than τ_i- that access at least one object $\theta \in \Theta_i^{ex}$.

4.1.3 G-EDF Interference

Claim 2. *Regardless of the used scheduler, maximum number of jobs of τ_j that can exist in time interval L is upper bounded by*

$$\left\lceil \frac{L}{T_j} \right\rceil + 1 \tag{4.1}$$

where at most two jobs τ_j can be partially included in L. The remaining jobs of τ_j are totally included in L.

Proof. Generally, $L = aT_j + b$, $0 \leq b < T_j$. a is the maximum number of jobs of τ_j that contribute by their minimum periods T_j during L. If $b \geq T_j$, then there are more than a jobs of τ_j contributing by their minimum periods T_j during L, which contradicts definition of a. The remaining interval $b(= L - aT_j,\, b > 0)$ can be divided between at most two jobs of τ_j. If b can be divided between more than two jobs of τ_j, then there is more than a jobs of τ_j that contribute by their minimum periods T_j during L. This contradicts definition of a. So, if $b > 0$, then maximum number of jobs of τ_j that can exist during L is $a + 2 = \left\lceil \frac{L}{T_j} \right\rceil + 1$.

If $b = 0$, then jobs of τ_j can be shifted to the left or the right during L. This results in $a+1$ jobs of τ_j during L. So, if $b = 0$, then maximum number of jobs of τ_j that can exist during L is $a + 1 = \left\lceil \frac{L}{T_j} \right\rceil + 1$. Claim follows. □

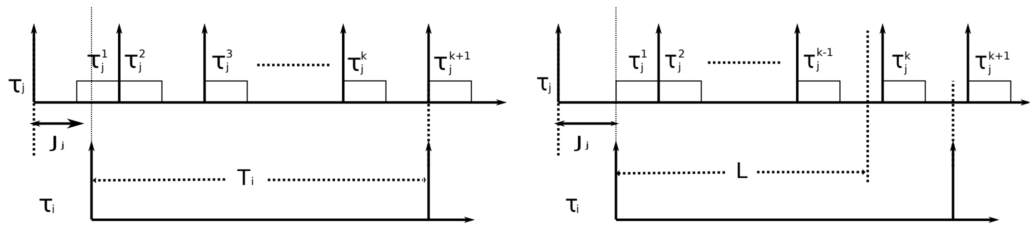
Claim 3. *Let $T_i = aT_j + b$, where $a = \left\lfloor \frac{T_i}{T_j} \right\rfloor$ and $0 \leq b < T_j$. Under G-EDF scheduler, maximum number of jobs of τ_j that can interfere with one job τ_i^x during time interval $L(= T_i - f$, $0 \leq f < T_i)$ is*

$$g_{ij}^{gedf}(L) = \begin{cases} \left\lceil \frac{T_i}{T_j} \right\rceil & , f \leq b \\ \\ \left\lceil \frac{L}{T_j} \right\rceil + 1 & , Otherwise \end{cases} \tag{4.2}$$

Proof. $L = T_i - f = aT_j + b - f$. If $b - f \geq 0$, then following proof of Claim 2, $b - f$ can be devided between at most two jobs of τ_j during L. These two jobs of τ_j are: 1) *carried-in job* (i.e., τ_j^s where $r_j^s < r_i^x$ and $d_j^s < d_i^x$ [18]). 2) *carried-out job* (τ_j^e where $r_j^e > r_i^x$ and $d_j^s > d_i^x$ [18]). Under G-EDF, only jobs of τ_j with absolute deadline less than d_i^x can interfere with τ_i^x. Thus, carried-out job of τ_j cannot interfere with τ_i^x. So, $b - f$ can be the contribution of only the carried-in job. Following proof of Claim 2, maximum number of jobs of τ_j that can interfere with τ_i^x is $a + 1 = \left\lceil \frac{T_i}{T_j} \right\rceil$ if $f \leq b$. Otherwise, Claim 2 is used to determine maximum number of jobs of τ_j during L. Claim follows. □

The maximum number of times a task τ_j interferes with τ_i under G-EDF is illustrated in Figure 4.1. Upper bound on maximum interference of τ_j to τ_i (when there are no atomic sections) in $L \leq T_i$ is given in [18]. It should be noted that we consider only implicit deadline systems (i.e., $\forall \tau_i$, $T_i = D_i$). Implicit deadline system is a special case of constrained deadline system (i.e., $\forall \tau_i$, $D_i \leq T_i$) considered by [18]. The interference of τ_j to τ_i during $L = T_i - f$ where $f \leq b$ (as shown in Fig 4.1(a)), in the absence of atomic sections, is upper bounded by:

$$\begin{aligned} I_{ij}^1(T_i) &\leq \left\lfloor \frac{T_i}{T_j} \right\rfloor c_j + min\left(c_j, T_i - \left\lfloor \frac{T_i}{T_j} \right\rfloor T_j\right) \\ &\leq \left\lceil \frac{T_i}{T_j} \right\rceil c_j \end{aligned} \tag{4.3}$$



(a) during $L = T_i - f,\ f \leq b$ (b) during $L = T_i - f,\ f > b$

Figure 4.1: Maximum interference of jobs of τ_j to τ_i^x running on different processors, under G-EDF. $T_i = aT_j + b$

The interference of τ_j to τ_i during an interval $L = T_i - f$ where $f > b$, as shown in Fig 4.1(b), in the absence of atomic sections is upper bounded by:

$$I_{ij}^2\left(L\right) \leq \left(\left\lceil \frac{L - c_j}{T_j} \right\rceil + 1\right) c_j \tag{4.4}$$

Here, τ_j^1 contributes by all its c_j, and d_j^{k-1} does not have to coincide with L, as τ_j^{k-1} has a higher priority than that of τ_i. Thus, the overall interference of τ_j to τ_i, over an interval $L \leq T_i$ is:

$$I_{ij}\left(L\right) = min\left(I_{ij}^1\left(T_i\right), I_{ij}^2\left(L\right)\right) \tag{4.5}$$

[18] upper bounds maximum response time of any job of τ_i. Upper bound on maximum response time of any job of τ_i is calculated by iteration of (4.6), starting from $R_i^{up} = c_i$.

$$R_i^{up} = c_i + \left\lfloor \frac{1}{m} \sum_{j \neq i} I_{ij}\left(R_i^{up}\right) \right\rfloor \tag{4.6}$$

where $I_{ij}\left(R_i^{up}\right)$ is calculate by (4.5).

4.1.4 Retry Cost of Atomic Sections

Claim 4. *Let s_i^k and s_j^l be two conflicting transactions. s_i^k has a lower priority than s_j^l. Let the lower priority transaction always aborts and retries due to the higher priority transaction. s_j^l interfere only once with s_i^k. s_i^k aborts and retries due to s_j^l for at most*

$$len\left(s_i^k + s_j^l\right) \tag{4.7}$$

Proof. s_j^l must start at least when s_i^k starts and not later than s_i^k finishes. Otherwise, there will be no conflict between s_i^k and s_j^l. s_i^k must retry during execution of s_j^l because of higher priority of s_j^l. The part of s_i^k that started before beginning of s_j^l will be repeated. Thus, the worst case interference between s_i^k and s_j^l occurs when s_j^l starts just when s_i^k reaches its end of execution. So, maximum retry cost of s_i^k due to s_j^l is calculated by 4.7. Claim follows. □

Claim 5. *Let conflict between transactions be resolved by priority (i.e., lower priority transaction aborts and retries due to higher priority transactions). Let* $conf\left\{s_i^k\right\}$ *be the set of all transactions that do not belong to any job of* τ_i *and are conflicting, directly or indirectly(transitively), with* s_i^k*. Each transaction* $s_j^l \in conf\left\{s_i^k\right\}$ *contributes to the retry cost of* s_i^k *by at most*

$$len\left(s_j^l + max\_s_{ik}^{jl}(\Theta)\right) \tag{4.8}$$

where $max\_s_{ik}^{jl}(\Theta)$ *is the maximum length atomic section (transaction) in* $conf\{s_i^k\}$ *that accesses* Θ *and its priority is lower than* $p(s_j^l)$ *and higher than or equal to* $p(s_i^k)$*.* $max\_s_{ik}^{jl} \notin s_j$ *and* $\Theta \subseteq \Theta_i^{k^{ex}} \cap \Theta_j^l$*.*

Proof. As conflict is resolved by transactional priority, then only transactions with higher priorities than $p(s_i^k)$ will cause s_i^k to abort and retry. Also, s_j^l will abort only transactions with lower priority than $p(s_j^l)$. As transactions that belong to the same job execute sequentially, and jobs of the same task execute sequentially , so s_i^k is not aborted by other transactions that belong to τ_i. So, at any point of time after s_i^k was first released, and before the last successful run of s_i^k (i.e., the run at which s_i^k commits), one of the following cases happens:

1. s_j^l has finished before s_i^k starts. Or, s_j^l starts after s_i^k finishes. In this case, s_j^l will not cause s_i^k to abort and retry. (4.8) still upper bounds effect of s_j^l to the retry cost of s_i^k.

2. s_j^l is the only transaction that is currently aborting s_i^k. So, (4.8) follows directly from Claim 4 as $len\left(s_i^k\right) \leq len\left(max\_s_{ik}^{jl}(\Theta)\right)$.

3. A set of transactions $S \subseteq conf\{s_i^k\}$ are currently aborting s_i^k. $s_j^l \in S$ and s_j^l itself is not aborting and retrying due to any other transaction with higher priority than $p(s_j^l)$. So, s_j^l executes only once. s_j^l aborts one of the transactions with lower priority than $p(s_j^l)$ for only once. Thus, (4.8) upper bounds effect of s_j^l to the retry cost of s_i^k.

4. A set of transactions $S \subseteq conf\{s_i^k\}$ are currently aborting s_i^k. $s_j^l \in S$ and s_j^l itself is aborting and retrying due to other transactions with higher priority than $p(s_j^l)$. Without losing generality, let s_h^u be the transaction that is currently aborting s_j^l, and s_h^u is not aborting and retrying due to any other higher priority transaction. Then, s_j^l and s_i^k are both waiting for s_h^u to finish. Thus, the time of retrial of s_j^l due to s_h^u is covered by effect of s_h^u to the retry cost of s_i^k. When s_h^u finishes and s_j^l is not aborted by any other higher priority transaction, effect of s_j^l to the retry cost of s_i^k is the same as in the third case. By expanding this case to more than three transactions, then each transaction s_j^l is either aborting one of the lower priority transactions only once (i.e., the last successful run of s_j^l), or s_i^k and s_j^l are aborted by a higher priority transaction s_h^u. When s_j^l is retrying due to the higher priority transaction s_h^u, s_j^l retrial time is not considered in retry cost of s_i^k because it is already covered by the effect of the higher priority transaction s_h^u to the retry cost of s_i^k.

Claim follows. □

Claim 6. *Under ECM, the total retry cost suffered by all transactions in any job $\tau_i^x \in \tau_i$ during interval $L \leq T_i$ due to direct and indirect conflict with other transactions in jobs with higher priority than τ_i^x is upper bounded by:*

$$RC_i(L) \leq \sum_{\tau_j \in \gamma_i^{ex}} \left(g_{ij}^{gedf} \sum_{\forall s_j^l,\, (\Theta = \Theta_j^l \cap \Theta_i^{ex}) \neq \emptyset} len\left(s_j^l + s_{max}(\Theta)\right)\right) \tag{4.9}$$

where $s_{max}(\Theta) \notin s_j$ and g_{ij}^{gedf} is calculated by (4.2).

Proof. ECM is used with G-EDF scheduler. Thus, $p(s_i^k)$ is a dynamic priority that depends on the absolute deadline of containing job τ_i^x. So, $conf\left\{s_i^k\right\}$ for any s_i^k includes each transaction $s_j^l \notin s_i$ where $\Theta_j^l \cap \Theta_i^{k^{ex}} \neq \emptyset$. The worst case retry cost of any s_i^k occurs when $p(s_i^k)$ is the lowest priority among all other conflicting transactions during T_i. g_{ij}^{gedf} is the maximum number of jobs of $\tau_j \in \gamma_i^{ex}$ that can interfere with one job of τ_j. Following Claims 3, 4 and 5, Claim follows. □

Claim 7. *Under ECM, upper bound on total retry cost given by (4.10) can be tightened by considering carried_in job of each τ_j (i.e., τ_j^{in} where $r_j^{in} < r_i^x$ and $d_j^{in} < d_i^x$ as defined in [18]) conflicting with τ_i^x during interval $L = T_i - f$, where $T_i = aT_j + b$, $a = \left\lfloor \frac{T_i}{T_j} \right\rfloor$ and $f \leq b$. (4.10) will be modified to*

$$RC_i(L) \leq \begin{cases} \sum_{\tau_j \in \gamma_i^{ex}} \left(\lambda_1(j) + \chi(i,j)\right) & , f \leq b \\ \sum_{\tau_j \in \gamma_i^{ex}} \left(\left(\left\lceil \frac{L}{T_j} \right\rceil + 1\right) \sum_{\forall s_j^l, \left(\Theta = \Theta_i^{ex} \cap \Theta_j^l\right) \neq \emptyset} len\left(s_j^l + s_{max}(\Theta)\right)\right) & , Otherwise \end{cases} \tag{4.10}$$

where

- $s_{max} \notin s_j$.
- $\lambda_1(j) = \sum_{\forall s_j^l \in \left[d_j^{in} - \delta, d_j^{in}\right], \left(\Theta = \Theta_i^{ex} \cap \Theta_j^l\right)} len\left(s_j^{l*} + s_{max}(\Theta)\right)$, *where $\delta = min(c_j, b)$ and s_j^{l*} is the part of s_j^l that is contained in interval $\left[d_j^{in} - \delta, d_j^{in}\right]$.*
- $\chi(i,j) = \left\lfloor \frac{T_i}{T_j} \right\rfloor \sum_{\forall s_j^l, \left(\Theta = \Theta_i^{ex} \cap \Theta_j^l\right) \neq \emptyset} len\left(s_j^l + s_{max}(\Theta)\right)$.

Proof. Following proof of Claim 3, maximum number of jobs of τ_j that can interfere with τ_i^x is $\left\lceil \frac{T_i}{T_j} \right\rceil$. By definition of carried-in jobs [18] and G-EDF scheduler, there will be $\left\lfloor \frac{T_i}{T_j} \right\rfloor$ jobs of τ_j that exist by their whole periods T_j in the interval L. Carried-in job of τ_j (i.e., τ_j^{in}) will exist by at most $\delta = min(c_j, b)$ during L. τ_j^{in} is delayed by its maximum jitter to give its maximum contribution during L. Thus, τ_j^{in} starts execution at $d_j^{in} - c_j$. Consequently, only transactions of τ_j^{in} that are contained in $\left[d_j^{in} - \delta, d_j^{in}\right]$ can exist in the interval L. Also, if transaction s_j^l is partially contained in $\left[d_j^{in} - \delta, d_j^{in}\right]$, only the part of s_j^l contained in

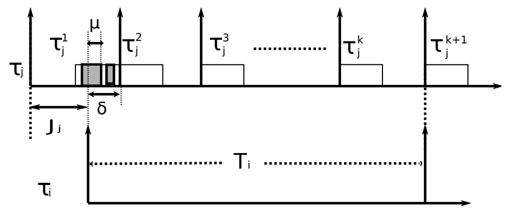


Figure 4.2: Effect of carried_in job of τ_j to retry cost of transactions in τ_i

$\left[d_j^{in}-\delta, d_j^{in}\right]$ (i.e., $s_j^{l^*}$) can conflict with transactions in τ_i^x. $\lambda(j)$ stands for the retry cost of transactions in τ_i^x due to conflict with transactions of τ_j^{in}. Whereas, $\chi(i,j)$ stands for the retry cost of transactions in τ_i^x due to conflict with transactions of other jobs of τ_j (i.e., non carried-in jobs). Combining the previous notions with Claim 6, Claim follows. □

Effect of transactions in carried_in job is shown in Figure 4.2. There are two sources of retry cost for any τ_i^x under ECM. First is due to conflict between τ_i^x's transactions and transactions of other jobs. This is denoted as RC_i. Second is due to the preemption of any transaction in τ_i^x due to the release of all higher priority jobs. This is denoted as $RC_{i_{re}}$. It is up to the implementation of the contention manager to avoid RC_{re}. Here, as we are concerned with maximum total retry cost introduced by ECM, we assume that ECM does not avoid RC_{re}. Thus, we introduce RC_{re} for ECM technique.

Claim 8. *Under ECM, the total retry cost suffered by all transactions in any job $\tau_i^x \in \tau_i$ during an interval $L \le T_i$ due to release of jobs with higher priority than τ_i^x is upper bounded by*

$$RC_{i_{re}}(L) \le \sum_{\forall \tau_j \in \zeta_i} \begin{cases} \left\lceil \frac{L}{T_j} \right\rceil s_{i_{max}} & ,L \le T_i - T_j \\ \\ \left\lfloor \frac{T_i}{T_j} \right\rfloor s_{i_{max}} & ,L > T_i - T_j \end{cases} \tag{4.11}$$

where $\zeta_i = \{\tau_j : (\tau_j \ne \tau_i) \wedge (D_j < D_i)\}$.

Proof. Two conditions must be satisfied for any τ_j^l to be able to preempt τ_i^x under G-EDF: $r_i^x < r_j^l < d_i^x$, and $d_j^l \le d_i^x$. Without the first condition, τ_j^l would have been already released before τ_i^x. Thus, τ_j^l will not preempt τ_i^x. Without the second condition, τ_j^l will be of lower priority than τ_i^x and will not preempt it. If $D_j \ge D_i$, then there will be at most one instance τ_j^l with higher priority than τ_i^x. τ_j^l must have been released at most at r_i^x, which violates the first condition. The other instance τ_j^{l+1} would have an absolute deadline greater than d_i^x. This violates the second condition. Hence, only tasks with shorter relative deadline than D_i are considered. These jobs are grouped in ζ_i.

The total number of released instances of τ_j during any interval $L \le T_i$ is $\left\lceil \frac{L}{T_i} \right\rceil + 1$. The "carried-in" jobs (i.e., each job released before r_i^x and has an absolute deadline before d_i^x [18])

are discarded as they violate the first condition. The "carried-out" jobs (i.e., each job released after r_i^x and has an absolute deadline after d_i^x [18]) are also discarded because they violate the second condition. Thus, the number of considered higher priority instances of τ_j during the interval $L \leq T_i - T_j$ is $\left\lceil \frac{L}{T_j} \right\rceil$. The number of considered higher priority instances of τ_j during interval $L > T_i - T_j$ is $\left\lfloor \frac{T_i}{T_j} \right\rfloor$.

The worst $RC_{i_{re}}$ for τ_i^x occurs when τ_i^x is always interfered at the end of execution of its longest atomic section, $s_{i_{max}}$. τ_i^x will have to retry for $len(s_{i_{max}})$. Claim follows. □

Claim 9. *Under ECM, the total retry cost suffered by all transactions in any job* $\tau_i^x \in \tau_i$ *during an interval* $L \leq T_i$ *is upper bounded by:*

$$RC_{i_{to}}(L) = RC_i(L) + RC_{i_{re}}(L) \tag{4.12}$$

where $RC_i(L)$ *is the maximum retry cost resulting from conflict between transactions in* τ_i^x *and transactions of other jobs.* $RC_i(L)$ *is calculated by (4.10).* $RC_{i_{re}}(L)$ *is the maximum retry cost resulting from the release of higher priority jobs, which preempt transactions in* τ_i^x*.* $RC_{i_{re}}(L)$ *is calculated by (4.11).*

Proof. Under ECM, transactions in any job $\tau_i^x \in \tau_i$ retry due to: 1) conflicting transactions of jobs with higher priority than τ_i^x. 2) release of higher priority jobs that preempt τ_i^x. Thus, (4.12) follows directly from Claims 7 and 8. Claim follows. □

4.1.5 Upper Bound on Response Time

Claim 10. *Under ECM, maximum response time of any job* $\tau_i^x \in \tau_i$ *is upper bounded by*

$$R_i^{up} = c_i + RC_{i_{to}}(R_i^{up}) + \left\lfloor \frac{1}{m} \sum_{j \neq i} I_{ij}(R_i^{up}) \right\rfloor \tag{4.13}$$

where:

- R_i^{up}*'s initial value is* $c_i + R_i^{up}(c_i)$.
- $RC_{i_{to}}(R_i^{up})$ *is calculated by (4.12).*
- c_j *of any job* $\tau_j^y \in \tau_j$ *with* $p_j^y > p_i^x$ *is modified to*

$$c_{ji} = c_j - \left(\sum_{s_j^l, \left(\Theta = \Theta_i^{ex} \cap \Theta_j^l\right) \neq \emptyset} len\left(s_j^l\right) \right) + RC_{ji_{to}}(R_i^{up}) \tag{4.14}$$

- $RC_{ji_{to}}(R_i^{up})$ *is the same as* $RC_{j_{to}}(R_i^{up})$ *excluding atomic sections in* τ_j *that access shared objects between* τ_i *and* τ_j*.* τ_i *does not contribute to* $RC_{j_{re}}(R_i^{up})$.

- *$I_{ij}(R_i^{up})$ is calculated by (4.5) with c_j replaced by c_{ji} and changing (4.4) to*

$$I_{ij}(R_i^{up}) = max\begin{cases}\left(\left\lceil\frac{R_i^{up}-\left(c_{ji}+\sum_{s_j^l,\left(\Theta=\Theta_i^{ex}\cap\Theta_j^l\right)\neq\emptyset}len(s_j^l)\right)}{T_j}\right\rceil+1\right)c_{ji}\\ \left\lceil\frac{R_i^{up}-c_j}{T_j}\right\rceil.c_{ji}+c_j-\sum_{s_j^l,\left(\Theta=\Theta_i^{ex}\cap\Theta_j^l\right)\neq\emptyset}len(s_j^l)\end{cases} \tag{4.15}$$

Proof. To obtain an upper bound on the maximum response time (i.e., R_i^{up}) of any job τ_i^x of τ_i, the term $RC_{i_{to}}(R_i^{up})$ must be added to the interference of other tasks during the non-atomic execution of τ_i^x. But this requires modification of the WCET of each task as follows.

c_j of each interfering task τ_j should be inflated to accommodate the interference of each task τ_k, $k \neq j, i$. Meanwhile, atomic regions that access shared objects between τ_j and τ_i should not be considered in the inflation cost, because they have already been calculated in τ_i's retry cost. As an upper bound on R_i^{up} is calculated, then jobs of τ_j with higher priority than τ_i^x are only considered. Thus, τ_i^x has no contribution in $RC_{j_{re}}(R_i^{up})$. Thus, τ_j's inflated WCET becomes:

$$c_{ji} = c_j - \left(\sum_{s_j^l,\left(\Theta=\Theta_i^{ex}\cap\Theta_j^l\right)\neq\emptyset} len\left(s_j^l\right)\right) + RC_{ji_{to}}(R_i^{up})$$

which is given by (4.14). c_{ji} is the new WCET of τ_j relative to τ_i. $\sum_{s_j^l,\left(\Theta=\Theta_i^{ex}\cap\Theta_j^l\right)\neq\emptyset}len(s_j^l)$ is the sum of lengths of all atomic sections in τ_j that access any object $\theta \in \Theta_i^{ex}$. $\sum_{s_j^l,\left(\Theta=\Theta_i^{ex}\cap\Theta_j^l\right)\neq\emptyset}$ $len(s_j^l)$ is subtracted from c_j because $\sum_{s_j^l,\left(\Theta=\Theta_i^{ex}\cap\Theta_j^l\right)\neq\emptyset}len(s_j^l)$ is already included in $RC_{i_{to}}(R_i^{up})$. $RC_{ji_{to}}(R_i^{up})$ is the $RC_{j_{to}}(R_i^{up})$ without including the shared objects between τ_i and τ_j. The calculated WCET is relative to task τ_i, as it changes from task to task. The upper bound on the response time of τ_i^x, denoted R_i^{up}, can be calculated iteratively, by modifying (4.6), as follows:

$$R_i^{up} = c_i + RC_{i_{to}}(R_i^{up}) + \left\lfloor\frac{1}{m}\sum_{j\neq i} I_{ij}(R_i^{up})\right\rfloor$$

which is given by (4.13). R_i^{up}'s initial value is $c_i + R_i^{up}(c_i)$. $I_{ij}(R_i^{up})$ is calculated by (4.5) with c_j replaced by c_{ji}, and changing (4.4) to

$$I_{ij}(R_i^{up}) = max\begin{cases}\left(\left\lceil\frac{R_i^{up}-\left(c_{ji}+\sum_{s_j^l,\left(\Theta=\Theta_i^{ex}\cap\Theta_j^l\right)\neq\emptyset}len(s_j^l)\right)}{T_j}\right\rceil+1\right)c_{ji}\\ \left\lceil\frac{R_i^{up}-c_j}{T_j}\right\rceil.c_{ji}+c_j-\sum_{s_j^l,\left(\Theta=\Theta_i^{ex}\cap\Theta_j^l\right)\neq\emptyset}len(s_j^l)\end{cases}$$

as given by (4.15). Eq(4.4) is modified to (4.15) because there are two cases for the first job of τ_j (i.e., τ_j^1) contributing to the retry cost of τ_i^x:

Case 1. τ_j^1 (shown in Figure 4.1(b)) contributes by c_{ji}. Thus, other instances of τ_j will begin after this modified WCET, but the sum of the shared objects' atomic section lengths is removed from c_{ji}, causing other instances to start earlier. Thus, the term $\sum_{s_j^l,(\Theta=\Theta_i^{ex}\cap\Theta_j^l)\neq\emptyset} len(s_j^l)$ is added to c_{ji} to obtain the correct start time.

Case 2. τ_j^1 contributes by its c_j, but the sum of the shared atomic section lengths between τ_i and τ_j should be subtracted from the contribution of τ_j^1, as they are already included in the retry cost.

It should be noted that subtraction of $\sum_{s_j^l,(\Theta=\Theta_i^{ex}\cap\Theta_j^l)\neq\emptyset} len(s_j^l)$ is done in the first case to obtain the correct start time of other instances, while in the second case, this is done to get the correct contribution of τ_j^1. The maximum is chosen from the two terms in (4.15), because they differ in the contribution of their τ_j^1s, and the number of instances after that. Claim follows. □

4.2 RCM

As G-RMA is a fixed priority scheduler, a task τ_i will be interfered by those tasks with priorities higher than τ_i (i.e., $p_j > p_i$). Upon a conflict, the RMA CM will commit the transaction that belongs to the higher priority task. Hereafter, we use *RCM* [54] to refer to a multicore system scheduled by G-RMA and resolves STM conflicts by the RMA CM. RCM is shown in Alogrithm 2.

Algorithm 2: RCM

Data: $s_i^k \rightarrow$ interfered atomic section. $s_j^l \rightarrow$ interfering atomic section
Result: which atomic section aborts

```
if T_i < T_j then
    s_j^l aborts;
else
    s_i^k aborts;
end
```

The same illustrative example in Section 4.1.1 is applied for RCM except that tasks' priorities are fixed.

Claim 11. *RCM suffers from transitive retry for multi-object transactions.*

Proof. The proof is the same as proof of Claim 1. □

4.2.1 Maximum Task Interference

Figure 4.3 illustrates the maximum interference caused by a task τ_j to a task τ_i under G-RMA. As τ_j is of higher priority than τ_i, τ_j^k will interfere with τ_i even if it is not totally included in T_i. Unlike the G-EDF case shown in Figure 4.2, where only the δ part of τ_j^1 is considered, in G-RMA, τ_j^k can contribute by the whole c_j, and all atomic sections contained in τ_j^k must be considered. This is because, in G-EDF, the worst-case pattern releases τ_i^a before d_j^1 by δ time units, and τ_i^a cannot be interfered before it is released. But in G-RMA, τ_i^a is already released, and can be interfered by the whole τ_j^k, even if this makes it infeasible.

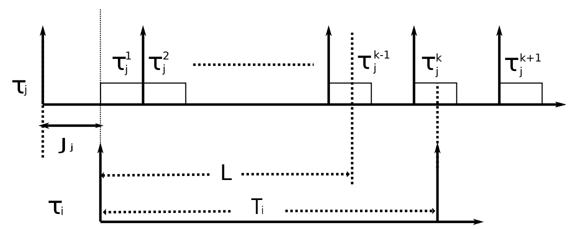


Figure 4.3: Max interference of τ_j to τ_i in G-RMA

Thus, the maximum contribution of τ_j^b to τ_i^a for any duration L is upper bounded by Claim 2, where L can extend to T_i. Note the contrast with ECM, where L cannot be extended directly to T_i, as this will have a different pattern of worst case interference from other tasks.

4.2.2 Retry Cost of Atomic Sections

Claim 12. *Under RCM, total retry cost suffered by all transactions in any job $\tau_i^x \in \tau_i$ during interval $L \leq T_i$ due to direct and indirect conflict with other transactions in jobs with higher priorities than τ_i^x is upper bounded by:*

$$RC_i\left(L\right) \leq \sum_{\tau_j \in \gamma_i^{ex}, p_j > p_i} \left(\sum_{s_j^l, \left(\Theta = \Theta_i^{ex} \cap \Theta_j^l\right) \neq \emptyset} \left(\left(\left\lceil \frac{L}{T_j} \right\rceil + 1 \right) len\left(s_j^l + s_{max}\left(\Theta\right)\right) \right) \right) \tag{4.16}$$

where $s_{max}\left(\Theta\right)$ belongs to a job with lower priority than p_j.

Proof. Under G-RMA, priorities of tasks are fixed. Thus, as $p_j > p_i$, then any job of τ_j will have a higher priority than τ_i^x. So, Claim 2 gives maximum number of jobs of τ_j that interfere with τ_i^x during interval L. By definition of RCM, only transactions with lower priority than p_j can be aborted and retried due to transactions in s_j. Thus, $s_{max}\left(\Theta\right)$ cannot belong to transactions with priorities at least equal to p_j. Following proof of Claim 6, Claim follows. □

Claim 13. *Under RCM, the total retry cost suffered by all transactions in any job $\tau_i^x \in \tau_i$ during an interval $L \leq T_i$ due to release of jobs with higher priority than τ_i^x is upper bounded by*

$$RC_{i_{re}}(L) = \sum_{\forall \tau_j, \, p_j > p_i} \left(\left\lceil \frac{L}{T_j} \right\rceil s_{i_{max}} \right) \tag{4.17}$$

Proof. The proof is the same as that for Claim 8, except that G-RMA uses static priority. Thus, the carried-out jobs will be considered in the interference with τ_i^x. The carried-in jobs are still not considered because they are released before r_i^x. Claim follows. □

Claim 14. *Under RCM, the total retry cost suffered by all transactions in any job $\tau_i^x \in \tau_i$ during an interval $L \leq T_i$ is upper bounded by:*

$$RC_{i_{to}}(L) = RC_i(L) + RC_{i_{re}}(L) \tag{4.18}$$

where $RC_i(L)$ is the maximum retry cost resulting from conflict between transactions in τ_i^x and transactions of other jobs. $RC_i(L)$ is calculated by (4.16). $RC_{i_{re}}(L)$ is the maximum retry cost resulting from the release of higher priority jobs, which preempt transactions in τ_i^x. $RC_{i_{re}}(L)$ is calculated by (4.17).

Proof. Using Claims 12 and 13, and following proof of Claim 9, Claim follows. □

4.2.3 Upper Bound on Response Time

Claim 15. *Under RCM, maximum response time of any job $\tau_i^x \in \tau_i$ is upper bounded by*

$$R_i^{up} = c_i + RC_{i_{to}}(R_i^{up}) + \left\lfloor \frac{1}{m} \sum_{j \neq i, p_j > p_i} I_{ij}(R_i^{up}) \right\rfloor \tag{4.19}$$

where:

- *R_i^{up}'s initial value is $c_i + R_i^{up}(c_i)$.*
- *$RC_{i_{to}}(R_i^{up})$ is calculated by (4.18).*
- *c_j of any job $\tau_j^y \in \tau_j$, where $p_j > p_i$ and $\Theta_j \cap \Theta_i^{ex} \neq \emptyset$, is calculated by (4.14).*
- *$I_{ij}(R_i^{up})$ is calculated by (4.4) with c_j replaced by c_{ji}.*

Proof. Using Theorem 7 in [18], Claim 14 and following proof of Claim 10, Claim follows. □

4.3 Analytical Performance Comparison

Let total utilization required by a synchronization technique A be $U^A = \sum_{\forall\tau_i} \frac{e_i+co_i^A}{T_i}$, where e_i is the worst case execution time for any job of τ_i, and co_i^A is the maximum additional cost added by synchronization technique A to any job in τ_i. All synchronization techniques in this dissertation are used under G-EDF and G-RMA schedulers. Thus, all synchronization techniques have the same schedulability criteria under the same scheduler. So, we compare performance of synchronization techniques using total utilization as done by Claim 3 in [60]. Claim 3 in [60] is extended to compare performance of not only contention managers against lock-free, but also contention managers against each other and contention managers against locking protocols. So, performance of synchronization technique A is equal or better than performance of synchronization technique B if

$$\begin{aligned} U^A &\leq U^B \\ \sum_{\forall\tau_i} \frac{e_i+co_i^A}{T_i} &\leq \sum_{\forall\tau_i} \frac{e_i+co_i^B}{T_i} \end{aligned} \tag{4.20}$$

Eq(4.20) holds if for each τ_i

$$co_i^A \leq co_i^B \tag{4.21}$$

Thus, (4.21) is a sufficient condition for synchronization technique A to have equal or better performance than synchronization technique B.

Retry-loop lock-free [49] and locking protocols (i.e., OMLP [22,29] and RNLP [149]) assume all atomic sections have the same length of the longest atomic section among all tasks. Claim 3 in [60] assumes equal lengths for atomic sections. Performance of synchronization technique A is equal or better than performance of synchronization technique B if (4.21) holds assuming all atomic sections have the same maximum atomic section length and co_i^A is linearly proportional to lengths of atomic sections. Thus, total utilization of any contention manager assumes the maximum transactional length for all transactions when performance of contention managers is compared against lock-free and locking protocols. As each transaction has the same length under different contention managers, then the assumption of equal length for all transactions still holds when performance of different contention managers is compared against each other.

4.4 STM versus Lock-Free

We now would like to understand when STM will be beneficial compared to lock-free synchronization. The retry-loop lock-free approach in [49] is the most relevant to our work. As lock-free instructions access only one object, then Θ_i^k for any s_i^k will be restricted to one object only (i.e., $\Theta_i^k = \theta_i^k$). Thus, transitive retry cannot happen, $\Theta_i^{ex} = \Theta_i$ and $\gamma_i^{ex} = \gamma_i$.

4.4.1 ECM versus Lock-Free

Claim 16. *Following notions in Section 4.3, ECM's total utilization is better or equal to that of [49]'s retry-loop lock-free approach if the size of* s_{max} *does not exceed one half of that of* r_{max}*, where* r_{max} *is the maximum execution cost of a single iteration of any lock-free retry loop of any task. With equal periods for conflicting tasks and high access times to shared objects, the size of* s_{max} *can be much larger than* r_{max}*.*

Proof. Using Claim 3, (4.10) can be upper bounded, during T_i, as:

$$RC_i^{max}(T_i) \leq \sum_{\tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i\right) \neq \emptyset} (2.s_{max}) \right)$$

where s_{max} is the maximum length atomic section among all tasks. Similarly, (4.11) is upper bounded, during T_i, as:

$$RC_{i_{re}}^{max} \leq \sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor s_{max}$$

where $\zeta_i = \{\tau_j : (\tau_j \neq \tau_i) \wedge (D_j < D_i)\}$. Thus, $RC_{i_{to}}$ given by (4.12) can be upper bounded, during T_i, as:

$$RC_{i_{to}}^{max} \leq \left(\sum_{\tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i\right) \neq \emptyset} (2.s_{max}) \right) \right) + \left(\sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor s_{max} \right) \tag{4.22}$$

Retry cost of τ_i during interval T_i due to conflict with other jobs under retry-loop lock-free is given in [49] as:

$$LRC \leq \sum_{\tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) .\beta_{ij}.r_{max} \tag{4.23}$$

where β_{ij} is the number of retry loops of τ_j that access shared objects between τ_i and τ_j. Eq(4.23) needs to be extended to include effect of release of any higher priority job, τ_j^l, preempting τ_i^k when τ_i^k is trying to access an object θ. Release of jobs under ECM and lock-free is independent from accessed objects. Thus, ECM and lock-free have the same pattern of jobs' release. Thus, retry cost of τ_i during T_i due to release of higher priority jobs under retry-loop lock-free is obtained directly from Claim 8 with replacing s_{max} by r_{max}. Thus, total retry cost of any job of τ_i during interval T_i due to conflict of other jobs and release of higher priority jobs is upper bounded by:

$$LRC_{to} \leq \left(\left(\sum_{\tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) .\beta_{ij} \right) + \left(\sum_{\tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor \right) \right) r_{max} \tag{4.24}$$

By substitution of (4.22) and (4.24) into (4.21), then ECM achieves equal or better total utilization than lock-free if

$$\begin{aligned} &\left(\left(\sum_{\forall \tau_j \in \gamma_i}\left(2\left\lceil\frac{T_i}{T_j}\right\rceil \sum_{\forall s_j^l,(\Theta=\Theta_j^l \cap \Theta_i)\neq\emptyset}\right)\right)+\left(\sum_{\forall \tau_j \in \zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor\right)\right) s_{max} \\ \leq \quad & \left(\left(\sum_{\forall \tau_j \in \gamma_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\beta_{ij}\right)+\left(\sum_{\forall \tau_j \in \zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor\right)\right) r_{max} \end{aligned}$$

$$\therefore \frac{s_{max}}{r_{max}} \leq \frac{\left(\sum_{\forall \tau_j \in \gamma_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\beta_{ij}\right)+\left(\sum_{\forall \tau_j \in \zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor\right)}{\left(\sum_{\forall \tau_j \in \gamma_i}\left(2\left\lceil\frac{T_i}{T_j}\right\rceil \sum_{\forall s_j^l,(\Theta=\Theta_j^l \cap \Theta_i)\neq\emptyset}\right)\right)+\left(\sum_{\forall \tau_j \in \zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor\right)} \tag{4.25}$$

Let $\sum_{\forall s_j^l,(\Theta=\Theta_j^l \cap \Theta_i)\neq\emptyset} = \beta_{ij}^*$ and $\sum_{\forall \tau_j \in \zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor = c1$. Then, (4.25) becomes

$$\frac{s_{max}}{r_{max}} \leq \frac{\left(\sum_{\forall \tau_j \in \gamma_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\beta_{ij}\right)+c1}{\left(\sum_{\forall \tau_j \in \gamma_i}\left(2\left\lceil\frac{T_i}{T_j}\right\rceil \beta_{ij}^*\right)\right)+c1} \tag{4.26}$$

We want to get the lower bound over s_{max}/r_{max} that preserves equal or better total utilization for ECM than lock-free:

Each lock-free instruction accesses only one object once. Each transaction accesses only one object to enable comparison with lock-free. An object θ can be accessed multiple times within the same transaction. Thus, $\beta_{ij} \leq \beta_{ij}^*$.

$$\because \frac{\left(\sum_{\forall \tau_j \in \gamma_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil\right)\beta_{ij}^*\right)+c1}{\left(\sum_{\forall \tau_j \in \gamma_i}\left(2\left\lceil\frac{T_i}{T_j}\right\rceil \beta_{ij}^*\right)\right)+2c1} \leq \frac{\left(\sum_{\forall \tau_j \in \gamma_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\beta_{ij}\right)+c1}{\left(\sum_{\forall \tau_j \in \gamma_i}\left(2\left\lceil\frac{T_i}{T_j}\right\rceil \beta_{ij}^*\right)\right)+c1}$$

Thus, (4.26) holds if

$$\frac{s_{max}}{r_{max}} \leq \frac{\left(\sum_{\forall \tau_j \in \gamma_i}\left\lceil\frac{T_i}{T_j}\right\rceil \beta_{ij}^*\right)+c1}{\left(\sum_{\forall \tau_j \in \gamma_i}\left(2\left\lceil\frac{T_i}{T_j}\right\rceil \beta_{ij}^*\right)\right)+2c1} = \frac{1}{2}$$

Thus, the lower bound over s_{max}/r_{max} that preserves equal or better total utilization for ECM than lock-free is 0.5. Now, we want to get the upper bound over s_{max}/r_{max} that preserves equal or better total utilization for ECM than lock-free:

Minimum value for $\left\lceil\frac{T_i}{T_j}\right\rceil$ is 1. So, $2\left\lceil\frac{T_i}{T_j}\right\rceil \geq \left\lceil\frac{T_i}{T_j}\right\rceil + 1, \forall i,j$. Thus, to get upper bound on s_{max}/r_{max}, $\left\lceil\frac{T_i}{T_j}\right\rceil$ assumes its minimum value (i.e., 1). Otherwise, the denominator of (4.26) gets larger than numerator, and s_{max}/r_{max} moves away from its upper bound. $\left\lceil\frac{T_i}{T_j}\right\rceil \rightarrow 1$ for any i, j if all conflicting tasks have equal periods. Thus, by substitution of $\left\lceil \frac{T_i}{T_j} \right\rceil = 1$ into (4.26), we get

$$\frac{s_{max}}{r_{max}} \leq \frac{\left(\sum_{\forall \tau_j \in \gamma_i} 2\beta_{ij}\right) + c1}{\left(\sum_{\forall \tau_j \in \gamma_i} 2\beta^*_{ij}\right) + c1} \tag{4.27}$$

As we are looking for the upper bound over s_{max}/r_{max}, then $\beta_{ij} >> \beta^*_{ij}$. Thus, s_{max} can be much larger than r_{max} while still maintaining equal or better total utilization for ECM than lock-free. From the previous notions, Claim follows. □

4.4.2 RCM versus Lock-Free

Claim 17. *Following notions in Section 4.3, RCM's total utilization is equal or better than that of [49]'s retry-loop lock-free approach if the size of s_{max} does not exceed one half of that of r_{max}, where r_{max} is the maximum execution cost of a single iteration of any lock-free retry loop of any task. With equal periods for conflicting tasks and high access times to shared objects, the size of s_{max} can be much larger than r_{max}.*

Proof. Following the same steps in proof of Claim 16 with the following modifications:

Equation (4.16) is upper bounded by:

$$\sum_{\tau_j \in \gamma_i,\, p_j > p_i} \left(\sum_{s^l_j,\, \left(\Theta = \Theta^l_j \cap \Theta_i\right) \neq \emptyset} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) 2 s_{max} \right) \right) \tag{4.28}$$

Equation (4.17) is upper bounded by:

$$RC_{i_{re}}(T_i) = \sum_{\forall \tau_j,\, p_j > p_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil s_{max} \right) \tag{4.29}$$

Thus,

$$RC^{max}_{i_{to}} \leq \sum_{\tau_j \in \gamma_i,\, p_j > p_i} \left(\sum_{s^l_j,\, \left(\Theta = \Theta^l_j \cap \Theta_i\right) \neq \emptyset} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) 2 s_{max} \right) \right) + \sum_{\forall \tau_j,\, p_j > p_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil s_{max} \right) \tag{4.30}$$

As lock-free is independent from the underlying scheduler, then LRC is still calculated by (4.23). Release of jobs under RCM and lock-free is independent from accessed objects. Thus, RCM and lock-free have the same pattern for object release. Thus, retry cost of transactions in τ_i during T_i due to release of higher priority jobs under retry-loop lock-free is obtained directly from Claim 13 with replacing s_{max} by r_{max}. Thus,

$$LRC_{to} \leq \left(\left(\sum_{\tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right).\beta_{ij}\right) + \left(\sum_{\tau_j,\, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil\right)\right) r_{max} \tag{4.31}$$

Similar to proof of Claim 16, RCM has equal or better total utilization than lock-free if for each τ_i

$$\frac{s_{max}}{r_{max}} \leq \frac{\left(\sum_{\forall \tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right)\beta_{ij}\right) + \left(\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil\right)}{\left(\sum_{\forall \tau_j \in \gamma_i,\, p_j > p_i} \left(2\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right) \sum_{\forall s_j^l, (\Theta = \Theta_j^l \cap \Theta_i) \neq \emptyset}\right)\right) + \left(\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil\right)} \tag{4.32}$$

$$\because \sum_{\forall \tau_j \in \gamma_i,\, p_j > p_i} \left(2\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right) \sum_{\forall s_j^l, (\Theta = \Theta_j^l \cap \Theta_i) \neq \emptyset}\right) \leq \sum_{\forall \tau_j \in \gamma_i} \left(2\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right) \sum_{\forall s_j^l, (\Theta = \Theta_j^l \cap \Theta_i) \neq \emptyset}\right)$$

$\therefore$ Eq(4.32) holds if

$$\frac{s_{max}}{r_{max}} \leq \frac{\left(\sum_{\forall \tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right)\beta_{ij}\right) + \left(\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil\right)}{\left(\sum_{\forall \tau_j \in \gamma_i} \left(2\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right) \sum_{\forall s_j^l, (\Theta = \Theta_j^l \cap \Theta_i) \neq \emptyset}\right)\right) + \left(\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil\right)} \tag{4.33}$$

Let $\sum_{\forall s_j^l, (\Theta = \Theta_j^l \cap \Theta_i) \neq \emptyset} = \beta_{ij}^*$ and $\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil = c1$. Then (4.32) becomes

$$\frac{s_{max}}{r_{max}} \leq \frac{\left(\sum_{\forall \tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right)\beta_{ij}\right) + c1}{\left(\sum_{\forall \tau_j \in \gamma_i} \left(2\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right)\beta_{ij}^*\right) + c1\right)} \tag{4.34}$$

We want to get lower bound over s_{max}/r_{max} that preserves equal or better total utilization for RCM than lock-free:

Similar to proof of Claim 16, β_{ij} assumes its minimum value β_{ij}^*.

$$\because \frac{\left(\sum_{\forall \tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right)\beta_{ij}\right) + c1}{\left(\sum_{\forall \tau_j \in \gamma_i} \left(2\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right)\beta_{ij}^*\right) + 2c1\right)} \leq \frac{\left(\sum_{\forall \tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right)\beta_{ij}\right) + c1}{\left(\sum_{\forall \tau_j \in \gamma_i} \left(2\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right)\beta_{ij}^*\right) + c1\right)} \tag{4.35}$$

Then (4.34) holds if

$$\frac{s_{max}}{r_{max}} \leq \frac{\left(\sum_{\forall \tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right)\beta_{ij}\right) + c1}{\left(\sum_{\forall \tau_j \in \gamma_i} \left(2\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right)\beta_{ij}^*\right) + 2c1\right)} = \frac{1}{2} \tag{4.36}$$

We want to get upper bound over s_{max}/r_{max} that preserves equal or better total utilization for RCM than lock-free:

Similar to proof of Claim 16, $\left\lceil \frac{T_i}{T_j} \right\rceil$ assumes its minimum value (i.e., 1), $\beta_{ij} >> \beta^*_{ij}$. Thus, s_{max} can be much larger than r_{max}. From the previous notions, Claim follows. □

4.5 STM versus Locking protocols

Performance of different CMs is compared against real-time locking protocols (i.e., OMLP [22, 29] and RNLP [149]) using total utilization under G-EDF and G-RMA. In [22, 29, 148, 149], priority inversion bound (*pi-blocking*) is considered part of each task's execution time. Thus, each task's WCET is inflated by *pi-blocking* bounds. Similarly, under different CMs, each task's WCET is inflated by its total retry cost (i.e., retry cost due to direct and indirect conflict with other tasks. Besides retry cost due to release of higher priority jobs). Following notions in Section 4.3, total utilization of a specific STM CM algorithm A is equal or better than total utilization of a real-time locking protocol B if

$$\forall \tau_i,\ RC_A(T_i) \leq PI_B(T_i) \tag{4.37}$$

If τ_i has no critical sections, then $RC_A(T_i) = PI_B(T_i) = 0$. Thus, independent tasks will not be considered in (4.37).

4.5.1 Priority Inversion under OMLP

Under OMLP [22, 29], $PI_{OMLP}(T_i)$ for any job τ_i^x is upper bounded by

$$PI_{OMLP}(T_i) \leq \sum_{k=1}^{n_r} N_{i,k}(2m-1)L_{max} \tag{4.38}$$

Where n_r is total number of resources. $N_{i,k}$ is maximum number of times resource k is accessed by τ_i. L_{max} is the maximum length critical section in all tasks. Let $N_i = \sum_{k=1}^{n_r} N_{i,k}$, which is the total number of critical sections in any job τ_i^x. Thus, (4.38) becomes

$$PI_{OMLP}(T_i) \leq N_i(2m-1)L_{max} \tag{4.39}$$

Let $N_{max} = max\left\{N_i\right\}_{\forall i}$, $N_{min} = min\left\{N_i\right\}_{\forall i}$, $\Phi_{max} = max\left\lceil \frac{T_i}{T_j} \right\rceil_{\forall i,j}$. As independent tasks are not considered in (4.37), $\therefore\ N_{max}, N_{min} \geq 1$.

OMLP uses group locking to access multiple (i.e., nested) objects in a critical section. Thus, all objects within the same atomic section are protected by the same lock (i.e., resource). Sections 4.5.5 and 4.5.6 investigates comparison between different CMs and fine-grained locking protocols (i.e., RNLP) to access multiple objects within a critical section without group locking.

4.5.2 ECM versus OMLP

Claim 18. *Following notions in Section 4.3, total utilization of ECM is equal or better than total utilization of OMLP if*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_{min}\,(2m-1)}{(2N_{max}+1)\,(n-1)\Phi_{max}} \tag{4.40}$$

As number of atomic sections in each task increases, all tasks have equal periods and equal number of atomic sections, and number of processors is at least equal to number of tasks, then s_{max} can be at least equal to L_{max} with total utilization of ECM equal or better than total utilization of OMLP.

Proof. Substitute $RC_A(T_i)$ in (4.37) by (4.22) with γ_i replaced with γ_i^{ex} and Θ_i replaced with Θ_i^{ex}. Substitute $PI_B(T_i)$ in (4.37) by (4.39). $\therefore$ (4.37) holds if $\forall \tau_i$

$$\left(\sum_{\tau_j\in\gamma_i^{ex}}\left(\left\lceil\frac{T_i}{T_j}\right\rceil\sum_{\forall s_j^l,\left(\Theta=\Theta_j^l\cap\Theta_i^{ex}\right)\neq\emptyset}(2.s_{max})\right)\right)+\left(\sum_{\forall\tau_j\in\zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor s_{max}\right) \leq N_i\,(2m-1)\,L_{max} \tag{4.41}$$

Let $N_{i,j}=\sum_{\forall s_j^l,\left(\Theta=\Theta_j^l\cap\Theta_i\right)\neq\emptyset}$. So, $N_{i,j}$ is number of transactions in any job of τ_j conflicting with any transaction in any job of τ_i. Thus, (4.41) becomes

$$\left(2\left(\sum_{\forall\tau_j\in\gamma_i^{ex}}\left(\left\lceil\frac{T_i}{T_j}\right\rceil N_{i,j}\right)\right)+\left(\sum_{\forall\tau_j\in\zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor\right)\right)s_{max} \leq N_i\,(2m-1)\,L_{max} \tag{4.42}$$

$$\therefore \frac{s_{max}}{L_{max}} \leq \frac{N_i\,(2m-1)}{2\left(\sum_{\forall\tau_j\in\gamma_i^{ex}}\left(\left\lceil\frac{T_i}{T_j}\right\rceil N_{i,j}\right)\right)+\left(\sum_{\forall\tau_j\in\zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor\right)} \tag{4.43}$$

Let $N_{max}=max\,\{N_i\}_{\forall i}$, $N_{min}=min\,\{N_i\}_{\forall i}$, $\Phi_{max}=max\left\lceil\frac{T_i}{T_j}\right\rceil_{\forall i,j}$. By definition of γ_i^{ex} and ζ_i, $n-1\geq|\zeta_i|,\,|\gamma_i^{ex}|$. $\because$ $N_{max}\geq N_{i,j}$, $N_{min}\leq N_i$ and $\Phi_{max}\geq\left\lceil\frac{T_i}{T_j}\right\rceil\geq\left\lfloor\frac{T_i}{T_j}\right\rfloor$. $\therefore$ Eq(4.43) holds if

$$\begin{aligned}\frac{s_{max}}{L_{max}} &\leq \frac{N_{min}\,(2m-1)}{2\left(\sum_{\forall\tau_j\in\gamma_i^{ex}}(\Phi_{max}N_{max})\right)+\left(\sum_{\forall\tau_j\in\zeta_i}\Phi_{max}\right)} \\ &\leq \frac{N_{min}\,(2m-1)}{(2N_{max}+1)\,(n-1)\Phi_{max}}\end{aligned} \tag{4.44}$$

To get the maximum upper bound over s_{max}/L_{max}, let N_{min} reaches its maximum value and N_{max} reaches its minimum value (i.e., $N_{min}=N_{max}$ by definition of N_{min} and N_{max}). Thus, all tasks have the same number of atomic sections. Let Φ_{max} reaches its minimum value (i.e.,

$\Phi_{max} = 1$ by definition of Φ_{max}). Thus, all tasks have the same periods. By substitution of $\Phi_{max} = 1$ and $N_{min} = N_{max} = N$, where N is constant, in (4.44), then

$$\frac{s_{max}}{L_{max}} \leq \frac{N(2m-1)}{(2N+1)(n-1)} \leq \frac{2m-1}{\left(2+\frac{1}{N}\right)(n-1)} \tag{4.45}$$

As we are looking for maximum upper bound over s_{max}/L_{max}, N assumes its maximum value (i.e., $N \to \infty$) in (4.45). Thus,

$$\frac{s_{max}}{L_{max}} \leq \frac{2m-1}{2(n-1)} \tag{4.46}$$

$\therefore s_{max} \geq L_{max}$ if $2m-1 \geq 2(n-1)$. As m and n are integers, $\therefore s_{max} \geq L_{max}$ if $m \geq n$. Claim follows. □

4.5.3 RCM versus OMLP

Claim 19. *Following notions in Section 4.3, total utilization of RCM is equal or better than total utilization of OMLP if*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_{min}(2m-1)}{(2(\Phi_{max}+1)N_{max}+\Phi_{max})(n-1)} \tag{4.47}$$

As number of atomic sections in each task increases, all tasks have equal periods and equal number of atomic sections, and number of processors is at least double the number of tasks, then s_{max} can be at least equal to L_{max} with total utilization of RCM equal or better than total utilization of OMLP.

Proof. Substitute $RC_A(T_i)$ in (4.37) by (4.30) with γ_i replaced with γ_i^{ex} and Θ_i replaced with Θ_i^{ex}. Following the same steps in proof of Claim 18, Claim follows. □

4.5.4 Priority Inversion under RNLP

Under RNLP [149] for global scheduling and *I-KGLP* token lock (introduced as R^2DGLP in [150]), $PI_{RNLP}(T_i)$ for any job τ_i^x is upper bounded by $(2m-1)L_{max}$ for each outermost request, where L_{max} is the maximum length of any outermost request. Thus, if N_i is total number of outermost critical sections in any job of τ_i, then

$$PI_{RNLP}(T_i) = N_i(2m-1)L_{max} \tag{4.48}$$

Let $N_{max} = max\{N_i\}_{\forall i}$, $N_{min} = min\{N_i\}_{\forall i}$, $\Phi_{max} = max\left\lceil \frac{T_i}{T_j} \right\rceil$. As independent tasks are not considered in (4.37), $\therefore N_{max}, N_{min} \geq 1$.

In contrast to OMLP, RNLP supports nesting of objects. Thus, each object can be accessed individually without being grouped with other objects in the same critical section.

4.5.5 ECM versus RNLP

Claim 20. *Following notions in Section 4.3, total utilization of ECM is equal or better than total utilization of RNLP if*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_{min}\,(2m-1)}{(2N_{max}+1)\,(n-1)\Phi_{max}} \tag{4.49}$$

As number of atomic sections in each task increases, all tasks have equal periods and equal number of atomic sections, and number of processors is at least equal to number of tasks, then s_{max} can be at least equal to L_{max} with total utilization of ECM equal or better than total utilization of RNLP.

Proof. Substitute $RC_A(T_i)$ in (4.37) by (4.22) with γ_i replaced with γ_i^{ex} and Θ_i replaced with Θ_i^{ex}. Substitute $PI_B(T_i)$ in (4.37) by (4.48). ∴ (4.37) becomes

$$\begin{aligned}&\left(\sum_{\tau_j\in\gamma_i^{ex}}\left(\left\lceil\frac{T_i}{T_j}\right\rceil\sum_{\forall s_j^l,\left(\Theta=\Theta_j^l\cap\Theta_i^{ex}\right)\neq\emptyset}(2s_{max})\right)\right)+\left(\sum_{\forall\tau_j\in\zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor s_{max}\right)\\ \leq\;& N_i\,(2m-1)\,L_{max}\end{aligned} \tag{4.50}$$

Following the same steps of proof of Claim 18, Claim follows. □

4.5.6 RCM versus RNLP

Claim 21. *Following notions in Section 4.3, total utilization of RCM is equal or better than total utilization of RNLP if*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_{min}\,(2m-1)}{(2\,(\Phi_{max}+1)\,N_{max}+\Phi_{max})\,(n-1)} \tag{4.51}$$

As number of atomic sections in each task increases, all tasks have equal periods and equal number of atomic sections, and number of processors is at least double the number of tasks, then s_{max} *can be at least equal to* L_{max} *with total utilization of RCM equal or better than total utilization of RNLP.*

Proof. Substitute $RC_A(T_i)$ in (4.37) by (4.30). γ_i is replaced with γ_i^{ex} and Θ_i is replaced with Θ_i^{ex}. Substitute $PI_B(T_i)$ in (4.37) by (4.48). Following the same steps of proof of Claim 18, Claim follows. □

4.6 Conclusions

ECM and RCM use jobs' priorities to resolve conflicts between transactions. The transaction with lower priority aborts and retries due to the transaction with higher priority. As each transaction can access multiple objects, a transaction may abort indirectly due to another transaction with no shared objects between them. The indirect retrial is denoted as transitive retry. Under both ECM and RCM, a task incurs at most $2s_{max}$ retry cost for each of its atomic sections due to a conflict with another task's atomic section. Transactions can also retry due to release of higher priority jobs that preempt a transaction in a lower priority job.

The s_{max}/r_{max} ratio is a sufficient condition to determine whether STM is better or as good as lock-free based on total utilization. ECM and RCM have equal or better total utilization than retry-loop lock-free if s_{max} does not exceed one half of r_{max}. s_{max} can exceed r_{max} with equal periods between conflicting tasks, and large access times to the same object within the same transaction.

Performance of ECM and RCM was compared against real-time locking protocols (i.e., OMLP and RNLP) in terms of total utilization. As number of atomic sections in each task increases, all tasks have equal periods and equal number of atomic sections, and number of processors is at least equal to number of tasks, then s_{max} can be at least equal to L_{max} with total utilization of ECM equal or better than total utilization of OMLP and RNLP. The same results apply to total utilization comparison between RCM and locking protocols except that number of processors should be at least double number of tasks.

Chapter 5

The LCM Contention Manager

Under ECM and RCM (Chapter 4), each atomic section can be aborted for at most $2s_{max}$ by a single interfering atomic section. We present a novel contention manager (CM) for resolving transactional conflicts, called length-based CM (or LCM) [53]. LCM can reduce the abortion time of a single atomic section due to an interfering atomic section below $2s_{max}$. We upper bound transactional retries and response times under LCM, when used with G-EDF and G-RMA schedulers. We identify the conditions under which LCM outperforms ECM, RCM, retry-loop lock-free [49] and real-time locking protocols (i.e., OMLP [22, 29] and RNLP [149]).

The rest of this Chapter is organized as follows: Section 5.1 presents Length-based Contention Manager (LCM) and illustrates its behaviour. Section 5.2 derives LCM properties. Retry cost and response time analysis of tasks under LCM/G-EDF is given in Section 5.3. Performance of LCM/G-EDF is compared to performance of ECM, lock-free and locking protocols based on total utilization in Section 5.4. Section 5.5 gives retry cost and response time analysis for LCM/G-RMA. Performance of LCM/G-RMA is compared against RCM, lock-free and locking protocols based on total utilization in Section 5.6. We conclude Chapter in Section 5.7.

5.1 Length-based CM

LCM resolves conflicts based on the priority of conflicting transactions, besides the length of the interfering atomic section, and the length of the interfered atomic section. Priority of each transaction equals priority of its containing job (i.e., $p\left(s_i^k\right) = p_i^x$ where $s_i^k \in \tau_i^x$). ECM and RCM (Chapter 4) use only priorities to resolve conflicts. LCM allows lower priority jobs to retry for lesser time than that under ECM and RCM, but higher priority jobs, sometimes, wait for lower priority ones with bounded priority-inversion.

5.1.1 Design and Rationale

Algorithm 3: LCM

Data: s_i^k and s_j^l are two conflicting atomic sections.
$\psi \rightarrow$ predefined threshold $\in [0, 1]$.
$\delta_i^k \rightarrow$ remaining execution length of s_i^k.
$s\left(s_i^k\right) \rightarrow$ start time of s_i^k. $s\left(s_i^k\right)$ is updated each time s_i^k aborts and retries to the start time of the new retry.
$s\left(s_j^l\right) \rightarrow$ the same as $s\left(s_i^k\right)$ but for s_j^l.
Result: which atomic section of s_i^k or s_j^l aborts

1 **if** $s\left(s_i^k\right) < s\left(s_j^l\right)$ **then**
2   **if** $p\left(s_i^k\right) > p\left(s_j^l\right)$ **then**
3   s_j^l aborts;
4   **else**
5         $c_{ij}^{kl} = len(s_j^l)/len(s_i^k)$;
6         $\alpha_{ij}^{kl} = ln(\psi)/(ln(\psi) - c_{ij}^{kl})$;
7         $\alpha = \left(len(s_i^k) - \delta_i^k\right)/len(s_i^k)$;
8   **if** $\alpha \leq \alpha_{ij}^{kl}$ **then**
9   s_i^k aborts;
10   **else**
11   s_j^l aborts;
12   **end**
13   **end**
14 **else**
15   Swap s_i^k and s_j^l;
16 **end**

For both ECM and RCM, s_i^k can be totally repeated if s_j^l — which belongs to a higher priority job τ_j^b than τ_i^a — conflicts with s_i^k at the end of its execution, while s_i^k is just about to commit. Thus, LCM, shown in Algorithm 3, uses the remaining length of s_i^k when it is interfered, as well as $len(s_j^l)$, to decide which transaction must be aborted. If s_i^k starts before s_j^l, then s_i^k is the interfered atomic section and s_j^l is the interfering atomic section (step 1). Otherwise, s_i^k and s_j^l are swapped (step 15). If $p\left(s_i^k\right)$ was greater than $p\left(s_j^l\right)$, then s_i^k would be the one that commits, because it belongs to a higher priority job, and it started before s_j^l (step 3). Otherwise, c_{ij}^{kl} is calculated (step 5) to determine whether it is worth aborting s_i^k in favour of s_j^l, because $len(s_j^l)$ is relatively small compared to the remaining execution length of s_i^k (explained further).

We assume that:

$$c_{ij}^{kl} = len(s_j^l)/len(s_i^k) \tag{5.1}$$

where $c_{ij}^{kl} \in]0, \infty[$, to cover all possible lengths of s_j^l. Our idea is to reduce the opportunity for the abort of s_i^k if it is close to committing when interfered and $len(s_j^l)$ is large. This abort opportunity is increasingly reduced as s_i^k gets closer to the end of its execution, or $len(s_j^l)$ gets larger.

On the other hand, as s_i^k is interfered early, or $len(s_j^l)$ is small compared to s_i^k's remaining length, the abort opportunity is increased even if s_i^k is close to the end of its execution. To decide whether s_i^k must be aborted or not, we use a threshold value $\psi \in [0,1]$ that determines α_{ij}^{kl} (step 6), where α_{ij}^{kl} is the maximum percentage of $len(s_i^k)$ below which s_j^l is allowed to abort s_i^k and is calculated as

$$\alpha_{ij}^{kl} = \frac{ln\,(\Psi)}{ln\,(\Psi) - c_{ij}^{kl}} \tag{5.2}$$

Thus, if the already executed part of s_i^k — when s_j^l interferes with s_i^k — does not exceed $\alpha_{ij}^{kl} len(s_i^k)$, then s_i^k is aborted (step 9). Otherwise, s_j^l is aborted (step 11).

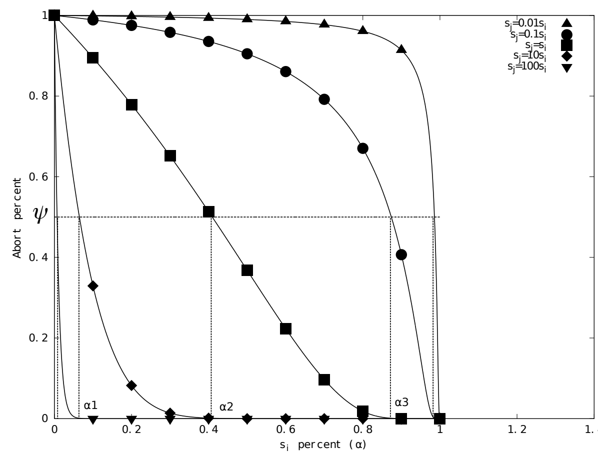


Figure 5.1: Interference of s_i^k by various lengths of s_j^l

The behaviour of LCM is illustrated in Figure 5.1. In this figure, the horizontal axis corresponds to different values of α ranging from 0 to 1, and the vertical axis corresponds to different values of abort opportunities, $f(c_{ij}^{kl}, \alpha)$, ranging from 0 to 1 and calculated by (5.3):

$$f(c_{ij}^{kl}, \alpha) = e^{\frac{-c_{ij}^{kl}\alpha}{1-\alpha}} \tag{5.3}$$

where c_{ij}^{kl} is calculated by (5.1).

Figure 5.1 shows one atomic section s_i^k (whose α changes along the horizontal axis) interfered by five different lengths of s_j^l. For a predefined value of $f(c_{ij}^{kl}, \alpha)$ (denoted as ψ in Algorithm 3), there corresponds a specific value of α (which is α_{ij}^{kl} in Algorithm 3) for each curve. For example, when $len(s_j^l) = 0.1 \times len(s_i^k)$, s_j^l aborts s_i^k if the latter has not executed more than $\alpha 3$ percentage (shown in Figure 5.1) of its execution length. As $len(s_j^l)$ decreases, the corresponding α_{ij}^{kl} increases (as shown in Figure 5.1, $\alpha 3 > \alpha 2 > \alpha 1$).

Equation (5.3) achieves the desired requirement that the abort opportunity is reduced as s_i^k gets closer to the end of its execution (as $\alpha \to 1$, $f(c_{ij}^{kl}, 1) \to 0$), or as the length of the conflicting transaction increases (as $c_{ij}^{kl} \to \infty$, $f(\infty, \alpha) \to 0$). Meanwhile, this abort opportunity is increased as s_i^k is interfered closer to its release (as $\alpha \to 0$, $f(c_{ij}^{kl}, 0) \to 1$), or as the length of the conflicting transaction decreases (as $c_{ij}^{kl} \to 0$, $f(0, \alpha) \to 1$).

LCM is not a centralized CM, which means that, upon a conflict, each transactions has to decide whether it must commit or abort. LCM suffers from transitive retry (Section 4.1.2).

Claim 22. *LCM suffers from transitive retry for multi-object transactions.*

Proof. Following the proof of Claim 1, Claim follows. □

5.1.2 LCM Illustrative Example

Behaviour of LCM can be illustrated by the following example:

- Transaction $s_i^k \in \tau_i^x$ begins execution. Currently, s_i^k does not conflict with any other transaction.
- Transaction $s_j^l \in \tau_j^y$ is released while s_i^k is still running. $\Theta_i^{k^{ex}} \cap \Theta_j^l \neq \emptyset$ and $p_j^y > p_i^x$ (where priority is dynamic in G-EDF, and fixed in G-RMA). c_{ij}^{kl}, α_{ij}^{kl} and α are calculated by steps 5 to 7 in Algorithm 3. s_i^k has not reached α percentage of its execution length yet.
- $\alpha < \alpha_{ij}^{kl}$. Then, s_j^l is allowed to abort and restart s_i^k.
- s_j^l commits. s_i^k executes again.
- Transaction $s_h^v \in \tau_h^u$ is released while s_i^k is running. $\Theta_i^{k^{ex}} \cap \Theta_h^v \neq \emptyset$ and $p_h^u > p_i^x$. c_{ih}^{kv}, α_{ih}^{kv} and α are calculated by steps 5 to 7 in Algorithm 3. s_i^k has already passed α percentage of its execution length. So, s_h^v aborts and restarts in favour of s_i^k.
- Transaction $s_a^b \in \tau_a^f$ is released. $\Theta_i^{k^{ex}} \cap \Theta_a^b \neq \emptyset$ and $p_a^f > p_i^x$ but $p_a^f < p_h^u$. c_{ia}^{kb}, α_{ia}^{kb} and α are calculated by steps 5 to 7 in Algorithm 3. s_i^k has not reached α percentage of its execution length yet. So, s_a^b is allowed to abort s_i^k. Because s_a^b is just starting, LCM allows s_h^v to abort s_a^b. So, the highest priority transaction is not blocked by an intermediate priority transaction s_a^b.
- When s_h^v commits. s_a^b is allowed to execute while s_i^k is retrying.
- When s_a^b commits, s_i^k executes.
- Transaction $s_c^n \in \tau_o^z$ is released while s_i^k is running. $\Theta_i^{k^{ex}} \cap \Theta_c^n \neq \emptyset$ and $p_o^z < p_i^x$. So, s_i^k commits first, then s_c^n is allowed to proceed.

5.2 Properties

LCM properties are given by the following Lemmas. These properties are used to derive retry cost and response time of transactions and tasks under LCM.

Claim 23. *$r\left(s_i^k\right)$ is updated each time s_i^k aborts and retries to the new start time of the new retry to avoid deadlock that can result from conflicting transactions aborting each other.*

Proof. Assume a set of transactions S that are conflicting together. Each transaction aborts and retries due to the others. So, a higher priority transaction s_j^l aborts and retries due to a lower priority transaction s_i^k. s_i^k itself aborts and retries due to another transaction. Thus, the new $r\left(s_i^k\right)$ will be at least equal to the new $r\left(s_j^l\right)$. By definition of LCM, s_j^l will be chosen to commit first because of its higher priority. By extending this result to all transactions in S, the highest priority transaction will commit. Thus, deadlock is avoided. Claim follows. □

Claim 24. *Let s_j^l interferes once with s_i^k at most at α_{ij}^{kl}. $p\left(s_j^l\right) > p\left(s_i^k\right)$. Then, the maximum contribution of s_j^l to s_i^k's retry cost is:*

$$W_i^k(s_j^l) \leq \alpha_{ij}^{kl} len\Big(s_i^k\Big) + len\Big(s_j^l\Big) \tag{5.4}$$

Proof. If s_j^l interferes with s_i^k at a Υ percentage, where $\Upsilon < \alpha_{ij}^{kl}$, then the retry cost of s_i^k is $\Upsilon len(s_i^k) + len(s_j^l)$, which is lower than that calculated in (5.4). Besides, if s_j^l interferes with s_i^k after α_{ij}^{kl} percentage, then s_i^k will not abort. □

Claim 25. *A higher priority transaction, s_j^l, aborts and retries due to a lower priority transaction, s_i^k, if s_j^l interferes with s_i^k after the α_{ij}^{kl} percentage. s_j^l's retry cost, due to s_i^k is upper bounded by:*

$$W_j^l(s_i^k) \leq \Big(1 - \alpha_{ij}^{kl}\Big) len\Big(s_i^k\Big) \tag{5.5}$$

Proof. It is derived directly from Claim 24, as s_j^l will have to retry for the remaining length of s_i^k. □

Claim 26. *As length of s_i^k- interfered by a higher priority transaction s_j^l - increases, then α_{ij}^{kl} also increases.*

Proof. As $len\left(s_i^k\right)$ increases, then c_{ij}^{kl} decreases by definition of (5.1). Noting that $ln(\Psi) \leq 0$ because $\Psi \in [0, 1]$. Thus, α_{ij}^{kl} increases as c_{ij}^{kl} decreases by definition of (5.2). Claim follows. □

Claim 27. *Let $conf\left\{s_i^k\right\}$ be the set of all transactions that do not belong to any job of τ_i and are conflicting, directly or indirectly(transitively), with s_i^k. Each transaction $s_j^l \in conf\left\{s_i^k\right\}, p\left(s_j^l\right) > p\left(s_i^k\right)$ contributes to the retry cost of s_i^k by at most*

$$len\left(s_j^l + \alpha_{max}^{jl} s_{max}(\Theta)\right) \tag{5.6}$$

where $s_{max}(\Theta)$ is the maximum length atomic section (transaction) in $conf\{s_i^k\}$ that accesses at least one object in Θ and its priority is lower than $p(s_j^l)$. $s_{max}(\Theta) \notin s_j$ and $\Theta \subseteq \Theta_i^{k^{ex}} \cap \Theta_j^l$. α_{max}^{jl} is calculated by (5.2) due to interference of $s_{max}(\Theta)$ by s_j^l.

Proof. Under ECM and RCM (Chapter 4), lower priority transactions abort and retry only due to higher priority transactions. Whereas, under LCM, a transaction s_i^k can be aborted due to higher priority transactions. s_i^k can also be delayed by lower priority transactions. Thus, proof follows proof of Claim 5 with the following modifications:

- According to Claims 24 and 25, s_j^l can cause lower priority transactions to retry and higher priority transactions to be delayed. From Claims 24 and 25, it appears that contribution of s_j^l to the retry cost of lower priority transactions is greater than delay caused by s_j^l to higher priority transactions. Thus, retry cost caused by s_j^l to lower priority transactions is taken as the contribution of s_j^l to the retry cost of s_i^k.
- By Claim 26 and definition of $s_{max}(\Theta)$, α_{max}^{jl} is the maximum α that results from interference of a lower priority transaction- accessing any object $\theta \in \Theta$ - by s_j^l.
- s_i^k can abort and retry due to higher priority transactions. Also, s_i^k can be delayed due to lower priority transactions. Thus, $p(s_{max}) < p\left(s_j^l\right)$, but $p\left(s_{max}^{jl}\right)$ does not have to be greater than $p\left(s_i^k\right)$.

Claim follows. □

Claim 28. *Let $conf\left\{s_i^k\right\}$ be the set of all transactions that do not belong to any job of τ_i and are conflicting, directly or indirectly(transitively), with s_i^k. Each transaction $s_j^l \in conf\left\{s_i^k\right\}, p\left(s_j^l\right) < p\left(s_i^k\right)$ contributes to the delay of s_i^k by at most*

$$\left(1 - \alpha_{min}^{jl}\right) len\left(s_j^l\right) \tag{5.7}$$

where α_{min}^{jl} is the minimum α_{jx}^{ly}- calculated by (5.2)- that results from delay of any higher priority transaction s_x^y by the lower priority s_j^l.

Proof. If s_j^l is to abort and retry, then the delay to s_i^k that results from each retry of s_j^l is covered by Claim 27. Thus, the delay that results from s_j^l when it does not retry is given by Claim 25 by minimizing α_{ij}^{kl} in (5.5) to its minimum value (i.e., α_{min}^{jl}). Claim follows. □

Claim 29. *Under LCM with G-EDF and G-RMA, priority inversion time for any job τ_i^x during T_i is bounded.*

Proof. Under LCM, priority of each transaction s_i^k equals priority of its containing job τ_i^x. Under G-EDF, number of lower priority jobs of τ_j that are released during T_i is upper bounded by 1. Under G-RMA, number of lower priority jobs of τ_j that are released during T_i is upper bounded by $\left\lceil \frac{T_i}{T_j} \right\rceil + 1$. Number of transactions is fixed for each job. So, by Claim 28, Claim follows. □

5.3 Retry Cost and Response Time of LCM/G-EDF

Claim 30. *Under LCM/G-EDF, the total retry cost suffered by all transactions in any job $\tau_i^x \in \tau_i$ during interval $L \leq T_i$ due to direct and indirect conflict with other transactions is upper bounded by:*

$$RC_i\left(L\right) \leq \sum_{\tau_j \in \gamma_i^{ex}} \left(g_{ij}^{gedf} \sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i^{ex}\right) \neq \emptyset} len\left(s_j^l + \alpha_{max}^{jl} s_{max}(\Theta)\right) \right) \tag{5.8}$$

where $s_{max}\left(\Theta\right) \notin s_j$ and α_{max}^{jl} is given by (5.2) due to interference of the lower priority $s_{max}\left(\Theta\right)$ by the higher priority s_j^l. g_{ij}^{gedf} is calculated by (4.2).

Proof. From Claims 24 and 25, it appears that contribution of s_j^l to the retry cost of lower priority transactions is greater than delay caused by s_j^l to higher priority transactions. Thus, retry cost caused by s_j^l to lower priority transactions is taken as the contribution of s_j^l to the retry cost of s_i^k. Under G-EDF, priorities are determined by the absolute deadline of the job. Thus, the same transaction s_j^l can be of higher or lower priority than $p\left(s_i^k\right)$ according to the absolute deadline of containing job of s_j^l. So, only jobs of $\tau_j \in \gamma_i$ that have an absolute deadline that at most coincides with d_i^x are considered. Thus, delay of lower priority transactions is ignored. Following Claim 27 and Claim 6, Claim follows □

Claim 31. *Under LCM/G-EDF, the total retry cost suffered by all transactions in any job $\tau_i^x \in \tau_i$ during an interval $L \leq T_i$ due to release of jobs with higher priority than τ_i^x is upper bounded by*

$$RC_{i_{re}}(L) \leq \sum_{\forall \tau_j \in \zeta_i} \begin{cases} \left\lceil \frac{L}{T_j} \right\rceil s_{i_{max}} & , L \leq T_i - T_j \\ \\ \left\lfloor \frac{T_i}{T_j} \right\rfloor s_{i_{max}} & , L > T_i - T_j \end{cases} \tag{5.9}$$

where $\zeta_i = \{\tau_j : (\tau_j \neq \tau_i) \wedge (D_j < D_i)\}$.

Proof. LCM/G-EDF and ECM has the same pattern for release of jobs. Thus, proof is the same as proof of Claim 8. Claim follows. □

Claim 32. *Under LCM/G-EDF, the total retry cost suffered by all transactions in any job $\tau_i^x \in \tau_i$ during an interval $L \leq T_i$ is upper bounded by:*

$$RC_{i_{to}}(L) = RC_i(L) + RC_{i_{re}}(L) \tag{5.10}$$

where $RC_i(L)$ is the maximum retry cost resulting from conflict between transactions in τ_i^x and transactions of other jobs. $RC_i(L)$ is calculated by (5.8). $RC_{i_{re}}(L)$ is the maximum retry cost resulting from the release of higher priority jobs, which preempt transactions in τ_i^x. $RC_{i_{re}}(L)$ is calculated by (5.9).

Proof. Proof follows directly from Claims 30, 31 and proof of Claim 9. □

Claim 33. *Under LCM/G-EDF, maximum response time of any job $\tau_i^x \in \tau_i$ is upper bounded by Claim 10 where $RC_{i_{to}}(R_i^{up})$ is upper bounded by (5.10).*

Proof. Proof follows directly from Claim 32 and proof of Claim 10. □

5.4 Total utilization of LCM/G-EDF

Following notions in Section 4.3, we compare performance of LCM/G-EDF against ECM (Chapter 4), lock-free [49] and locking protocols (i.e., OMLP [22, 29] and RNLP [149]) in terms of total utilization to understand when LCM/G-EDF will perform better.

5.4.1 LCM/G-EDF versus ECM

Claim 34. *Following notions in Section 4.3, total utilization of LCM/G-EDF is always equal or better than ECM.*

Proof. Under ECM, $RC_{ECM}^{to}(T_i)$ is upper bounded by (4.22) with replacing γ_i by γ_i^{ex}.

$RC^{to}_{LCM/G-EDF}(T_i)$ is given by (5.10) and upper bounded by

$$\begin{aligned} RC^{to}_{LCM/G-EDF}\left(T_i\right) &\leq \left(\sum_{\tau_j \in \gamma_i^{ex}} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i^{ex}\right) \neq \emptyset} len\left(s_j^l + \alpha_{max}^{jl} s_{max}(\Theta)\right)\right)\right) \\ &+ \left(\sum_{\tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor s_{i_{max}}\right) \\ &\leq \left((1+\alpha_{max}) \sum_{\tau_j \in \gamma_i^{ex}} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i^{ex}\right) \neq \emptyset} \right) s_{max}\right) \\ &+ \left(\sum_{\tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor s_{max}\right) \end{aligned} \tag{5.11}$$

where s_{max} is the length of the longest transaction among all tasks. α_{max} is the maximum value of α_{xy}^{kl} for any two transactions s_x^k and s_y^l. By substitution of $RC^{to}_{LCM/G-EDF}$ and $RC^{to}_{ECM}(T_i)$ into (4.21), the LCM/G-EDF has equal or better total utilization than ECM if $\alpha_{max} \leq 1$. But α_{max} is always less than or equal to 1. Claim follows. □

5.4.2 LCM/G-EDF versus Lock-free

As mentioned in Section 4.4, the retry-loop lock-free approach in [49] is the most relevant to our work. As lock-free instructions access only one object, then Θ_i^k for any s_i^k will be restricted to one object only (i.e., $\Theta_i^k = \theta_i^k$). Thus, transitive retry cannot happen, $\Theta_i^{ex} = \Theta_i$ and $\gamma_i^{ex} = \gamma_i$.

Claim 35. *Following notions in Section 4.3, total utilization of LCM/G-EDF is equal or better than that of [49]'s retry-loop lock-free approach if s_{max} does not exceed $r_{max}/\left(1+\alpha_{max}\right)$, where s_{max} is the length of longest transaction among all tasks, r_{max} is the maximum execution cost of a single iteration of any lock-free retry loop of any task, and $\alpha_{max} = max\left\{\alpha_{xy}^{kl}\right\}_{\forall s_x^k, s_y^l}$. With equal periods for conflicting tasks and high access times to shared objects, s_{max} can be much larger than r_{max}.*

Proof. Using Claim 32 and following the same steps of proof of Claim 16, Claim follows. □

5.4.3 LCM/G-EDF versus OMLP

Claim 36. *Following the same notations in Sections 4.3 and 4.5.1, total utilization of LCM/G-EDF is equal or better than total utilization of OMLP if*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_{min}\left(2m-1\right)}{\left(\left(1+\alpha_{max}\right)N_{max}+1\right)\left(n-1\right)\Phi_{max}} \tag{5.12}$$

where $\alpha_{max} = max\left\{\alpha_{xy}^{kl}\right\}_{\forall s_x^k, s_y^l}$. *As number of atomic sections in each task increases, all tasks have equal periods and equal number of atomic sections,* α_{max} *approaches 0, and number of processors is at least equal to half number of tasks, then* s_{max} *can be at least equal to* L_{max} *with total utilization of LCM/G-EDF equal or better than total utilization of OMLP.*

Proof. $\alpha_{max} \geq 0$ as defined in Section 5.1.1. Using Claim 32 and following the same steps of proof of Claim 18, Claim follows. □

5.4.4 LCM/G-EDF versus RNLP

Claim 37. *Following the same notations in Sections 4.3 and 4.5.4, total utilization of LCM/G-EDF is equal or better than total utilization of RNLP if*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_{min}\left(2m-1\right)}{\left(\left(1+\alpha_{max}\right)N_{max}+1\right)\left(n-1\right)\Phi_{max}} \tag{5.13}$$

where $\alpha_{max} = max\left\{\alpha_{xy}^{kl}\right\}_{\forall s_x^k, s_y^l}$. *As number of atomic sections in each task increases, all tasks have equal periods and equal number of atomic sections,* α_{max} *approaches 0, and number of processors is at least equal to half number of tasks, then* s_{max} *can be at least equal to* L_{max} *with total utilization of LCM/G-EDF equal or better than total utilization of RNLP.*

Proof. $\alpha_{max} \geq 0$ as defined in Section 5.1.1. Using Claim 32 and following the same steps of proof of Claim 20, Claim follows. □

5.5 Retry Cost and Response Time of LCM/G-RMA

Claim 38. *Under LCM/G-RMA, the total retry cost suffered by all transactions in any job* $\tau_i^x \in \tau_i$ *during interval* $L \leq T_i$ *due to direct and indirect conflict with other transactions is* *upper bounded by:*

$$\begin{aligned} RC_i\left(L\right) \leq & \left(\sum_{\tau_j \in \gamma_i^{ex},\, p_j > p_i} \left(\left\lceil \frac{L}{T_j} \right\rceil + 1 \right) \sum_{s_j^l, \left(\Theta = \Theta_i^{ex} \cap \Theta_j^l\right) \neq \emptyset} len\left(s_j^l + \alpha_{max}^{jl} s_{max}\left(\Theta\right) \right) \right) \\ & + \left(\sum_{\tau_z \in \gamma_i^{ex},\, p_z < p_i} \left(\left\lceil \frac{L}{T_z} \right\rceil + 1 \right) \sum_{s_z^l, \left(\Theta = \Theta_i^{ex} \cap \Theta_z^l\right) \neq \emptyset} len\left(\left(1 - \alpha_{min}^{zl}\right) s_z^l \right) \right) \end{aligned} \quad (5.14)$$

where $s_{max}\left(\Theta\right) \notin s_j$ *and* α_{max}^{jl} *is given by (5.2) due to interference of the lower priority* $s_{max}\left(\Theta\right)$ *by the higher priority* s_j^l*.* α_{min}^{zl} *is the minimum* α_{zx}^{ly}*- calculated by (5.2)- that results from delay of a any higher priority transaction* s_x^y *by the lower priority* s_z^l*.*

Proof. Proof follows from Claims 27, 28 and proof of Claim 12. □

Claim 39. *Under LCM/G-RMA, the total retry cost suffered by all transactions in any job* $\tau_i^x \in \tau_i$ *during an interval* $L \leq T_i$ *due to release of jobs with higher priority than* τ_i^x *is upper bounded by*

$$RC_{i_{re}}(L) = \sum_{\forall \tau_j,\, p_j > p_i} \left(\left\lceil \frac{L}{T_j} \right\rceil s_{i_{max}} \right) \quad (5.15)$$

Proof. LCM/G-RMA and RCM has the same pattern for release of jobs. Thus, proof is the same as proof of Claim 13. Claim follows. □

Claim 40. *Under LCM/G-RMA, the total retry cost suffered by all transactions in any job* $\tau_i^x \in \tau_i$ *during an interval* $L \leq T_i$ *is upper bounded by:*

$$RC_{i_{to}}(L) = RC_i(L) + RC_{i_{re}}(L) \quad (5.16)$$

where $RC_i(L)$ *is the maximum retry cost resulting from conflict between transactions in* τ_i^x *and transactions of other jobs.* $RC_i(L)$ *is calculated by (5.14).* $RC_{i_{re}}(L)$ *is the maximum retry cost resulting from the release of higher priority jobs, which preempt transactions in* τ_i^x*.* $RC_{i_{re}}(L)$ *is calculated by (5.15).*

Proof. Using Claims 38, 39 and proof of Claim 14, Claim follows. □

Claim 41. *Under LCM/G-RMA, maximum response time of any job* $\tau_i^x \in \tau_i$ *is upper bounded by Claim 15 where* $RC_{i_{to}}(R_i^{up})$ *is upper bounded by (5.16).*

Proof. Proof follows directly from Claim 40 and proof of Claim 15. □

5.6 Total utilization of LCM/G-RMA

As in Section 5.4, we compare the total utilization of LCM/G-RMA against RCM (Chapter 4), lock-free [49] and locking protocols (i.e., OMLP [22,29] and RNLP [149]) to understand when LCM/G-RMA will perform better.

5.6.1 LCM/G-RMA versus RCM

Claim 42. *Following notions in Section 4.3, LCM/G-RMA's total utilization is equal or better than RCM if:*

$$\frac{1-\alpha_{min}}{1-\alpha_{max}} \leq \frac{\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j > p_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i^{ex}\right) \neq \emptyset} \right)}{2 \sum_{\forall \tau_z \in \gamma_i^{ex}, p_z < p_i} \sum_{\forall s_z^l, \left(\Theta = \Theta_z^l \cap \Theta_i^{ex}\right) \neq \emptyset}} \tag{5.17}$$

where $\alpha_{max} = max\{\alpha_{xy}^{kl}\}_{\forall s_x^k, s_y^l}$, $\alpha_{min} = min\{\alpha_{xy}^{kl}\}_{\forall s_x^k, s_y^l}$.

Proof. Let $RC_{LCM/G-RMA}^{to}$ be the total retry cost for any job of τ_i under LCM/G-RMA. $RC_{LCM/G-RMA}^{to}$ is given by (5.16) and upper bounded by:

$$\begin{aligned} & (1+\alpha_{max}) \left(\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j > p_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i^{ex}\right) \neq \emptyset} \right) s_{max} \right) \\ + \quad & (1-\alpha_{min}) \left(\sum_{\forall \tau_z \in \gamma_i^{ex}, p_z < p_i} \left(\left(\left\lceil \frac{T_i}{T_z} \right\rceil + 1 \right) \sum_{\forall s_z^l, \left(\Theta = \Theta_z^l \cap \Theta_i^{ex}\right) \neq \emptyset} \right) s_{max} \right) \\ + \quad & \left(\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil \right) s_{max} \end{aligned} \tag{5.18}$$

Let RC_{RCM}^{to} be the total retry cost for any job of τ_i under RCM. RC_{RCM}^{to} is given by (4.18) and upper bounded by:

$$\left(\left(2 \sum_{\forall \tau_j \in \gamma_i^{ex}, p_j > p_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i^{ex}\right) \neq \emptyset} \right) \right) + \left(\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil \right) \right) s_{max} \tag{5.19}$$

$\left\lceil \frac{T_i}{T_z} \right\rceil = 1$, $\forall \tau_i, \tau_z$ where $p_z < p_i$ because $T_i < T_z$ by definition of G-RMA and implicit deadline tasks. By substitution of (5.18) and (5.19) into (4.21), Claim follows. □

5.6.2 LCM/G-RMA versus Lock-free

As mentioned in Section 4.4, the retry-loop lock-free approach in [49] is the most relevant to our work. As lock-free instructions access only one object, then Θ_i^k for any s_i^k will be restricted to one object only (i.e., $\Theta_i^k = \theta_i^k$). Thus, transitive retry cannot happen, $\Theta_i^{ex} = \Theta_i$ and $\gamma_i^{ex} = \gamma_i$.

Claim 43. *Following notions in Section 4.3, total utilization of LCM/G-RMA's is equal or better than that of [49]'s retry-loop lock-free approach if s_{max} does not exceed $r_{max}/\left(1+\alpha_{max}\right)$, where s_{max} is the length of longest transaction among all tasks, r_{max} is the maximum execution cost of a single iteration of any lock-free retry loop of any task, and $\alpha_{max} = max\left\{\alpha_{xy}^{kl}\right\}_{\forall s_x^k, s_y^l}$. With high access times to shared objects, s_{max} can be much larger than r_{max}.*

Proof. Let $RC_{LCM/G-RMA}^{to}$ be the total retry cost for any job of τ_i under LCM/G-RMA. $RC_{LCM/G-RMA}^{to}$ is given by (5.16) and upper bounded by (5.18) where γ_i^{ex} is replaced with γ_i and Φ_i^{ex} is replaced with Φ_i. Let LRC_{to} be the total retry cost for any job of τ_i under retry-loop lock-free with G-RMA. LRC_{to} is upper bounded by (4.31). Similar to proof of Claim 17, total utilization of LCM/G-RMA is equal or better than total utilization of retry-loop lock-free if for each τ_i:

$$\begin{aligned}
& \left(1+\alpha_{max}\right)\left(\sum_{\forall\tau_j\in\gamma_i,p_j>p_i}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\sum_{\forall s_j^l,\left(\Theta=\Theta_j^l\cap\Theta_i\right)\neq\emptyset}\right)s_{max}\right)\\
+ & \left(1-\alpha_{min}\right)\left(\sum_{\forall\tau_j\in\gamma_i,p_j<p_i}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\sum_{\forall s_j^l,\left(\Theta=\Theta_j^l\cap\Theta_i\right)\neq\emptyset}\right)s_{max}\right)\\
+ & \left(\sum_{\forall\tau_j,p_j>p_i}\left\lceil\frac{T_i}{T_j}\right\rceil\right)s_{max}\\
\leq & \left(\left(\sum_{\forall\tau_j\in\gamma_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\beta_{ij}\right)+\left(\sum_{\forall\tau_j,p_j>p_i}\left\lceil\frac{T_i}{T_j}\right\rceil\right)\right)r_{max}
\end{aligned} \tag{5.20}$$

By definition of G-RMA and implicit deadline tasks, $\left\lceil\frac{T_i}{T_j}\right\rceil = 1$, $\forall\tau_i$, τ_j where $p_j < p_i$. So, (5.20) becomes

$$\begin{aligned}
& \left(1+\alpha_{max}\right)\left(\sum_{\forall\tau_j\in\gamma_i,p_j>p_i}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\sum_{\forall s_j^l,\left(\Theta=\Theta_j^l\cap\Theta_i\right)\neq\emptyset}\right)s_{max}\right)\\
+ & 2\left(1-\alpha_{min}\right)\left(\sum_{\forall\tau_j\in\gamma_i,p_j<p_i}\left(\sum_{\forall s_j^l,\left(\Theta=\Theta_j^l\cap\Theta_i\right)\neq\emptyset}\right)s_{max}\right)\\
+ & \left(\sum_{\forall\tau_j,p_j>p_i}\left\lceil\frac{T_i}{T_j}\right\rceil\right)s_{max}\\
\leq & \left(\left(\sum_{\forall\tau_j\in\gamma_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\beta_{ij}\right)+\left(\sum_{\forall\tau_j,p_j>p_i}\left\lceil\frac{T_i}{T_j}\right\rceil\right)\right)r_{max}
\end{aligned} \tag{5.21}$$

The set of tasks $\{\tau_j|\tau_j\neq\tau_i\}$ can be devided into four sets dependinong on priority and object sharing between τ_i and τ_j. So, $\{\tau_j|\tau_j\neq\tau_i\} = \{\tau_l\}\cup\{\tilde{\tau}_l\}\cup\{\tau_h\}\cup\{\tilde{\tau}_h\}$, where:

- $\{\tau_l\} = \{\tau_j|\left(\tau_j\neq\tau_i\right)\wedge\left(p_j<p_i\right)\wedge\left(\tau_j\in\gamma_i\right)\}$ is the set of tasks τ_j other than τ_i where τ_j has direct conflict with τ_i and priority of τ_j is lower than priority of τ_i. Let β_{il}^* be the number of transactions in $\tau_l\in\{\tau_l\}$ that has direct conflict with τ_i (i.e., $\beta_{il}^* = \sum_{\forall s_l^x,\left(\Theta=\Theta_l^x\cap\Theta_i\right)\neq\emptyset}$). Let β_{il} be the number of times a lower priority job of τ_l accesses shared objects with a higher priority job of τ_i using retry-loop lock-free [49]. As one object can be accessed multiple times within the same transaction, and lock-free instruction accesses one object only once, then $\beta_{il}\geq\beta_{il}^*$.

- $\{\tilde{\tau}_l\} = \{\tau_j | (\tau_j \neq \tau_i) \wedge (p_j < p_i) \wedge (\tau_j \notin \gamma_i)\}$ is the set of tasks τ_j other than τ_i where τ_j has no direct conflict with τ_i and priority of τ_j is lower than priority of τ_i.
- $\{\tau_h\} = \{\tau_j | (\tau_j \neq \tau_i) \wedge (p_j > p_i) \wedge (\tau_j \in \gamma_i)\}$ is the set of tasks τ_j other than τ_i where τ_j has direct conflict with τ_i and priority of τ_j is higher than priority of τ_i. Let β^*_{ih} be the number of transactions in $\tau_h \in \{\tau_h\}$ that has direct conflict with τ_i (i.e., $\beta^*_{ih} = \sum_{\forall s_h^x, (\Theta = \Theta_h^x \cap \Theta_i) \neq \emptyset}$). Let β_{ih} be the number of times a higher priority job of τ_h accesses shared objects with a lower priority job of τ_i using retry-loop lock-free [49]. As one object can be accessed multiple times within the same transaction, and lock-free instruction accesses one object only once, then $\beta_{ih} \geq \beta^*_{ih}$.
- $\{\tilde{\tau}_h\} = \{\tau_j | (\tau_j \neq \tau_i) \wedge (p_j > p_i) \wedge (\tau_j \notin \gamma_i)\}$ is the set of tasks τ_j other than τ_i where τ_j has no direct conflict with τ_i and priority of τ_j is higher than priority of τ_i.

Thus, (5.21) becomes

$$
\begin{aligned}
& (1+\alpha_{max}) \left(\sum_{\forall \tau_h \in \{\tau_h\}} \left(\left(\left\lceil \frac{T_i}{T_h} \right\rceil + 1 \right) \beta^*_{ih} \right) s_{max} \right) \\
+ \quad & 2(1-\alpha_{min}) \left(\sum_{\forall \tau_l \in \{\tau_l\}} \beta^*_{il} s_{max} \right) \\
+ \quad & \left(\left(\sum_{\forall \tau_h \in \{\tau_h\}} \left\lceil \frac{T_i}{T_h} \right\rceil \right) + \left(\sum_{\forall \tilde{\tau}_h \in \{\tilde{\tau}_h\}} \left\lceil \frac{T_i}{\tilde{T}_h} \right\rceil \right) \right) s_{max} \\
\leq \quad & \left(\left(\sum_{\forall \tau_h \in \{\tau_h\}} \left(\left\lceil \frac{T_i}{T_h} \right\rceil + 1 \right) \beta_{ih} \right) + \left(2 \sum_{\forall \tau_l \in \{\tau_l\}} \beta_{il} \right) \right) r_{max} \\
+ \quad & \left(\left(\sum_{\forall \tau_h \in \{\tau_h\}} \left\lceil \frac{T_i}{T_h} \right\rceil \right) + \left(\sum_{\forall \tilde{\tau}_h \in \{\tilde{\tau}_h\}} \left\lceil \frac{T_i}{\tilde{T}_h} \right\rceil \right) \right) r_{max}
\end{aligned} \tag{5.22}
$$

$$
\begin{aligned}
\therefore \quad & \sum_{\forall \tau_h \in \{\tau_h\}} \left((1+\alpha_{max}) \left(\left(\left\lceil \frac{T_i}{T_h} \right\rceil + 1 \right) \beta^*_{ih} \right) + \left\lceil \frac{T_i}{T_h} \right\rceil \right) s_{max} \\
+ \quad & 2(1-\alpha_{min}) \left(\sum_{\forall \tau_l \in \{\tau_l\}} \beta^*_{il} s_{max} \right) \\
+ \quad & \left(\left(\sum_{\forall \tilde{\tau}_h \in \{\tilde{\tau}_h\}} \left\lceil \frac{T_i}{\tilde{T}_h} \right\rceil \right) \right) s_{max} \\
\leq \quad & \sum_{\forall \tau_h \in \{\tau_h\}} \left(\left(\left(\left\lceil \frac{T_i}{T_h} \right\rceil + 1 \right) \beta_{ih} \right) + \left\lceil \frac{T_i}{T_h} \right\rceil \right) r_{max} \\
+ \quad & \left(2 \sum_{\forall \tau_l \in \{\tau_l\}} \beta_{il} \right) r_{max} \\
+ \quad & \left(\sum_{\forall \tilde{\tau}_h \in \{\tilde{\tau}_h\}} \left\lceil \frac{T_i}{\tilde{T}_h} \right\rceil \right) r_{max}
\end{aligned} \tag{5.23}
$$

(5.23) is satisfied if for each τ_i:

- $$\frac{s_{max}}{r_{max}} \leq \frac{\sum_{\forall \tau_h \in \{\tau_h\}} \left(\left(\left(\left\lceil \frac{T_i}{T_h} \right\rceil + 1 \right) \beta_{ih} \right) + \left\lceil \frac{T_i}{T_h} \right\rceil \right)}{\sum_{\forall \tau_h \in \{\tau_h\}} \left((1+\alpha_{max}) \left(\left(\left\lceil \frac{T_i}{T_h} \right\rceil + 1 \right) \beta^*_{ih} \right) + \left\lceil \frac{T_i}{T_h} \right\rceil \right)} \tag{5.24}$$

To find the lower bound over s_{max}/r_{max} that satisfies (5.24), let β_{ih} assumes its minimum value (i.e., $\beta_{ih} = \beta^*_{ih}$). Thus, (5.24) is satisfied if

$$\begin{aligned}\frac{s_{max}}{r_{max}} &\leq \frac{\sum_{\forall\tau_h\in\{\tau_h\}}\left(\left(\left(\left\lceil\frac{T_i}{T_h}\right\rceil+1\right)\beta^*_{ih}\right)+\left\lceil\frac{T_i}{T_h}\right\rceil\right)}{\sum_{\forall\tau_h\in\{\tau_h\}}(1+\alpha_{max})\left(\left(\left(\left\lceil\frac{T_i}{T_h}\right\rceil+1\right)\beta^*_{ih}\right)+\left\lceil\frac{T_i}{T_h}\right\rceil\right)} \\ &= \frac{1}{1+\alpha_{max}} \leq \frac{1}{2} \end{aligned} \tag{5.25}$$

To find the upper bound over s_{max}/r_{max} that satisfies (5.24), let $\beta_{ih} >> (1+\alpha_{max})\,\beta^*_{ih}$. Thus, s_{max} can be much larger than r_{max}.

- $$2(1-\alpha_{min})\left(\sum_{\forall\tau_l\in\{\tau_l\}}\beta^*_{il}s_{max}\right) \leq \left(2\sum_{\forall\tau_l\in\{\tau_l\}}\beta_{il}\right)r_{max}$$

$$\therefore \frac{s_{max}}{r_{max}} \leq \frac{\sum_{\forall\tau_l\in\{\tau_l\}}\beta_{il}}{(1-\alpha_{min})\left(\sum_{\forall\tau_l\in\{\tau_l\}}\beta^*_{il}\right)} \tag{5.26}$$

To find the lower bound over s_{max}/r_{max} that satisfies (5.26), let β_{il} assumes its minimum value (i.e., $\beta_{il} = \beta^*_{il}$). Thus, (5.26) is satisfied if

$$\frac{s_{max}}{r_{max}} \leq \frac{1}{1-\alpha_{min}} \leq 1 \tag{5.27}$$

To find the upper bound over s_{max}/r_{max} that satisfies (5.26), let $\beta_{il} >> (1-\alpha_{min})\,\beta^*_{il}$. Thus, s_{max} can be much larger than r_{max}.

- $$\left(\sum_{\forall\tilde{\tau}_h\in\{\tilde{\tau}_h\}}\left\lceil\frac{T_i}{\tilde{T}_h}\right\rceil\right)s_{max} \leq \left(\sum_{\forall\tilde{\tau}_h\in\{\tilde{\tau}_h\}}\left\lceil\frac{T_i}{\tilde{T}_h}\right\rceil\right)r_{max} \tag{5.28}$$

$$\therefore \frac{s_{max}}{r_{max}} \leq 1$$

By taking the minimum lower bound and the maximum upper bound from the previous cases, Claim follows. □

5.6.3 LCM/G-RMA versus OMLP

Claim 44. *Following the same notations in Sections 4.3 and 4.5.1, total utilization of LCM/G-RMA is equal or better than total utilization of OMLP if*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_{min}(2m-1)}{(n-1)\left((1+\alpha_{max})\left((\Phi_{max}+1)N_{max}\right)+2(1-\alpha_{min})N_{max}+\Phi_{max}\right)} \tag{5.29}$$

where $\alpha_{max} = max\left\{\alpha_{xy}^{kl}\right\}_{\forall s_x^k, s_y^l}$ *and* $\alpha_{min} = min\{\alpha_{xy}^{kl}\}_{\forall s_x^k, s_y^l}$. *As number of atomic sections in each task increases, all tasks have equal periods and equal number of atomic sections,* α_{max} *approaches* α_{min}*, and number of processors is at least double number of tasks, then* s_{max} *can be at least equal to* L_{max} *with total utilization of LCM/G-RMA equal or better than total utilization of OMLP.*

Proof. $\alpha_{max} \geq \alpha_{min}$ by definition. Substitute $RC_A(T_i)$ in (4.37) by (5.18). Following the same steps in proof of Claim 18, Claim follows. □

5.6.4 LCM/G-RMA versus RNLP

Claim 45. *Following the same notations in Sections 4.3 and 4.5.4, total utilization of LCM/G-RMA is equal or better than total utilization of RNLP if*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_{min}\left(2m-1\right)}{\left(n-1\right)\left(\left(1+\alpha_{max}\right)\left(\left(\Phi_{max}+1\right)N_{max}\right)+2\left(1-\alpha_{min}\right)N_{max}+\Phi_{max}\right)} \tag{5.30}$$

where $\alpha_{max} = max\left\{\alpha_{xy}^{kl}\right\}_{\forall s_x^k, s_y^l}$ *and* $\alpha_{min} = min\{\alpha_{xy}^{kl}\}_{\forall s_x^k, s_y^l}$. *As number of atomic sections in each task increases, all tasks have equal periods and equal number of atomic sections,* α_{max} *approaches* α_{min}*, and number of processors is at least double number of tasks, then* s_{max} *can be at least equal to* L_{max} *with total utilization of LCM/G-RMA equal or better than total utilization of RNLP.*

Proof. $\alpha_{max} \geq \alpha_{min}$ by definition. Substitute $RC_A(T_i)$ in (4.37) by (5.18). Following the same steps of proof of Claim 20, Claim follows. □

5.7 Conclusions

In ECM and RCM, a task incurs at most $2s_{max}$ retry cost for each of its atomic section due to conflict with another task's atomic section. With LCM, this retry cost is reduced to $(1+\alpha_{max})s_{max}$ for each aborted atomic section. In ECM and RCM, higher priority tasks are not delayed due to lower priority tasks, whereas in LCM, they are. In LCM/G-EDF, delay due to a lower priority job is encountered only from a task τ_j's last job instance during τ_i's period. Contribution of a transaction s_j^l to the retry cost of a lower priority transaction is higher than delay caused by s_j^l to a higher priority transaction. Thus, under LCM/G-EDF, each transaction is assumed to contribute in the abort and retry of a lower priority transaction. Hence, delay of higher priority transactions due to lower priority transactions is ignored under LCM/G-EDF. This is not the case with LCM/G-RMA, because of fixed priority under G-RMA.

Performance of LCM/G-EDF is always equal or better than ECM's in terms of total utilization. Whereas, total utilization of LCM/G-RMA is equal or better than RCM's depending on α_{min} and α_{max}. Total utilization of LCM (with G-EDF and G-RMA) is equal or better than total utilization of retry-loop lock-free if s_{max} does not exceed $r_{max}/\left(1+\alpha_{max}\right)$. With high number of object access within each transaction, s_{max} can be much larger than r_{max} with equal or better total utilization for LCM (with G-EDF and G-RMA) than total utilization of retry-loop lock-free.

Total utilization of LCM was compared against real-time locking protocols (i.e., OMLP and RNLP) under G-EDF and G-RMA. As number of atomic sections in each task increases, all tasks have equal periods and equal number of atomic sections, α_{max} approaches 0, and number of processors is at least equal to half number of tasks, then s_{max} can be at least equal to L_{max} with total utilization of LCM/G-EDF equal or better than total utilization of OMLP and RNLP. The same results apply to total utilization comparison between LCM/G-RMA and locking protocols except that α_{max} approaches α_{min} and number of processors should be at least double number of tasks.

Chapter 6

The PNF Contention Manager

In this chapter, we present a novel contention manager for resolving transactional conflicts, called PNF [57]. We upper bound transactional retries and task response times under PNF, when used with the G-EDF and G-RMA schedulers. We formally identify the conditions under which PNF outperforms previous real-time STM contention managers, lock-free and locking protocols.

The rest of this Chapter is organized as follows: Section 6.1 discusses limitations of previous contention managers and the motivation to PNF. Section 6.2 give a formal description of PNF. Section 6.3 derives PNF's properties. We upper bound retry cost and response time under PNF in Section 6.4. Performance comparison between PNF and other synchronization techniques in terms of total utilization is given in Section 6.5. We conclude Chapter in Section 6.6.

6.1 Limitations of ECM, RCM, and LCM

With multiple objects per transaction, ECM, RCM (Chapter 4) and LCM (Chapter 5) face transitive retry as shown by Claims 1, 11 and 22. Thus, a transaction s_i^k can abort and retry due to another transaction s_j^l where $\Theta_i^k \cap \Theta_j^l = \emptyset$. Retry cost and response time analysis-presented in Chapters 4 and 5- extend the set of objects accessed by any task τ_i to include any object that can cause direct or indirect(transitive) retry to any transaction in τ_i. However, this solution may over-extend the set of conflicting objects, and may even contain all objects accessed by all tasks.

In addition to the *transitive retry* problem, retrying higher priority transactions can prevent lower priority tasks from running. This happens when all processors are busy with higher priority jobs. When a transaction retries, the processor time is wasted. Thus, it would be better to give the processor to some other task.

Essentially, what we present is a new contention manager that avoids the effect of transitive retry. We call it, Priority contention manager with Negative values and First access (or PNF). PNF also tries to enhance processor utilization. This is done by allocating processors to jobs with non-retrying transactions. PNF is described in Section 6.2.

6.2 The PNF Contention Manager

Algorithm 4 describes PNF. It manages two sets. The first is the m-set, which contains at most m non-conflicting transactions, where m is the number of processors, as there cannot be more than m executing transactions (or generally, m executing jobs) at the same time. When a transaction is entered in the m-set, it executes non-preemptively and no other transaction can abort it. A transaction in the m-set is called an *executing transaction*. This means that, when a transaction is executing before the arrival of higher priority conflicting transactions, then the one that started executing first will be committed (Step 8) (hence the term "First access" in the algorithm's name). The second set is the n-set, which holds the transactions that are retrying because of a conflict with one or more of the executing transactions (Step 6), where n stands for the number of tasks in the system. Transactions in the n-set are known as *retrying transaction*. n-set also holds transactions that cannot currently execute, because processors are busy, either due to processing executing transactions and/or higher priority jobs. Any transaction in the n-set is assigned a temporal priority of -1 (Step 7) (hence the word "Negative" in the algorithm's name). A negative priority is considered smaller than any normal priority, and a transaction continues to hold this negative priority until it is moved to the m-set, where it restores its normal priority.

A job τ_x^y holding a transaction in the n-set can be preempted by any other job τ_z^l with normal priority, even if τ_z^l does not have transactions conflicting with τ_x^y. Hence, the n-set is of length n, as there can be at most n jobs. Transactions in the n-set whose jobs have been preempted are called *preempted transactions*. The n-set list keeps track of preempted transactions, because as it will be shown, all preempted and non-preempted transactions in the n-set are examined when any executing transaction commits. Then, one or more transactions are selected from the n-set to be executing transactions. If a retrying transaction is selected as an executing transaction, the task that owns the retrying transaction regains its priority.

When a new transaction is released, and if it does not conflict with any of the executing transactions (Step 1), then it will allocate a slot in the m-set and becomes an executing transaction. When this transaction is released (i.e., its containing task is already allocated to a processor), it will be able to access a processor immediately. This transaction may have a conflict with any of the transactions in the n-set. However, since transactions in the n-set have priorities of -1, they cannot prevent this new transaction from executing if it does not

[1] An idle processor or at least one that runs a non-atomic section task with priority lower than the task holding $n(z)$.

Algorithm 4: PNF

Data: *Executing Transaction:* is one that cannot be aborted by any other transaction, nor preempted by a higher priority task;
m-set: m-length set that contains only non-conflicting executing transactions;
n-set: n-length set that contains retrying transactions for n tasks in non-increasing order of priority;
n(z): transaction at index z of the n-set;
s_i^k: a newly released transaction;
s_j^l: one of the executing transactions;
Result: atomic sections that will commit

```
1  if s_i^k does not conflict with any executing transaction then
2      Assign s_i^k as an executing transaction;
3      Add s_i^k to the m-set;
4      Select s_i^k to commit
5  else
6      Add s_i^k to the n-set according to its priority;
7      Assign temporary priority -1 to the job that owns s_i^k ;
8      Select transaction(s) conflicting with s_i^k for commit;
9  end
10 if s_j^l commits then
11     for z=1 to size of n-set do
12         if n(z) does not conflict with any executing transaction then
13             if processor available[1] then
14                 Restore priority of task owning n(z);
15                 Assign n(z) as executing transaction;
16                 Add n(z) to m-set and remove it from n-set;
17                 Select n(z) for commit;
18             else
19                 Wait until processor available
20             end
21         end
22         move to the next n(z);
23     end
24 end
```

conflict with any of the executing transactions.

When one of the executing transactions commits (Step 10), it is time to select one of the n-set transactions to commit. The n-set is traversed from the highest priority to the lowest priority (priority here refers to the original priority of the transactions, and not -1) (Step 11). If an examined transaction in the n-set, s_h^b, does not conflict with any executing transaction (Step 12), and there is an available processor for it (Step 13) ("available" means either an idle processor, or one that is executing a job of lower priority than s_h^b), then s_h^b is moved from the n-set to the m-set as an executing transaction and its original priority is restored. If s_h^b is added to the m-set, the new m-set is compared with other transactions in the n-set with lower priority than s_h^b. Hence, if one of the transactions in the n-set, s_d^g, is of lower priority than s_h^b and conflicts with s_h^b, it will remain in the n-set.

The choice of the new transaction from the n-set depends on the original priority of transactions (hence the term "P" in the algorithm name). The algorithm avoids interrupting an already executing transaction to reduce its retry cost. In the meanwhile, it tries to avoid delaying the highest priority transaction in the n-set when it is time to select a new one to commit, even if the highest priority transaction arrives after other lower priority transactions in the n-set.

6.2.1 Illustrative Example

We illustrate PNF with an example. We use the following notions: $s_a^b(\theta_1, \theta_2, \theta_3)$ means that s_a^b accesses objects $\theta_1, \theta_2, \theta_3$. If $s_a^b \in \tau_a^j$, $\therefore p_o(s_a^b) = p_a^j$, where $p_o(s_a^b)$ is the original priority of s_a^b. $p(s_a^b) = -1$, if s_a^b is a retrying transaction; $p(s_a^b) = p_o(s_a^b)$ otherwise. m-set= $\{s_a^b, s_i^k\}$ means that the m-set contains transactions s_a^b and s_i^k regardless of their order. n-set= $\{s_a^b, s_i^k\}$ means that the n-set contains transactions s_a^b and s_i^k in that order, where $p_o(s_a^b) > p_o(s_i^k)$. m-set (n-set) = $\{\phi\}$ means that m-set (n-set) is empty. Assume there are five processors.

1. Initially, m-set= n-set= $\{\phi\}$. $s_a^b(\theta_1, \theta_2) \in \tau_a^b$ is released and checks m-set for conflicting transactions. As m-set is empty, s_a^b finds no conflict and becomes an executing transaction. s_a^b is added to m-set. m-set= $\{s_a^b\}$ and n-set= $\{\phi\}$. s_a^b is executing on processor 1.
2. $s_c^d(\theta_3, \theta_4) \in \tau_c^d$ is released and checks m-set for conflicting transactions. s_c^d does not conflict with s_a^b as they access different objects. s_c^d becomes an executing transaction and is added to m-set. m-set= $\{s_a^b, s_c^d\}$ and n-set= $\{\phi\}$. s_c^d is executing on processor 2.
3. $s_e^f(\theta_1, \theta_5) \in \tau_e^f$ is released and $p_o(s_e^f) < p_o(s_a^b)$. s_e^f conflicts with s_a^b when it checks m-set. s_e^f is added to n-set and becomes a retrying transaction. $p(s_e^f)$ becomes -1. m-set= $\{s_a^b, s_c^d\}$ and n-set= $\{s_e^f\}$. s_e^f is retrying on processor 3.
4. $s_g^h(\theta_1, \theta_6) \in \tau_g^h$ is released and $p_o(s_g^h) > p_o(s_a^b)$. s_g^h conflicts with s_a^b. Though s_g^h is of higher priority than s_a^b, s_a^b is an executing transaction. So s_a^b runs non-preemptively. s_g^h is added to n-set before s_e^f, because $p_o(s_g^h) > p_o(s_e^f)$. $p(s_g^h)$ becomes -1. m-set= $\{s_a^b, s_c^d\}$ and n-set= $\{s_g^h, s_e^f\}$. s_g^h is retrying on processor 4.
5. $s_i^j(\theta_5, \theta_7) \in \tau_i^j$ is released. $p_o(s_i^j) < p_o(s_e^f)$. s_i^j does not conflict with any transaction in m-set. Though s_i^j conflicts with s_e^f and $p_o(s_i^j) < p_o(s_e^f) < p_o(s_g^h)$, s_e^f and s_g^h are retrying transactions. s_i^j becomes an executing transaction and is added to m-set. m-set= $\{s_a^b, s_c^d, s_i^j\}$ and n-set= $\{s_g^h, s_e^f\}$. s_i^j is executing on processor 5.
6. τ_k^l is released. τ_k^l does not access any object. $p_k^l < p_o(s_e^f) < p_o(s_g^h)$, but $p(s_e^f) = p(s_g^h) = -1$. Since there are no more processors, τ_k^l preempts τ_e^f, because the currently assigned priority to $\tau_e^f = p(s_e^f) = -1$ and $p_o(s_g^h) > p_o(s_e^f)$. τ_k^l is running on processor 3. This way, PNF optimizes processor usage. The m-set and n-set are not changed. Although s_e^f is preempted, n-set still records it, as s_e^f might be needed (as will be shown in the following steps).

7. s_i^j commits. s_i^j is removed from m-set. Transactions in n-set are checked from the first (highest p_o) to the last (lowest p_o) for conflicts against any executing transaction. s_g^h is checked first because $p_o(s_g^h) > p_o(s_e^f)$. s_g^h conflicts with s_a^b, so s_g^h cannot be an executing transaction. Now it is time to check s_e^f, even though s_e^f is preempted in step 6. s_e^f also conflicts with s_a^b, so s_e^f cannot be an executing transaction. m-set= $\{s_a^b, s_c^d\}$ and n-set= $\{s_g^h, s_e^f\}$. Now, s_e^f can be retrying on processor 5 if τ_i^j has finished execution. Otherwise, τ_i^j continues running on processor 5 and s_e^f is still preempted. This is because, $p(s_e^f) = -1$ and $p_i^j > p(s_e^f)$. Let us assume that τ_i^j is still running on processor 5.
8. s_a^b commits. s_a^b is removed from m-set. Transactions in n-set are checked as done in step 7. s_g^h does not conflict with any executing transaction any more. s_g^h becomes an executing transaction. s_g^h is removed from n-set and added to m-set, so m-set= $\{s_c^d, s_g^h\}$. Now, s_e^f is checked against the new m-set. s_e^f conflicts with s_g^h, so s_e^f cannot be an executing transaction. s_e^f can be retrying on processor 1 if τ_a^b has finished execution. Otherwise, s_e^f remains preempted, because $p(s_e^f) = -1$ and $p_a^b > p(s_e^f)$. n-set= $\{s_e^f\}$. Let us assume that τ_a^b is still running on processor 1.
9. s_g^h commits. s_g^h is removed from m-set. τ_g^h continues execution on processor 4. Transactions in n-set are checked again. s_e^f is the only retrying transaction in the n-set, and it does not conflict with any executing transactions. Now, the system has τ_a^b running on processor 1, s_c^d executing on processor 2, τ_k^l running on processor 3, τ_g^h running on processor 4, and τ_i^j running on processor 5. s_e^f can become an executing transaction if it can find a processor. Since p_i^j, $p_k^l < p_o(s_e^f)$, s_e^f can preempt the lowest in priority between τ_i^j and τ_k^l. s_e^f now becomes an executing transaction. s_e^f is removed from the n-set and added to the m-set. So, m-set= $\{s_c^d, s_e^f\}$ and n-set= $\{\phi\}$. If p_i^j, p_k^l were of higher priority than $p_o(s_e^f)$, then s_e^f would have remained in n-set until a processor becomes available.

The example shows that PNF avoids transitive retry. This is illustrated in step 5, where $s_i^j(\theta_5, \theta_7)$ is not affected by the retry of $s_e^f(\theta_1, \theta_5)$. The example also explains how PNF optimizes processor usage. This is illustrated in step 6, where the retrying transaction s_e^f is preempted in favor of τ_k^l.

6.3 Properties

Claim 46. *Transactions scheduled under PNF do not suffer from transitive retry.*

Proof. Proof is by contradiction. Assume that a transaction s_i^k is retrying because of a higher priority transaction s_j^l, which in turn is retrying because of another higher priority transaction s_z^h. Assume that s_i^k and s_z^h do not conflict, yet, s_i^k is transitively retrying due to s_z^h. Note that s_z^h and s_j^l cannot exit together in the m-set as they have shared objects. But they both can be in the n-set, as they can conflict with other *executing transactions*. We have three cases:

Case 1: Assume that s_z^h is an executing transaction. This means that s_j^l is in the n-set. When s_i^k arrives, by the definition of PNF, it will be compared with the m-set, which contains s_z^h. Now, it will be found that s_i^k does not conflict with s_z^h. Also, by the definition of PNF, s_i^k is not compared with transactions in the n-set. When s_i^k newly arrives, priorities of n-set transactions are lower than any normal priority. Therefore, as s_i^k does not conflict with any other executing transaction, it joins the m-set and becomes an *executing transaction*. This contradicts the assumption that s_i^k is transitively retrying because of s_z^h.

Case 2: Assume that s_z^h is in the n-set, while s_j^l is an executing transaction. When s_i^k arrives, it will conflict with s_j^l and joins the n-set. Now, s_i^k retries due to s_j^l, and not s_z^h. When s_j^l commits, the n-set is traversed from the highest priority transaction to the lowest one: if s_z^h does not conflict with any other executing transaction and there are available processors, s_z^h becomes an executing transaction. When s_i^k is compared with the m-set, it is found that it does not conflict with s_z^h. Additionally, if it also does not conflict with any other executing transaction and there are available processors, then s_i^k becomes an executing transaction. This means that s_i^k and s_z^h are executing concurrently, which violates the assumption of transitive retry.

Case 3: Assume that s_z^h and s_j^l both exist in the n-set. When s_i^k arrives, it is compared with the m-set. If s_i^k does not conflict with any executing transactions and there are available processors, then s_i^k becomes an executing transaction. Even though s_i^k has common objects with s_j^l, s_i^k is not compared with s_j^l, which is in the n-set. If s_i^k joins the n-set, it is because, it conflicts with one or more executing transactions, not because of s_z^h, which violates the transitive retry assumption. If the three transactions s_i^k, s_j^l and s_z^h exist in the n-set, and s_z^h is chosen as a new executing transaction, then s_j^l remains in the n-set. This leads to Case 1. If s_j^l is chosen, because s_z^h conflicts with another executing transaction and s_j^l does not, then this leads to Case 2. □

Claim 47. *The first access property of PNF prevents transitive retry.*

Proof. The proof is by contradiction. Assume that the retry cost of transactions in the absence of the first access property is the same as when first access exists. Now, assume that PNF is devoid of the first access property. This means that executing transactions can be aborted.

Assume three transactions s_i^k, s_j^l, and s_z^h, where s_z^h's priority is higher than s_j^l's priority, and s_j^l's priority is higher than s_i^k's priority. Assume that s_j^l conflicts with both s_i^k and s_z^h. s_i^k and s_z^h do not conflict together. If s_i^k arrives while s_z^h is an executing transaction and s_j^l exists in the n-set, then s_i^k becomes an executing transaction itself while s_j^l is retrying. If s_i^k did not commit at least when s_z^h commits, then s_j^l becomes an executing transaction. Due to the lack of the first access property, s_j^l will cause s_i^k to retry. So, the retry cost for s_i^k will be $len(s_z^h + s_j^l)$. This retry cost for s_i^k is the same if it had been transitively retrying because of s_z^h. This contradicts the first assumption. Claim follows. □

From Claims 46 and 47, PNF does not increase the retry cost of multi-object transactions. However, this is not the case for ECM, RCM and LCM as shown by Claims 1, 11 and 22.

Claim 48. *Under PNF, any job τ_i^x is not affected by the retry cost in any other job τ_j^l.*

Proof. As explained in Section 4, PNF assigns a temporary priority of -1 to any job that includes a retrying transaction. So, retrying transactions have lower priority than any other normal priority for any real-time task. When τ_i^x is released and τ_j^l has a retrying transaction, τ_i^x will have a higher priority than τ_j^l. Thus, τ_i^x can run on any available processor while τ_j^l is retrying one of its transactions. Claim follows. □

6.4 Retry Cost and Response Time Under PNF

We now derive an upper bound on the retry cost of any job τ_i^x under PNF during an interval $L \leq T_i$. Since all tasks are sporadic (i.e., each task τ_i has a minimum period T_i), T_i is the maximum study interval for each task τ_i.

Claim 49. *Under PNF, the maximum retry cost suffered by a transaction s_i^k due to a transaction s_j^l is $len(s_j^l)$.*

Proof. By PNF's definition, s_i^k cannot have started before s_j^l. Otherwise, s_i^k would have been an executing transaction and s_j^l cannot abort it. So, the earliest release time for s_i^k would have been just after s_j^l starts execution. Then, s_i^k would have to wait until s_j^l commits. Claim follows. □

Claim 50. *The retry cost for any job τ_i^x due to conflicts between its transactions and transactions of other jobs under PNF during an interval $L \leq T_i$ is upper bounded by:*

$$RC_i(L) \leq \sum_{\tau_j \in \gamma_i} \left(\sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i\right) \neq \emptyset} \left(\left(\left\lceil \frac{L}{T_j} \right\rceil + 1 \right) len\left(s_j^l\right) \right) \right) \tag{6.1}$$

Proof. Consider a transaction s_i^k belonging to job τ_i^x. Under PNF, higher priority transactions than s_i^k can become executing transaction before s_i^k. A lower priority transaction s_v^f can also become an executing transaction before s_i^k. This happens when s_i^k conflicts with any executing transaction while s_v^f does not. The worst case scenario for s_i^k occurs when s_i^k has to wait in the n-set, while all other conflicting transactions with s_i^k are chosen to be executing transactions. The maximum number of jobs of any task τ_j that can interfere with τ_i^x during interval L is $\left\lceil \frac{L}{T_j} \right\rceil + 1$. From the previous observations and Claim 49, Claim follows. □

Claim 51. *In contrast to ECM, RCM and LCM, release of any higher priority job τ_j^l during execution of a lower priority transaction s_i^k does not increase retry cost of s_i^k. Thus, $RC_{i_{re}}(L) = 0$ and $RC_{i_{to}}(L) = RC_i(L)$, where $L \leq T_i$ and $RC_i(L)$ is given by (6.1).*

Proof. Under PNF, executing transactions have higher priority than any other real-time task. Thus, release of a higher priority task τ_j^l will not preempt any executing transaction s_i^k. Retrying transactions are already retrying when higher priority tasks are released. When a retrying transaction s_i^k is chosen to be an executing transaction, and all processors are busy with executing transactions except the processor running τ_j^l, then τ_j^l is preempted in favour of the executing transaction s_i^k by definition of PNF. Thus, τ_j^l does not increase retry cost of s_i^k. Claim follows. □

Claim 52. *The maximum blocking time for any job in τ_i due to lower priority jobs during an interval $L \leq T_i$ is upper bounded by:*

$$D_i(L) \leq max_{\forall \tau_i^x \in \tau_i} \left(\left\lfloor \frac{1}{m} \sum_{\forall \tau_j^l, p_j^l < p_i^x} \left(\left(\left\lceil \frac{L}{T_j} \right\rceil + 1 \right) \sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset} len\left(s_j^h\right) \right) \right\rfloor \right) \tag{6.2}$$

During $D_i(L)$, all processors are unavailable for τ_i^x.

Proof. Under PNF, executing transactions are non preemptive. So, an executing transaction s_i^k can delay a higher priority job τ_i^x, where $p_o\left(s_i^k\right) < p_i^x$, if no other processors are available. Through this proof, we call an s_i^k with $p_o\left(s_i^k\right) < p_i^x$ an original lower priority transaction compared to priority of τ_i^x. An original lower priority executing transactions can be conflicting or non-conflicting with any transaction in τ_i^x. They also can exist when τ_i^x is newly released, or after that. So, we have the following cases:

Original lower priority conflicting transactions after τ_i^x is released: This case is already covered by the retry cost in (6.1).

Original lower priority conflicting transactions when τ_i^x is newly released: Each original lower priority conflicting transaction s_j^h will delay τ_i^x for $len(s_j^h)$. The effect of s_j^h is already covered by (6.1). Besides, (6.1) does not divide the retry cost by m as done in (6.2). Thus, the worst case scenario requires inclusion of s_j^h in (6.1), and not in (6.2).

Original lower priority non-conflicting transactions when τ_i^x is newly released: τ_i^x is delayed if there are no available processors for it. Otherwise, τ_i^x can run in parallel with these non-conflicting original lower priority transactions. Each original lower priority non-conflicting transaction s_j^h will delay τ_i^x for $len(s_j^h)$.

Original lower priority non-conflicting transactions after τ_i^x is released: This situation can happen if τ_i^x is not currently running any executing transaction. A retrying transaction s_i^k is chosen to be an executing transaction. All processors are busy with executing transactions except the processor running τ_i^x. Thus, τ_i^x is preempted in favour of executing transaction s_i^k. Otherwise, τ_i^x can run in parallel with these original lower priority non-conflicting transactions.

Each original lower priority non-conflicting transaction s_j^h will delay τ_i^x for $len(s_j^h)$.

From the previous cases, original lower priority non-conflicting transactions act as if they were higher priority jobs interfering with τ_i^x. So, the blocking time can be calculated by the interference workload given by Theorem 7 in [18]. Claim follows. □

Claim 53. *The response time R_i^{up} of a job τ_i^x under PNF/G-EDF is upper bounded by:*

$$R_i^{up} = c_i + RC_{i_{to}}(R_i^{up}) + D_i(R_i^{up}) + \left\lfloor \frac{1}{m} \sum_{\forall j \neq i} I_{ij}(R_i^{up}) \right\rfloor \tag{6.3}$$

where $RC_{i_{to}}(R_i^{up})$ is calculated by (6.1). $D_i(R_i^{up})$ is modified from (6.2) to fit G-EDF as follows:

$$D_i(R_i^{up}) \leq \left\lfloor \frac{1}{m} \sum_{\forall \tau_j} \begin{cases} 0 & , R_i^{up} \leq T_i - T_j \\ \sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset} len\left(s_j^h\right) & , R_i^{up} > T_i - T_j \end{cases} \right\rfloor \tag{6.4}$$

and $I_{ij}(R_i^{up})$ is calculated by (4.5).

Proof. Proof is similar to proof of Claim 10 except that: 1) Total retry cost given by (6.1) (due to Claim 51) and blocking time given by (6.2) are added to each c_i. 2) Due to Claim 48, each c_j is not changed to c_{ji}. G-EDF uses absolute deadlines for scheduling. This defines which jobs of the same task can be of lower priority than τ_i^x, and which will not. Any instance τ_j^h, released between $r_i^x - T_j$ and $d_i^x - T_j$, will be of higher priority than τ_i^x. Before $r_i^x - T_j$, τ_j^h would have finished before τ_i^x is released. After $d_i^x - T_j$, d_j^h would be greater than d_i^x. Thus, τ_j^h will be of lower priority than τ_i^x. So, during T_i, there can be only one instance τ_j^h of τ_j with lower priority than τ_i^x. τ_j^h is released between $d_i^x - T_j$ and d_i^x. Consequently, during $R_i^{up} < T_i - T_j$, no existing instance of τ_j is of lower priority than τ_i^x. Hence, 0 is used in the first case of (6.4). But if $R_i^{up} > T_i - T_j$, there can be only one instance τ_j^h of τ_j with lower priority than τ_i^x. Hence, $\left\lceil \frac{R_i^{up}}{T_i} \right\rceil + 1$ in (6.2) is replaced with 1 in the second case in (6.4). Claim follows. □

Claim 54. *The response time R_i^{up} of a job τ_i^x under PNF/G-RMA is upper bounded by:*

$$R_i^{up} = c_i + RC_i(R_i^{up}) + D_i(R_i^{up}) + \left\lfloor \frac{1}{m} \sum_{\forall j \neq i, p_j > p_i} I_{ij}(R_i^{up}) \right\rfloor \tag{6.5}$$

where $RC(R_i^{up})$ is calculated by (6.1), $D_i(R_i^{up})$ is calculated by (6.2), and $I_{ij}(R_i^{up})$ is calculated by (4.4).

Proof. Proof is same as of Claim 53, except that G-RMA assigns fixed priorities. Hence, (6.2) can be used directly for calculating $D_i(R_i^{up})$ without modifications. Claim follows. □

6.5 PNF versus Competitors

We now (formally) compare the performance of G-EDF (G-RMA) with PNF against ECM (Chapter 4), RCM (Chapter 4), LCM (Chapter 5), retry-loop lock-free [49] and locking protocols((i.e., OMLP [22,29] and RNLP [149]). Such a comparison will reveal when PNF outperforms others. Toward this, we compare the total utilization under G-EDF (G-RMA)/PNF, with that under the other synchronization methods as outlined in Section 4.3. Total utilization comparison between PNF and other synchronization techniques is done as in Sections 5.4 and 5.6 with the addition of $D_i(T_i)$ - given by (6.4) under G-EDF and (6.2) under G-RMA - to the inflated execution time of any job of τ_i under PNF.

6.5.1 PNF versus ECM

Claim 55. *Following notions in Section 4.3, total utilization of PNF/G-EDF is equal or better than ECM's if for each task τ_i total number of transactions in any task $\tau_j \neq \tau_i$ - that has no direct conflict with any transaction in τ_i - divided by number of processors is not greater than maximum number of jobs- with higher priority than current job of τ_i - that can be released during T_i.*

Proof. Proof follows from proof of Claim 34 with the following modification: Under PNF, c_i is inflated with $RC^{to}_{PNF/G-EDF}(T_i)$ given by (6.1) and $D_i(T_i)$ given by (6.4). Thus, total utilization of PNF/G-EDF is equal or better than ECM's if for each τ_i:

$$\begin{aligned}&\left(\sum_{\forall \tau_j \in \gamma_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset}\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\right)\right) + \left\lfloor \frac{\sum_{\forall \tau_j}\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset}}{m}\right\rfloor \\ \leq \quad & \left(\sum_{\forall \tau_j \in \gamma_i^{ex}}\left(2\left\lceil \frac{T_i}{T_j}\right\rceil \sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset}\right)\right) + \left(\sum_{\forall \tau_j \in \zeta_i}\left\lfloor \frac{T_i}{T_j}\right\rfloor\right) \end{aligned} \tag{6.6}$$

$\because\ \gamma_i \subseteq \gamma_i^{ex}$, $\Theta_i \subseteq \Theta_i^{ex}$ and $2\left\lceil \frac{T_i}{T_j}\right\rceil \geq \left\lceil \frac{T_i}{T_j}\right\rceil + 1$, $\therefore\ \sum_{\forall \tau_j \in \gamma_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset}\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\right) \leq \sum_{\forall \tau_j \in \gamma_i^{ex}}\left(2\left\lceil \frac{T_i}{T_j}\right\rceil \sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset}\right)$. So, (6.6) holds if $\left\lfloor \frac{\sum_{\forall \tau_j}\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset}}{m}\right\rfloor \leq \sum_{\forall \tau_j \in \zeta_i}\left\lfloor \frac{T_i}{T_j}\right\rfloor$.

$\sum_{\forall \tau_j}\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset}$ is the total number of transactions in any task $\tau_j \neq \tau_i$ that has no direct conflict with any transaction in τ_i. $\sum_{\forall \tau_j \in \zeta_i}\left\lfloor \frac{T_i}{T_j}\right\rfloor$ is the maximum number of jobs- with higher priority than current job of τ_i - that can be released during T_i. Thus, Claim follows. □

6.5.2 PNF versus RCM

Claim 56. *Following notions in Section 4.3, total utilization of PNF/G-RMA is equal or better than RCM's if for each task τ_i total number of transactions in tasks with lower priority than p_i does not exceed one half of maximum number of jobs with higher priority than p_i that can be released during T_i.*

Proof. Proof follows from proof of Claim 42 with the following modification: Under PNF, c_i is inflated with $RC^{to}_{PNF/G-RMA}(T_i)$ given by (6.1) and $D_i(T_i)$ given by (6.2). Thus, total utilization of PNF/G-RMA is equal or better than RCM's if for each τ_i:

$$\begin{aligned} &\left(\sum_{\forall \tau_j \in \gamma_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset}\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\right)\right) + \left\lfloor \frac{2\sum_{\forall \tau_j, p_j < p_i}\left(\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset}\right)}{m} \right\rfloor \\ \leq\ &\left(2\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j > p_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset}\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\right)\right) + \left(\sum_{\forall \tau_j, p_j > p_i}\left\lceil \frac{T_i}{T_j}\right\rceil\right) \end{aligned} \tag{6.7}$$

$$\begin{aligned} \therefore\ &\left(\sum_{\forall \tau_j \in \gamma_i, p_j > p_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset}\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\right)\right) + \left(2\sum_{\forall \tau_j \in \gamma_i, p_j < p_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset}\right)\right) \\ +\ &\left\lfloor \frac{2\sum_{\forall \tau_j, p_j < p_i}\left(\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset}\right)}{m} \right\rfloor \\ \leq\ &\left(2\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j > p_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset}\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\right)\right) + \left(\sum_{\forall \tau_j, p_j > p_i}\left\lceil \frac{T_i}{T_j}\right\rceil\right) \end{aligned} \tag{6.8}$$

Eq(6.8) holds if

$$\begin{aligned} \therefore\ &\left(\sum_{\forall \tau_j \in \gamma_i, p_j > p_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset}\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\right)\right) + \left(2\sum_{\forall \tau_j, p_j < p_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset}\right)\right) \\ +\ &\left(2\sum_{\forall \tau_j, p_j < p_i}\left(\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset}\right)\right) \\ \leq\ &\left(2\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j > p_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset}\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\right)\right) + \left(\sum_{\forall \tau_j, p_j > p_i}\left\lceil \frac{T_i}{T_j}\right\rceil\right) \end{aligned} \tag{6.9}$$

$$\begin{aligned} \therefore\ &\left(\sum_{\forall \tau_j \in \gamma_i, p_j > p_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset}\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\right)\right) + \left(2\sum_{\forall \tau_j, p_j < p_i}\left(\sum_{\forall s_j^l}\right)\right) \\ \leq\ &\left(2\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j > p_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset}\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\right)\right) + \left(\sum_{\forall \tau_j, p_j > p_i}\left\lceil \frac{T_i}{T_j}\right\rceil\right) \end{aligned} \tag{6.10}$$

$\because\ \gamma_i \subseteq \gamma_i^{ex}$ and $\Theta_i \subseteq \Theta_i^{ex}$, $\therefore\ \sum_{\forall \tau_j \in \gamma_i, p_j > p_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset}\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\right)$ is always less than $2\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j > p_i}\left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset}\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\right)$. Thus, (6.10) holds if $\sum_{\forall \tau_j, p_j < p_i}\left(\sum_{\forall s_j^l}\right)$ does not exceed one half of $\sum_{\forall \tau_j, p_j > p_i}\left\lceil \frac{T_i}{T_j}\right\rceil$. $\sum_{\forall \tau_j, p_j < p_i}\left(\sum_{\forall s_j^l}\right)$ is total number of transactions in tasks with lower priority than τ_i. $\sum_{\forall \tau_j, p_j > p_i}\left\lceil \frac{T_i}{T_j}\right\rceil$ is maximum number of jobs with higher priority than p_i that can be released during T_i. Claim follows. □

6.5.3 PNF versus LCM/G-EDF

Claim 57. *Following notions in Section 4.3, PNF/G-EDF's total utilization is equal or better than LCM/G-EDF's if for each task τ_i:*

- *Maximum number of jobs of $\tau_j \in \gamma_i$ - with higher priority than current job of τ_i - that can exist during T_i is not less than $1/\alpha_{max}$.*
- *Total number of transactions in any task $\tau_j \neq \tau_i$ - that has no direct conflict with any transaction in τ_i - divided by number of processors is not greater than maximum number of jobs- with higher priority than current job of τ_i - that can be released during T_i.*

Proof. Proof follows from proof of Claim 55 where $RC^{to}_{LCM/G-EDF}(T_i)$ is upper bounded by (5.11). Total utilization of PNF/G-EDF is equal or better than total utilization of LCM/G-EDF if for each τ_i

$$\begin{aligned}&\left(\sum_{\forall\tau_j\in\gamma_i}\left(\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i\neq\emptyset}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\right)\right)+\left\lfloor\frac{\sum_{\forall\tau_j}\sum_{\forall s_j^h,\Theta_j^h\cap\Theta_i=\emptyset}}{m}\right\rfloor\\ \leq\ &\left((1+\alpha_{max})\sum_{\forall\tau_j\in\gamma_i^{ex}}\left(\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i^{ex}\neq\emptyset}\left\lceil\frac{T_i}{T_j}\right\rceil\right)\right)+\left(\sum_{\forall\tau_j\in\zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor\right)\end{aligned}\tag{6.11}$$

$\because\ \gamma_i\subseteq\gamma_i^{ex}$ and $\Theta_i\subseteq\Theta_i^{ex}$. $\therefore$ (6.11) holds if

$$\begin{aligned}&\left(\sum_{\forall\tau_j\in\gamma_i}\left(\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i\neq\emptyset}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\right)\right)+\left\lfloor\frac{\sum_{\forall\tau_j}\sum_{\forall s_j^h,\Theta_j^h\cap\Theta_i=\emptyset}}{m}\right\rfloor\\ \leq\ &\left((1+\alpha_{max})\sum_{\forall\tau_j\in\gamma_i}\left(\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i\neq\emptyset}\left\lceil\frac{T_i}{T_j}\right\rceil\right)\right)+\left(\sum_{\forall\tau_j\in\zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor\right)\end{aligned}\tag{6.12}$$

Eq(6.12) holds if:

1. For each τ_i and $\tau_j\in\gamma_i$

$$\sum_{\forall\tau_j\in\gamma_i}\left(\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i\neq\emptyset}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\right)\leq(1+\alpha_{max})\sum_{\forall\tau_j\in\gamma_i}\left(\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i\neq\emptyset}\left\lceil\frac{T_i}{T_j}\right\rceil\right)$$

$$\therefore\ \forall\tau_j\in\gamma_i,\ \left\lceil\frac{T_i}{T_j}\right\rceil+1\leq(1+\alpha_{max})\left\lceil\frac{T_i}{T_j}\right\rceil$$

$$\therefore\ \forall\tau_j\in\gamma_i,\ \frac{1}{\alpha_{max}}\leq\left\lceil\frac{T_i}{T_j}\right\rceil$$

By (4.2), $\left\lceil\frac{T_i}{T_j}\right\rceil$ is maximum number of jobs of τ_j - with higher priority than current job of τ_i - that can exist during T_i.

2. $\left\lfloor \frac{\sum_{\forall \tau_j} \sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset}}{m} \right\rfloor \leq \sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor$. $\sum_{\forall \tau_j} \sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset}$ is total number of transactions in any task $\tau_j \neq \tau_i$ that has no direct conflict with any transaction in τ_i. $\sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor$ is the maximum number of jobs- with higher priority than current job of τ_i - that can be released during T_i.

From the previous observations, Claim follows. □

6.5.4 PNF versus LCM/G-RMA

Claim 58. *Following notions in Section 4.3, total utilization of PNF/G-RMA is equal or better than LCM/G-RMA's if:*

- α_{min} *is small (i.e.,* $\alpha_{min \to 0}$*).*
- *For each task* τ_i*, total number of transactions in tasks with lower priority than* p_i *and have no direct conflict with any transaction in* τ_i *divided by number of processors does not exceed one half of maximum number of jobs with higher priority than* p_i *that can be released during* T_i*.*

Proof. Proof follows from proof of Claim 56 where $RC^{to}_{LCM/G-RMA}(T_i)$ is upper bounded by (5.18). Total utilization of PNF/G-RMA is equal or better than total utilization of LCM/G-RMA if for each τ_i:

$$\begin{aligned}
&\left(\sum_{\forall \tau_j \in \gamma_i} \left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right)\right)\right) + \left\lfloor \frac{\sum_{\forall \tau_j, p_j < p_i} \left(\left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset} \right)}{m} \right\rfloor \\
\leq \quad & (1 + \alpha_{max}) \left(\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j > p_i} \left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right)\right)\right) \\
+ \quad & (1 - \alpha_{min}) \left(\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j < p_i} \left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right)\right)\right) \\
+ \quad & \left(\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil \right)
\end{aligned} \tag{6.13}$$

$$\begin{aligned}
\therefore \quad & \left(\sum_{\forall \tau_j \in \gamma_i, p_j > p_i} \left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right)\right)\right) + \left(2 \sum_{\forall \tau_j \in \gamma_i, p_j < p_i} \left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset}\right)\right) \\
+ \quad & \left\lfloor \frac{2 \sum_{\forall \tau_j, p_j < p_i} \left(\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset}\right)}{m} \right\rfloor \\
\leq \quad & (1 + \alpha_{max}) \left(\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j > p_i} \left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right)\right)\right) \\
+ \quad & (1 - \alpha_{min}) \left(2 \sum_{\forall \tau_j \in \gamma_i^{ex}, p_j < p_i} \left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset}\right)\right) \\
+ \quad & \left(\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil \right)
\end{aligned} \tag{6.14}$$

$\because$ $\gamma_i \subseteq \gamma_i^{ex}$, $\Theta_i \subseteq \Theta_i^{ex}$ and $\alpha_{max} \geq 0$, then $\sum_{\forall \tau_j \in \gamma_i, p_j > p_i} \left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \right)$ is never bigger than $(1+\alpha_{max}) \left(\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j > p_i} \left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \right) \right)$. Thus, (6.14) holds if:

1. For each τ_i

$$\sum_{\forall \tau_j \in \gamma_i, p_j < p_i} \left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset} \right) \leq (1 - \alpha_{min}) \left(\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j < p_i} \left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i^{ex} \neq \emptyset} \right) \right) \tag{6.15}$$

   Eq(6.15) holds if $\alpha_{min} \to 0$.

2. For each τ_i

$$\left\lfloor \frac{2 \sum_{\forall \tau_j, p_j < p_i} \left(\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset} \right)}{m} \right\rfloor \leq \sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil \tag{6.16}$$

   Eq(6.15) holds if

$$\frac{\sum_{\forall \tau_j, p_j < p_i} \left(\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset} \right)}{m} \leq \frac{\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil}{2}$$

   $\sum_{\forall \tau_j, p_j < p_i} \left(\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset} \right)$ is total number of transactions in tasks with lower priority than p_i that do not have direct conflict with any transaction in τ_i. $\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil$ is maximum number of jobs with higher priority than p_i that can be released during T_i.

From previous observations, Claim follows. □

6.5.5 PNF versus Lock-free Synchronization

As mentioned in Section 4.4, the retry-loop lock-free approach in [49] is the most relevant to our work. As lock-free instructions access only one object, then Θ_i^k for any s_i^k will be restricted to one object only (i.e., $\Theta_i^k = \theta_i^k$). Thus, transitive retry cannot happen, $\Theta_i^{ex} = \Theta_i$ and $\gamma_i^{ex} = \gamma_i$.

Claim 59. *Following notions in Section 4.3. If, for each task τ_i, maximum number of jobs-with higher priority than current job of τ_i - that can be released during T_i is not less than total number of transactions in any task $\tau_j \neq \tau_i$ that has no direct conflict with any transaction in τ_i, then total utilization of PNF under G-EDF is equal or better than that of retry-loop lock-free [49] with $s_{max}/r_{max} \geq 1$. s_{max} is the length of longest transaction among all tasks. r_{max} is the maximum execution cost of a single iteration of any lock-free retry loop of any task.*

Proof. Following the same steps of proof Claim 35, total utilization of PNF is equal or better than total utilization of retry-loop lock-free under G-EDF if for each task τ_i

$$\begin{aligned}&\left(\left(\sum_{\forall\tau_j\in\gamma_i}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i\neq\emptyset}\right)\right)+\left\lfloor\frac{\sum_{\forall\tau_j}\sum_{\forall s_j^h,\Theta_j^h\cap\Theta_i=\emptyset}}{m}\right\rfloor\right)s_{max}\\ \leq\ &\left(\left(\sum_{\forall\tau_j\in\gamma_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\beta_{ij}\right)+\left(\sum_{\forall\tau_j\in\zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor\right)\right)r_{max}\end{aligned}\tag{6.17}$$

Let $\beta_{ij}^*=\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i\neq\emptyset}$. Thus, (6.17) becomes

$$\frac{s_{max}}{r_{max}}\leq\frac{\left(\left(\sum_{\forall\tau_j\in\gamma_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\beta_{ij}\right)+\left(\sum_{\forall\tau_j\in\zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor\right)\right)}{\left(\left(\sum_{\forall\tau_j\in\gamma_i}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\beta_{ij}^*\right)\right)+\left\lfloor\frac{\sum_{\forall\tau_j}\sum_{\forall s_j^h,\Theta_j^h\cap\Theta_i=\emptyset}}{m}\right\rfloor\right)}\tag{6.18}$$

$\because\ \beta_{ij}\geq\beta_{ij}^*$, then (6.18) holds if $\sum_{\forall\tau_j\in\zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor\geq\left\lfloor\frac{\sum_{\forall\tau_j}\sum_{\forall s_j^h,\Theta_j^h\cap\Theta_i=\emptyset}}{m}\right\rfloor$. $\sum_{\forall\tau_j\in\zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor$ is maximum number of jobs- with higher priority than current job of τ_i - that can be released during T_i. $\sum_{\forall\tau_j}\sum_{\forall s_j^h,\Theta_j^h\cap\Theta_i=\emptyset}$ is total number of transactions in any task $\tau_j\neq\tau_i$ that has no direct conflict with any transaction in τ_i. $\sum_{\forall\tau_j\in\zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor\geq\left\lfloor\frac{\sum_{\forall\tau_j}\sum_{\forall s_j^h,\Theta_j^h\cap\Theta_i=\emptyset}}{m}\right\rfloor$ allows $s_{max}/r_{max}\geq 1$ with equal or better total utilization for PNF than retry-loop lock-free with G-EDF. □

Claim 60. *Following notions in Section 4.3. If, for each task τ_i, maximum number of jobs with higher priority than p_i that can be released during T_i is not less than double of total number of transactions in tasks with lower priority than p_i that have no direct conflict with any transaction in τ_i divided by number of processors, then total utilization of PNF is equal or better than that of retry-loop lock-free [49] under G-RMA with $s_{max}/r_{max}\geq 1$. s_{max} is the length of longest transaction among all tasks. r_{max} is the maximum execution cost of a single iteration of any lock-free retry loop of any task.*

Proof. Following the same steps of proof Claim 43, total utilization of PNF is equal or better than total utilization of retry-loop lock-free under G-RMA if for each task τ_i

$$\begin{aligned}&\left(\left(\sum_{\forall\tau_j\in\gamma_i}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i\neq\emptyset}\right)\right)+\left\lfloor\frac{2\sum_{\forall\tau_j,p_j<p_i}\left(\sum_{\forall s_j^h,\Theta_j^h\cap\Theta_i=\emptyset}\right)}{m}\right\rfloor\right)s_{max}\\ \leq\ &\left(\left(\sum_{\forall\tau_j\in\gamma_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\beta_{ij}\right)+\left(\sum_{\forall\tau_j,p_j>p_i}\left\lceil\frac{T_i}{T_j}\right\rceil\right)\right)r_{max}\end{aligned}\tag{6.19}$$

Let $\beta_{ij}^*=\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i\neq\emptyset}$, then (6.19) becomes

$$\frac{s_{max}}{r_{max}}\leq\frac{\left(\left(\sum_{\forall\tau_j\in\gamma_i}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\beta_{ij}\right)+\left(\sum_{\forall\tau_j,p_j>p_i}\left\lceil\frac{T_i}{T_j}\right\rceil\right)\right)}{\left(\sum_{\forall\tau_j\in\gamma_i}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\beta_{ij}^*\right)\right)+\left\lfloor\frac{2\sum_{\forall\tau_j,p_j<p_i}\left(\sum_{\forall s_j^h,\Theta_j^h\cap\Theta_i=\emptyset}\right)}{m}\right\rfloor}\tag{6.20}$$

$\because\ \beta_{ij} \geq \beta^*_{ij}$, then (6.20) holds if $\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil \geq \frac{2\sum_{\forall \tau_j, p_j < p_i} \left(\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset} \right)}{m}$. $\sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil$ is maximum number of jobs with higher priority than p_i that can be released during T_i. $\sum_{\forall \tau_j, p_j < p_i} \left(\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset} \right)$ is total number of transactions in tasks with lower priority than p_i that have no direct conflict with any transaction in τ_i. □

6.5.6 PNF versus Locking Protocols

Claim 61. *Following the same notations in Sections 4.3 and 4.5.1, total utilization of PNF is equal or better than total utilization of OMLP under G-EDF if*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_{min}\left(2m-1\right)}{\left(n-1\right)N_{max}\left(\Phi_{max}+1+\frac{1}{m}\right)} \tag{6.21}$$

As all tasks have equal periods and equal number of atomic sections, and number of processors exceeds number of tasks, then s_{max} can be at least equal to L_{max} with shcedulability of PNF equal or better than total utilization of OMLP under G-EDF.

Proof. Use (6.1) for $RC_{i_{to}}(T_i)$ and (6.4) for $D_i(T_i)$ under PNF/G-EDF. Following the same steps of proof of Claim 18, Claim follows. □

Claim 62. *Following the same notations in Sections 4.3 and 4.5.1, total utilization of PNF is equal or better than total utilization of OMLP under G-RMA if*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_{min}\left(2m-1\right)}{\left(n-1\right)N_{max}\left(\Phi_{max}+1+\frac{2}{m}\right)} \tag{6.22}$$

As all tasks have equal periods and equal number of atomic sections, and number of processors exceeds number of tasks, then s_{max} can be at least equal to L_{max} with total utilization of PNF equal or better than total utilization of OMLP under G-RMA.

Proof. Use (6.1) for $RC_{i_{to}}(T_i)$ and (6.2) for $D_i(T_i)$ under PNF/G-RMA. Following the same steps of proof of Claim 18, Claim follows. □

Claim 63. *Following the same notations in Sections 4.3 and 4.5.4, total utilization of PNF is equal or better than total utilization of RNLP under G-EDF if*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_{min}\left(2m-1\right)}{\left(n-1\right)N_{max}\left(\Phi_{max}+1+\frac{1}{m}\right)} \tag{6.23}$$

As all tasks have equal periods and equal number of atomic sections, and number of processors exceeds number of tasks, then s_{max} can be at least equal to L_{max} with total utilization of PNF equal or better than total utilization of RNLP under G-EDF.

Proof. Use (6.1) for $RC_{i_{to}}(T_i)$ and (6.4) for $D_i(T_i)$ under PNF/G-EDF. Following the same steps of proof of Claim 20, Claim follows. □

Claim 64. *Following the same notations in Sections 4.3 and 4.5.4, total utilization of PNF is equal or better than total utilization of RNLP under G-RMA if*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_{min}\left(2m-1\right)}{\left(n-1\right)N_{max}\left(\Phi_{max}+1+\frac{2}{m}\right)} \tag{6.24}$$

As all tasks have equal periods and equal number of atomic sections, and number of processors exceeds number of tasks, then s_{max} can be at least equal to L_{max} with total utilization of PNF equal or better than total utilization of RNLP under G-RMA.

Proof. Use (6.1) for $RC_{i_{to}}(T_i)$ and (6.2) for $D_i(T_i)$ under PNF/G-RMA. Following the same steps of proof of Claim 20, Claim follows. □

6.6 Conclusions

Transitive retry increases transactional retry cost under ECM, RCM, and LCM. PNF avoids transitive retry by avoiding transactional preemptions. PNF reduces the priority of aborted transactions to enable other tasks to execute, increasing processor usage. Executing transactions are not preempted due to the release of higher priority jobs. On the negative side of PNF, higher priority jobs can be blocked by executing transactions of lower priority jobs.

PNF/G-EDF's total utilization is equal or better than ECM's if, for each task τ_i, total number of transactions in any task $\tau_j \neq \tau_i$ - that has no direct conflict with any transaction in τ_i - divided by number of processors is not greater than maximum number of higher priority jobs than current job of τ_i that can be released during T_i. Similar condition holds for the total utilization comparison between PNF/G-EDF and LCM/G-EDF, in addition to maintain a lower bound of $1/\alpha_{max}$ over maximum number of higher priority jobs of τ_j that can exist during T_i and have direct conflict with any transaction in τ_i.

Total utilization of PNF/G-RMA is equal or better than RCM's if, for each task τ_i, total number of transactions in tasks with lower priority than p_i does not exceed one half of maximum number of jobs with higher priority than p_i that can be released during T_i. Total utilization of PNF/G-RMA is equal or better than LCM/G-RMA's if $\alpha_{min} \to 0$ and, for each task τ_i, total number of transactions in tasks with lower priority than p_i and have no direct conflict with any transaction in τ_i divided by number of processors does not exceed one half of maximum number of jobs with higher priority than p_i that can be released during T_i.

Total utilization of PNF under G-EDF and G-RMA is equal or better than total utilization of retry-loop lock-free [49] with $s_{max}/r_{max} \geq 1$ if, for each task τ_i, maximum number of higher priority jobs than current job of τ_i - that can be released during T_i - is not less than maximum number of lower priority transactions in any task $\tau_j \neq \tau_i$ that has no direct conflict with any transaction in τ_i.

Total utilization of PNF was compared against real-time locking protocols (i.e., OMLP and RNLP) under G-EDF and G-RMA. As all tasks have equal periods and equal number of atomic sections, and number of processors exceeds number of tasks, then s_{max} can be at least equal to L_{max} with total utilization of PNF equal or better than total utilization of OMLP and RNLP under G-EDF and G-RMA.

Chapter 7

The FBLT Contention Manager

In this chapter, we present a novel contention manager for resolving transactional conflicts, called FBLT [56]. We upper bound transactional retries and task response times under FBLT, when used with the G-EDF and G-RMA schedulers. We formally identify the conditions under which FBLT has better performance than the previous previous CMs, lock-free and locking protocols in terms of total utilization.

The rest of this Chapter is organized as follows: Section 7.1 discusses limitations of previous contention managers and the motivation to FBLT. Section 7.2 give a formal description of FBLT. We upper bound retry cost and response time under FBLT in Section 7.3. Total utilization comparison between FBLT and previous synchronization techniques is given in Section 7.4. We conclude Chapter in Section 7.5.

7.1 Motivation

With multiple objects per transaction, ECM, RCM (Chapter 4) and LCM (Chapter 5) face transitive retry as shown by Claims 1, 11 and 22. PNF (Chapter 6) is designed to avoid transitive retry by concurrently executing at most m non-conflicting transactions together as shown by Claim 46. These executing transactions are non-preemptive. Thus, executing transactions cannot be aborted due to direct or indirect conflict with other transactions. However, with PNF, all objects accessed by each transaction must be known a-priori. Therefore, this is not suitable with dynamic STM implementations [74]. Additionally, PNF is a centralized CM. This implementation increases overhead.

Thus, we propose the *First Bounded, Last Timestamp contention manager* (or FBLT) that achieves the following goals:

1. Reduce the retry cost of each transaction s_i^k due to another transaction s_j^l, just as LCM does compared to ECM and RCM.

2. Avoid or bound the effect of transitive retry, similar to PNF, without prior knowledge of accessed objects by each transaction, enabling dynamic STM.
3. Reduce overhead through decentralized design.

7.2 The FBLT Contention Manager

Algorithm 5 illustrates FBLT. Each transaction s_i^k can be aborted during T_i for at most Ω_i^k times. η_i^k records the number of times s_i^k has already been aborted up to now. If s_i^k and s_j^l have not joined the m_set yet, then they are preemptive transactions. Preemptive transactions resolve conflicts using LCM (Algorithm 3) (step 2). Thus, FBLT defaults to LCM when no transaction reaches its Ω. If only one of the transactions is in the m_set, then the non-preemptive transaction (the one in m_set) aborts the other one (steps 15 to 26). η_i^k is incremented each time s_i^k is aborted as long as $\eta_i^k < \Omega_i^k$ (steps 5 and 18). Otherwise, s_i^k is added to the m_set and its priority is increased to m_prio (steps 7 to 9 and 20 to 22). When the priority of s_i^k is increased to m_prio, s_i^k becomes a non-preemptive transaction. Non-preemptive transactions cannot be aborted by other preemptive transactions, nor by any other real-time job. The m_set can hold at most m concurrent transactions because there are m processors in the system. $r(s_i^k)$ records the time s_i^k joined the m_set (steps 8 and 21). When non-preemptive transactions conflict together (step 27), the transaction that joined m_set first is the one to commit first (steps 29 and 31). Thus, non-preemptive transactions are executed in increasing order of joining the m_set.

7.2.1 Illustrative Example

We now illustrate FBLT's behavior with the following example:

1. Transaction $s_i^k(\theta_1, \theta_2)$ is released while $m\_set = \emptyset$. $\eta_i^k = 0$ and $\Omega_i^k = 3$.
2. Transaction $s_a^b(\theta_2)$ is released while $s_i^k(\theta_1, \theta_2)$ is running. $p(s_a^b) > p(s_i^k)$ and $\eta_i^k < \Omega_i^k$. Applying LCM, $s_i^k(\theta_1, \theta_2)$ is aborted in favor of s_a^b and η_i^k is incremented to 1.
3. $s_a^b(\theta_2)$ commits. $s_i^k(\theta_1, \theta_2)$ runs again. Transaction $s_c^d(\theta_2)$ is released while $s_i^k(\theta_1, \theta_2)$ is running. $p(s_c^d) > p(s_i^k)$. Applying LCM, $s_i^k(\theta_1, \theta_2)$ is aborted again in favor of $s_c^d(\theta_2)$. η_i^k is incremented to 2.
4. $s_c^d(\theta_2)$ commits. $s_e^f(\theta_2, \theta_3)$ is released. $p(s_e^f) > p(s_i^k)$ and $\Omega_e^f = 2$. $s_i^k(\theta_1, \theta_2)$ is aborted in favour of $s_e^f(\theta_2, \theta_3)$ and η_i^k is incremented to 3.
5. $s_j^l(\theta_3)$ is released. $p(s_j^l) > p(s_e^f)$. $s_e^f(\theta_2, \theta_3)$ is aborted in favor of $s_j^l(\theta_3)$ and η_e^f is incremented to 1.
6. $s_i^k(\theta_1, \theta_2)$ and $s_e^f(\theta_2, \theta_3)$ are compared again. $\because \eta_i^k = \Omega_i^k$, $\therefore$ $s_i^k(\theta_1, \theta_2)$ is added to m_set. $m\_set = \left\{s_i^k(\theta_1, \theta_2)\right\}$. $s_i^k(\theta_1, \theta_2)$ becomes a non-preemptive transaction. As $s_e^f(\theta_2, \theta_3)$ is a preemptive transaction, $\therefore$ $s_e^f(\theta_2, \theta_3)$ is aborted in favour of $s_i^k(\theta_1, \theta_2)$, despite $p(s_e^f)$ being greater than the original priority of $s_i^k(\theta_1, \theta_2)$. η_e^f is incremented to 2.

Algorithm 5: FBLT

Data: s_i^k: interfered transaction;
s_j^l: interfering transaction;
Ω_i^k: maximum number of times s_i^k can be aborted during T_i;
η_i^k: number of times s_i^k has already been aborted up to now;
m_set: contains at most m non-preemptive transactions. m is number of processors;
m_prio: priority of any transaction in m_set. *m_prio* is higher than any priority of any real-time task;
$r(s_i^k)$: time point at which s_i^k joined m_set;
Result: atomic sections that will abort

1 **if** s_i^k, $s_j^l \notin m\_set$ **then**
2   Apply LCM (Algorithm 3);
3   **if** s_i^k *is aborted* **then**
4   **if** $\eta_i^k < \Omega_i^k$ **then**
5   Increment η_i^k by 1;
6   **else**
7   Add s_i^k to m_set;
8   Record $r(s_i^k)$;
9   Increase priority of s_i^k to *m_prio*;
10   **end**
11   **else**
12   Swap s_i^k and s_j^l;
13   Go to Step 3;
14   **end**
15 **else if** $s_j^l \in m\_set, s_i^k \notin m\_set$ **then**
16   Abort s_i^k;
17   **if** $\eta_i^k < \Omega_i^k$ **then**
18   Increment η_i^k by 1;
19   **else**
20   Add s_i^k to m_set;
21   Record $r(s_i^k)$;
22   Increase priority of s_i^k to *m_prio*;
23   **end**
24 **else if** $s_i^k \in m\_set, s_j^l \notin m\_set$ **then**
25   Swap s_i^k and s_j^l;
26   Go to Step 15;
27 **else**
28   **if** $r(s_i^k) < r(s_j^l)$ **then**
29   Abort s_j^l;
30   **else**
31   Abort s_i^k;
32   **end**
33 **end**

7. $s_j^l(\theta_3)$ commits but $s_g^h(\theta_3)$ is released. $p(s_g^h) > p(s_e^f)$ but $\eta_e^f = \Omega_e^f$. So, $s_e^f(\theta_2, \theta_3)$ becomes a non-preemptive transaction. $m\_set = \left\{s_i^k(\theta_1, \theta_2), s_e^f(\theta_2, \theta_3)\right\}$.
8. $s_i^k(\theta_1, \theta_2)$ and $s_e^f(\theta_2, \theta_3)$ are now non-preemptive transactions. $s_i^k(\theta_1, \theta_2)$ and $s_e^f(\theta_2, \theta_3)$ still conflict together. So, they are executed according to their addition order to the $m_$set. So, $s_i^k(\theta_1, \theta_2)$ commits first, followed by $s_e^f(\theta_2, \theta_3)$.
9. $s_g^h(\theta_3)$ will continue to abort and retry in favour of $s_e^f(\theta_2, \theta_3)$ until $s_e^f(\theta_2, \theta_3)$ commits or $\eta_g^h = \Omega_g^h$. Even if $s_g^h(\theta_3)$ joined the $m_$set, $s_g^h(\theta_3)$ will still abort and retry in favour of $s_e^f(\theta_2, \theta_3)$, because $s_e^f(\theta_2, \theta_3)$ joined the $m_$set earlier than $s_g^h(\theta_3)$.

It is seen from steps 2 to 6 that $s_i^k(\theta_1, \theta_2)$ can be aborted due to direct conflict with other transactions, or due to transitive retry. Irrespective of the reason for the conflict, once a transaction has reached its Ω, the transaction becomes a non-preemptive one (steps 6 and 7). Non-preemptive transactions have higher priority than other preemptive transactions (steps 6 and 7). Non-preemptive transactions execute in their arrival order to the $m_$set.

7.3 Retry Cost and Response Time Bounds

We now derive an upper bound on the retry cost of any job τ_i^x under FBLT during an interval $L \leq T_i$. Since all tasks are sporadic (i.e., each task τ_i has a minimum period T_i), T_i is the maximum study interval for each task τ_i.

Claim 65. *The total retry cost for any job τ_i^x under FBLT with G-EDF and G-RMA due to 1) conflicts between its transactions and transactions of other jobs during an interval $L \leq T_i$ and 2) release of higher priority jobs during L is upper bounded by:*

$$RC_{i_{to}}(L) \leq \sum_{\forall s_i^k} \left(\Omega_i^k len(s_i^k) + \sum_{\forall s_{iz}^k \in \chi_i^k} len(s_{iz}^k) \right) + RC_{i_{re}}(L) \tag{7.1}$$

where χ_i^k is the set of at most $m-1$ maximum length transactions conflicting directly or indirectly (through transitive retry) with s_i^k. Each transaction $s_{iz}^k \in \chi_i^k$ belongs to a distinct task τ_j. $RC_{i_{re}}(L)$ is the retry cost resulting from the release of higher priority jobs which preempt τ_i^x. $RC_{i_{re}}(L)$ is calculated by (4.11) for G-EDF, and (4.17) for G-RMA schedulers.

Proof. By the definition of FBLT, $s_i^k \in \tau_i^x$ can be aborted a maximum of Ω_i^k times before s_i^k joins the $m_$set. Transactions preceding s_i^k in the $m_$set can conflict directly with s_i^k, or indirectly through transitive retry. The worst case scenario for s_i^k after joining the $m_$set occurs if s_i^k is preceded by $m-1$ maximum length conflicting transactions. Hence, in the worst case, s_i^k has to wait for the previous $m-1$ transactions to commit first. The priority of s_i^k after joining the $m_$set is higher than any real-time job. Therefore, the non-preemptive s_i^k is not aborted due to release of any real-time job with higher priority than original priority of s_i^k. Following proofs of Claims 8 and 13, retry cost of s_i^k - before s_i^k joins $m_$set- due to release of higher priority jobs is calculated by (4.11) for G-EDF, and (4.17) for G-RMA. Transactions of each task execute sequentially. Thus, the non-preemptive s_i^k cannot be preceded in the m_set by two or more transactions of the same task. So, each transaction $s_{iz}^k \in \chi_i^k$ belong to a distinct task. Claim follows. □

Claim 66. *Under FBLT with G-EDF and G-RMA, the blocking time of a job τ_i^x due to lower priority jobs is upper bounded by:*

$$D(\tau_i^x) = \sum_{max_m} \{s_{j_{max}}\}_{\forall \tau_j^l, p_j^l < p_i^x} \tag{7.2}$$

where $s_{j_{max}}$ is the maximum length transaction in any job τ_j^l with original priority lower than p_i^x. The right hand side of (7.2) is the sum of the m maximum transactional lengths in all jobs with lower priority than τ_i^x.

Proof. The worst case blocking time for τ_i^x occurs when the maximum length m transactions in lower priority jobs than τ_i^x are executing non-preemptively. The m non-preemptive transactions execute sequentially if they conflict with each other. τ_i^x is delayed by the sequential execution of non-preemptive transactions if jobs with higher priority than p_i^x are released as soon as one of the non-preemptive transactions commits. No transaction with lower priority than p_i^x can be released while τ_i^x is waiting for a processor. Claim follows. □

Claim 67. *The response time R_i^{up} of any job τ_i^x under FBLT with G-EDF and G-RMA is upper bounded by:*

$$R_i^{up} = c_i + RC_{i_{to}}(R_i^{up}) + D(\tau_i^x) + \left\lfloor \frac{1}{m} \sum_{\forall j \neq i} I_{ij}(R_i^{up}) \right\rfloor \tag{7.3}$$

where $RC_{i_{to}}(R_i^{up})$ is calculated by (7.1), $D(\tau_i^x)$ is calculated by (7.2), and $I_{ij}(R_i^{up})$ is calculated by (4.15) for G-EDF, and (4.4) for G-RMA. c_j of any job $\tau_j^y \neq \tau_i^x$, $p_j^y > p_i^x$ is inflated to c_{ji} as given by (4.14).

Proof. Using Claims 10, 15, 65 and 66, Claim follows. □

7.4 Total utilization Comparison

We now (formally) compare performance of FBLT with G-EDF and G-RMA against ECM (Chapter 4), RCM (Chapter 4), LCM (Chapter 5), PNF (Chapter 6), retry-loop lock-free [49] and locking protocols((i.e., OMLP [22, 29] and RNLP [149]). Such a comparison will reveal when FBLT outperforms others. Toward this, we compare the total utilization under FBLT, with that under the other synchronization methods as outlined in Section 4.3. Total utilization comparison between FBLT and other synchronization techniques is done as in Sections 5.4 and 5.6 with the addition of $D(\tau_i^x)$ - as given by (7.2)- to the inflated execution time of any job τ_i^x under FBLT.

7.4.1 FBLT versus ECM

Claim 68. *Let $\Omega_i^{max} = max\{\Omega_i^k\}_{\forall s_i^k}$ be the maximum abort number for any preemptive transaction s_i^k in τ_i. Following notions in Section 4.3, total utilization of FBLT is equal to or better than ECM's if Ω_i^{max} of any τ_i is not greater than double the difference between ratio of maximum number of transactions in all jobs with higher priority than current job of τ_i and have direct or indirect conflict with transactions in τ_i to total number of transactions in any job of τ_i and number of processors. Formally, total utilization of FBLT is equal or better than ECM's if for each τ_i*

$$\Omega_i^{max} \leq 2\left(\frac{\sum_{\forall \tau_j \in \gamma_i^{ex}} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l, (\Theta = \Theta_j^l \cap \Theta_i^{ex}) \neq \emptyset}\right)}{|s_i|} - m\right) \tag{7.4}$$

Proof. Proof follows from proof of Claim 34 with the following modification: Under FBLT, c_i is inflated with $RC_{FBLT}^{to}(T_i)$ given by (7.1) and $D(\tau_i^x)$ given by (7.2). Thus, total utilization of FBLT is equal or better than ECM's if for each τ_i:

$$m + \sum_{\forall s_i^k} \left(\Omega_i^k + m - 1\right) \leq \sum_{\forall \tau_j \in \gamma_i^{ex}} \left(2 \left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i^{ex}\right) \neq \emptyset}\right) \tag{7.5}$$

$\because$ $|s_i| = \sum_{\forall s_i^k}$, where $|s_i|$ is total number of transactions in any job of τ_i. $\because$ $\Omega_i^{max} \geq \Omega_i^k$, $\therefore$ (7.5) holds if

$$m + |s_i| \left(\Omega_i^{max} + m - 1\right) \leq \sum_{\forall \tau_j \in \gamma_i^{ex}} \left(2 \left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i^{ex}\right) \neq \emptyset}\right) \tag{7.6}$$

$$\therefore \Omega_i^{max} \leq \frac{\left(\sum_{\forall \tau_j \in \gamma_i^{ex}} \left(2 \left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l, (\Theta = \Theta_j^l \cap \Theta_i^{ex}) \neq \emptyset}\right)\right) - (1 + |s_i|)\, m + |s_i|}{|s_i|} \tag{7.7}$$

$\because$ $|s_i| \geq 1$, $\therefore$ $\frac{1+|s_i|}{|s_i|} \leq 2$. Thus, (7.7) holds if

$$\Omega_i^{max} \leq 2\left(\frac{\sum_{\forall \tau_j \in \gamma_i^{ex}} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l, (\Theta = \Theta_j^l \cap \Theta_i^{ex}) \neq \emptyset}\right)}{|s_i|} - m\right) \tag{7.8}$$

$\because$ $\sum_{\forall \tau_j \in \gamma_i^{ex}} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l, (\Theta = \Theta_j^l \cap \Theta_i^{ex}) \neq \emptyset}\right)$ is the maximum number of transactions in all jobs with higher priority than current job of τ_i and have direct or indirect conflict with transactions in τ_i, Claim follows. □

7.4.2 FBLT versus RCM

Claim 69. *Let* $\Omega_i^{max} = max\{\Omega_i^k\}_{\forall s_i^k}$ *be the maximum abort number for any preemptive transaction* s_i^k *in* τ_i*. Following notions in Section 4.3, total utilization of FBLT is equal to or better than RCM's if* Ω_i^{max} *of any* τ_i *is not greater than double the difference between ratio of maximum number of transactions in all jobs with higher priority than* p_i *and have direct or indirect conflict with transactions in* τ_i *to total number of transactions in any job of* τ_i *and number of processors. Formally, total utilization of FBLT is equal to or better than RCM's if for each* τ_i

$$\Omega_i^{max} \leq 2\left(\frac{\sum_{\forall \tau_j \in \gamma_i^{ex}, p_j > p_i}\left(\left(\left\lceil \frac{T_i}{T_j}\right\rceil + 1\right)\sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i^{ex}\right) \neq \emptyset}\right)}{|s_i|} - m\right) \tag{7.9}$$

Proof. Proof is similar to proof of Claim 68 except that $RC_{RCM}^{to}(T_i)$ is given by (5.19). □

7.4.3 FBLT versus LCM/G-EDF

Claim 70. *Let* $\Omega_i^{max} = max\{\Omega_i^k\}_{\forall s_i^k}$ *be the maximum abort number for any preemptive transaction* s_i^k *in* τ_i*. Following notions in Section 4.3, total utilization of FBLT is equal to or better than LCM/G-EDF's if double number of processors subtracted from ratio of* $1 + \alpha_{max}$ *multiplied by maximum number of transactions in all jobs with higher priority than current job of* τ_i *and have direct or indirect conflict with transactions in* τ_i *to total number of transactions in any job of* τ_i *is not less than* Ω_i^{max} *of any* τ_i*. Formally, total utilization of FBLT is equal to or better than LCM/G-EDF's if for each* τ_i

$$\Omega_i^{max} \leq \frac{(1 + \alpha_{max})\sum_{\forall \tau_j \in \gamma_i^{ex}}\left(\left\lceil \frac{T_i}{T_j}\right\rceil \sum_{\forall s_j^l, \left(\Theta = \Theta_j^l \cap \Theta_i^{ex}\right) \neq \emptyset}\right)}{|s_i|} - 2m \tag{7.10}$$

Proof. Proof is similar to proof of Claim 68 except that $RC_{LCM/G-EDF}^{to}(T_i)$ is given by (5.11). □

7.4.4 FBLT versus G-RMA/LCM

Claim 71. *Let* $\Omega_i^{max} = max\{\Omega_i^k\}_{\forall s_i^k}$ *be the maximum abort number for any preemptive transaction* s_i^k *in* τ_i*. Following notions in Section 4.3, total utilization of FBLT is equal to or better than G-RMA/LCM's if double number of processors subtracted from ratio of sum of:*

- *product of* $1 + \alpha_{max}$ *by maximum number of transactions in all jobs with higher priority than current job of* τ_i *and have direct or indirect conflict with transactions in* τ_i*.*

- *product of $1-\alpha_{min}$ by maximum number of transactions in all jobs with lower priority than current job of τ_i and have direct or indirect conflict with transactions in τ_i*

to total number of transactions in any job of τ_i is not less than Ω_i^k of any τ_i. Formally, total utilization of FBLT is equal to or better than G-RMA/LCM's if for each τ_i

$$\begin{aligned}\Omega_i^{max} & \\ \leq & \frac{\left((1+\alpha_{max})\sum_{\forall\tau_j\in\gamma_i^{ex},p_j>p_i}\left(\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i^{ex}\neq\emptyset}\right)\right)}{|s_i|} \\ + & \frac{\left(2(1-\alpha_{min})\sum_{\forall\tau_j\in\gamma_i^{ex},p_j<p_i}\left(\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i^{ex}\neq\emptyset}\right)\right)}{|s_i|}-2m\end{aligned} \tag{7.11}$$

Proof. Proof is similar to proof of Claim 68 except that $RC^{to}_{G-RMA/LCM}(T_i)$ is given by (5.18). □

7.4.5 FBLT versus PNF/G-EDF

Claim 72. *Let $\Omega_i^{max}=max\{\Omega_i^k\}_{\forall s_i^k}$ be the maximum abort number for any preemptive transaction s_i^k in τ_i. Following notions in Section 4.3, total utilization of FBLT is equal or better than PNF/G-EDF's if sum of double number of processors and maximum number of higher priority jobs- other than current job of τ_i - that can be released during T_i subtracted from ratio of sum of:*

- *Maximum number of transactions- in any job of $\tau_j\neq\tau_i$ that can exist during T_i- that have direct conflict with any transaction in τ_i.*
- *Floor of total number of transactions in any task $\tau_j\neq\tau_i$ - that has no direct conflict with any transaction in τ_i - divided by number of processors*

to total number of transactions in any job of τ_i is not less than Ω_i^{max} of any τ_i. Formally, total utilization of FBLT is equal to or better than PNF/G-EDF's if for each τ_i

$$\Omega_i^{max}\leq\frac{\sum_{\forall\tau_j\in\gamma_i}\left(\sum_{\forall s_j^l,\Theta_j^l\cap\Theta_i\neq\emptyset}\left(\left\lceil\frac{T_i}{T_j}\right\rceil+1\right)\right)+\left\lfloor\frac{\sum_{\forall\tau_j}\sum_{\forall s_j^h,\Theta_j^h\cap\Theta_i=\emptyset}}{m}\right\rfloor}{|s_i|}-\sum_{\forall\tau_j\in\zeta_i}\left\lfloor\frac{T_i}{T_j}\right\rfloor-2m \tag{7.12}$$

Proof. Proof is similar to proof of Claim 68 except that $RC^{to}_{PNF/G-EDF}(T_i)$ is given by (6.1) and $D_i(T_i)$ given by (6.4). □

7.4.6 FBLT versus PNF/G-RMA

Claim 73. *Let $\Omega_i^{max}=max\{\Omega_i^k\}_{\forall s_i^k}$ be the maximum abort number for any preemptive transaction s_i^k in τ_i. Following notions in Section 4.3, total utilization of FBLT is equal or* *better than PNF/G-RMA's if sum of double number of processors and maximum number of higher priority jobs- other than current job of τ_i - that can be released during T_i subtracted from ratio of sum of:*

- *Maximum number of transactions- in any job of $\tau_j \neq \tau_i$ that can exist during T_i- that have direct conflict with any transaction in τ_i.*
- *Floor of double of total number of transactions in any task τ_j with lower priority than p_i - that has no direct conflict with any transaction in τ_i - divided by number of processors*

to total number of transactions in any job of τ_i is not less than Ω_i^k of any τ_i. Formally, total utilization of FBLT is equal to or better than PNF/G-RMA's if for each τ_i

$$\Omega_i^{max} \leq \frac{\sum_{\forall \tau_j \in \gamma_i} \left(\sum_{\forall s_j^l, \Theta_j^l \cap \Theta_i \neq \emptyset} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \right) + \left\lfloor \frac{2 \sum_{\forall \tau_j, p_j < p_i} \left(\sum_{\forall s_j^h, \Theta_j^h \cap \Theta_i = \emptyset} \right)}{m} \right\rfloor}{|s_i|} - 2m - \sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil \tag{7.13}$$

Proof. Proof is similar to proof of Claim 68 except that $RC^{to}_{PNF/G-RMA}(T_i)$ is given by (6.1) and $D_i(T_i)$ given by (6.2). □

7.4.7 FBLT versus Lock-free

Claim 74. *Let $\Omega_i^{max} = max\{\Omega_i^k\}_{\forall s_i^k}$ be the maximum abort number for any preemptive transaction s_i^k in τ_i. Following notions in Section 4.3, total utilization of FBLT under G-EDF is equal or better than total utilization of lock-free if for each task τ_i*

$$\frac{s_{max}}{r_{max}} \leq \frac{\sum_{\forall \tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \beta_{ij} + \sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor}{m + |s_i| \left(\Omega_i^{max} + m - 1 \right) + \sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor}$$

Proof. Using Claim 65 and following the same steps of proof of Claim 16, total utilization of FBLT is equal or better than that of retry-loop lock-free under G-EDF if for each task τ_i

$$\begin{aligned} & \left(m + \sum_{\forall s_i^k} \left(\Omega_i^k + m - 1 \right) + \sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor \right) s_{max} \\ \leq & \left(\sum_{\forall \tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \beta_{ij} + \sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor \right) r_{max} \end{aligned} \tag{7.14}$$

$$\frac{s_{max}}{r_{max}} \leq \frac{\sum_{\forall \tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \beta_{ij} + \sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor}{m + \sum_{\forall s_i^k} \left(\Omega_i^k + m - 1 \right) + \sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor} \tag{7.15}$$

$\because |s_i| = \sum_{\forall s_i^k}$, where $|s_i|$ is total number of transactions in any job of τ_i. $\because \Omega_i^{max} \geq \Omega_i^k$, $\therefore$ (7.15) holds if

$$\frac{s_{max}}{r_{max}} \leq \frac{\sum_{\forall \tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \beta_{ij} + \sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor}{m + |s_i| \left(\Omega_i^{max} + m - 1 \right) + \sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor} \tag{7.16}$$

Claim follows. □

Claim 75. *Let $\Omega_i^{max} = max\{\Omega_i^k\}_{\forall s_i^k}$ be the maximum abort number for any preemptive transaction s_i^k in τ_i. Following notions in Section 4.3, total utilization of FBLT under G-RMA is equal or better than that of lock-free if for each task τ_i*

$$\frac{s_{max}}{r_{max}} \leq \frac{\sum_{\forall \tau_j \in \gamma_i} \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right) \beta_{ij} + \sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil}{m + |s_i| \left(\Omega_i^{max} + m - 1 \right) + \sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil}$$

Proof. Proof is the same as proof of Claim 74 except that $RC_{i_{re}}$ under FBLT is given by (4.17) and LRC_{to} under lock-free is given by (4.31). □

7.4.8 FBLT versus Locking Protocols

Claim 76. *Following the same notations in Sections 4.3 and 4.5.1, total utilization of FBLT is equal or better than total utilization of OMLP under G-EDF if for each τ_i*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_i \left(2m - 1 \right)}{m + N_i \left(\Omega_i^{max} + m - 1 \right) + \sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor}$$

Let $\Omega_{max} = max\{\Omega_i^{max}\}_{\forall \tau_i}$. As number of atomic sections in each task increases, all tasks have equal number of atomic sections, and number of processors is not less than Ω_{max}, then s_{max} can be at least equal to L_{max} with total utilization of FBLT equal or better than total utilization of OMLP under G-EDF. In any case, s_{max} should not exceed $2.L_{max}$.

Proof. Use (7.1) for $RC_{i_{to}}(T_i)$ and (7.2) for $D_i(T_i)$ under G-EDF/FBLT. $|s_i| = N_i$ by definition of $|s_i|$ and N_i. Following the same steps of proof of Claim 18, Claim follows. □

Claim 77. *Following the same notations in Sections 4.3 and 4.5.1, total utilization of FBLT is equal or better than total utilization of OMLP under G-RMA if for each τ_i*

$$\frac{s_{max}}{L_{max}} \leq \frac{N_i \left(2m - 1 \right)}{m + N_i \left(\Omega_i^{max} + m - 1 \right) + \sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil}$$

Let $\Omega_{max} = max\{\Omega_i^{max}\}_{\forall \tau_i}$. As number of atomic sections in each task increases, all tasks have equal number of atomic sections, and number of processors is not less than Ω_{max}, then s_{max} can be at least equal to L_{max} with total utilization of FBLT equal or better than total utilization of OMLP under G-RMA. In any case, s_{max} should not exceed $2.L_{max}$.

Proof. Proof is the same as proof of Claim 76. □

Claim 78. *Following the same notations in Sections 4.3 and 4.5.4, total utilization of FBLT is equal or better than total utilization of RNLP under G-EDF if for each* τ_i

$$\frac{s_{max}}{L_{max}} \leq \frac{N_i\,(2m-1)}{m + N_i\,(\Omega_i^{max} + m - 1) + \sum_{\forall \tau_j \in \zeta_i} \left\lfloor \frac{T_i}{T_j} \right\rfloor}$$

Let $\Omega_{max} = max\{\Omega_i^{max}\}_{\forall \tau_i}$. *As number of atomic sections in each task increases, all tasks have equal number of atomic sections, and number of processors is not less than* Ω_{max}*, then* s_{max} *can be at least equal to* L_{max} *with total utilization of FBLT equal or better than that of RNLP under G-EDF. In any case,* s_{max} *should not exceed* $2.L_{max}$.

Proof. Use (7.1) for $RC_{i_{to}}(T_i)$ and (7.2) for $D_i(T_i)$ under G-EDF/FBLT. Following the same steps of proof of Claim 20, Claim follows. □

Claim 79. *Following the same notations in Sections 4.3 and 4.5.4, total utilization of FBLT is equal or better than total utilization of RNLP under G-RMA if for each* τ_i

$$\frac{s_{max}}{L_{max}} \leq \frac{N_i\,(2m-1)}{m + N_i\,(\Omega_i^{max} + m - 1) + \sum_{\forall \tau_j, p_j > p_i} \left\lceil \frac{T_i}{T_j} \right\rceil}$$

Let $\Omega_{max} = max\{\Omega_i^{max}\}_{\forall \tau_i}$. *As number of atomic sections in each task increases, all tasks have equal number of atomic sections, and number of processors is not less than* Ω_{max}*, then* s_{max} *can be at least equal to* L_{max} *with total utilization of FBLT equal or better than that of RNLP under G-RMA. In any case,* s_{max} *should not exceed* $2.L_{max}$.

Proof. Proof is the same as proof of Claim 78. □

7.5 Conclusions

Transitive retry increases transactional retry costs under ECM, RCM, and LCM. PNF avoids transitive retry by avoiding transactional preemptions. PNF avoids transitive retry cost by concurrently executing non-conflicting transactions, which are non-preemptive. However, PNF requires a-priori knowledge about objects accessed by each transaction. Besides, PNF is a centralized CM. This is incompatible with dynamic STM implementations. Thus, we introduce the FBLT contention manager. Under FBLT, each transaction is allowed to abort for no larger than a specified number of times. Afterwards, the transaction becomes non-preemptive. Non-preemptive transactions have higher priorities than other preemptive transactions and real-time jobs. Non-preemptive transactions resolve their conflicts according to the order they become non-preemptive (i.e., FBLT aborts the later non-preemptive transaction in favour of the earlier non-preemptive transaction).

By proper adjustment of the maximum abort number for any preemptive transaction of any task τ_i (i.e., Ω_i^{max}), FBLT's total utilization is equal to or better than total utilization of other CMs. Ratio between s_{max} for FBLT on one side and r_{max} for lock-free and L_{max} for locking protocols on the other side also depends on Ω_i^{max}. As Ω_i^{max} decreases, s_{max}/r_{max} and s_{max}/L_{max} increase. As number of atomic sections in each task increases, all tasks have equal number of atomic sections, and number of processors is not less than max Ω_i^{max} for any τ_i, then s_{max} can be at least equal to L_{max} with total utilization of FBLT equal or better than total utilization of OMLP and RNLP under G-EDF and G-RMA. In any case, s_{max} should not exceed $2.L_{max}$.

Chapter 8

FBLT Contention Manager with Checkpointing

In this chapter, we consider checkpointing [91] with software transactional memory (STM) concurrency control for embedded multicore real-time software, and present a modified version of FBLT contention manager called *Checkpointing FBLT* (CP-FBLT). We upper bound transactional retries and task response times under CP-FBLT, and identify when CP-FBLT is a more appropriate alternative to FBLT without checkpointing.

The rest of this Chapter is organized as follows: We present the motivation for introducing checkpointing into FBLT in Section 8.1. We introduce checkpointing FBLT (CP-FBLT) that combines original FBLT with checkpointing in Section 8.2. We establish CP-FBLT's retry and response time upper bounds under G-EDF and G-RMA schedulers in Section 8.3. We also identify the conditions under which CP-FBLT is a better alternative to non-checkpointing FBLT in Section 8.4. We conclude Chapter in Section 8.5.

8.1 Motivation

[91] introduces checkpointing as an alternative to closed nesting transactions [142]. [91] uses boosted transactions [75] instead of closed nesting [88, 117, 142] to implement checkpointing. Upon a conflict, a transaction does not need to revert to its beginning, but rather to a point where the conflict can be avoided. Thus, checkpointing enables partial abort. [143] applies checkpointing in distributed transactional memory using Hyflow [123]. Checkpointing is used for fault tolerance in real-time systems [2, 3, 93, 94, 102, 105].

Under checkpointing [91], a transaction $s_i^k \in \tau_i$ does not need to restart from the beginning upon a conflict on object θ. s_i^k just needs to return to the first point it accessed θ. If s_i^k needs to restart from its beginning, then the time between the beginning of s_i^k and the first access to θ is wasted. Besides, restarting s_i^k from its beginning increases the chances of aborting s_i^k by other transactions. Thus, response time of τ_i can be improved by checkpointing unless s_i^k acquires all its objects at its beginning. While previous CMs (i.e., ECM, RCM (Chapter 4), LCM (Chapter 5), PNF (Chapter 6) and FBLT (Chapter 7)) without checkpointing try to resolve conflicts using proper strategies, checkpointing enhances performance by reducing aborted part of each transaction. Thus, checkpointing acts as a complementary component to different CMs to further enhance response time. Checkpointing integrated into CMs allows programmers to reap STM's significant programmability and composability benefits for multicore embedded real-time software.

Behaviour of some CMs, like PNF (Chapter 6), can make checkpointing useless. PNF requires a priori knowledge of accessed objects within transactions. Only the first m non-conflicting transactions are allowed to execute concurrently and non-preemptively. Thus, PNF makes no use of checkpointing because there is no conflict between non-preemptive transactions.

Other CMs (e. g., FBLT (Chapter 7)) allow conflicting transaction to run concurrently. So, FBLT can benefit from checkpointing. FBLT, by definition, depends on LCM. LCM, in turn, depends on ECM for G-EDF and RCM for G-RMA. Like PNF, FBLT allows any transaction s_i^k to be a non-preemptive transaction if s_i^k has been aborted for a specified number of times Ω_i^k. Non-preemptive transactions cannot be aborted by preemptive transactions, nor by non-critical sections in real-time jobs. FBLT, unlike PNF, allow non-preemptive transactions to abort each other. Non-preemptive transactions resolve conflicts using time of being a non-preemptive transaction. As FBLT tries to combine advantages of other CMs, we extend FBLT to checpointing FBLT (CP-FBLT) to improve response time over original FBLT.

8.2 Checkpointing FBLT (CP-FBLT)

CP-FBLT depends on LCM (Chapter 5) with checkpointing. So, we initially illustrate LCM integrated with checkpointing (Section 8.2.1). Afterwards, we illustrate FBLT with checkpointing in (Section 8.2.2).

8.2.1 Checkpointing LCM (CPLCM)

Algorithm 6 presents LCM integrated with checkpointing to give CPLCM. A new checkpoint is recorded for each newly accessed object θ by any transaction s_h^u (step 2). Checkpoint is recorded when θ is accessed for the first time because any further changes to θ will be discarded upon conflict. CPLCM uses priorities of s_i^k and s_j^l, the remaining length of s_i^k when it is interfered, as well as $len(s_j^l)$, to decide which transaction must be aborted. If s_j^l starts after s_i^k and $p_i^k > p_j^l$, then s_j^l would be the transaction to abort (step 6). Otherwise, c_{ij}^{kl}, α_{ij}^{kl} and α are calculated (steps 9, 10 and 11) to determine whether it is worth aborting s_i^k in favour of s_j^l. If $len(s_j^l)$ is relatively small compared to $len(s_i^k)$, then c_{ij}^{kl} approaches

Algorithm 6: CPLCM

Data: s_i^k and s_j^l are two conflicting atomic sections.
$\psi \rightarrow$ predefined threshold $\in [0, 1]$.
$\delta_i^k \rightarrow$ remaining execution length of s_i^k when interfered by s_j^l.
$s\left(s_i^k\right) \rightarrow$ start time of s_i^k. $s\left(s_i^k\right)$ is updated each time s_i^k aborts and retries to the start time of the new retry.
$s\left(s_j^l\right) \rightarrow$ the same as $s\left(s_i^k\right)$ but for s_j^l.
$cp_h^u(\theta) \rightarrow$ recorded checkpoint in transaction s_h^u for newly accessed object θ.
Result: which atomic section of s_i^k or s_j^l aborts

foreach *newly accessed* θ *requested by any transaction* s_h^u **do**
    Add a checkpoint $cp_h^u(\theta)$
end
if $s\left(s_i^k\right) < s\left(s_j^l\right)$ **then**
    if $p\left(s_i^k\right) > p\left(s_j^l\right)$ **then**
        s_j^l aborts and retreats to $cp_j^l(\theta_{ij}^{kl})$;
        Remove all checkpoints in s_j^l recorded after $cp_j^l(\theta_{ij}^{kl})$
    else
        $c_{ij}^{kl} = len(s_j^l)/len(s_i^k)$;
        $\alpha_{ij}^{kl} = ln(\psi)/(ln(\psi) - c_{ij}^{kl})$;
        $\alpha = \left(len(s_i^k) - \delta_i^k\right)/len(s_i^k)$;
        if $\alpha \leq \alpha_{ij}^{kl}$ **then**
            s_i^k aborts and retreats to $cp_i^k(\theta_{ij}^{kl})$;
            Remove all checkpoints in s_i^k recorded after $cp_i^k(\theta_{ij}^{kl})$
        else
            s_j^l aborts and retreats to $cp_j^l(\theta_{ij}^{kl})$;
            Remove all checkpoints in s_j^l recorded after $cp_j^l(\theta_{ij}^{kl})$
        end
    end
else
    Swap s_i^k and s_j^l;
end

its minimum value (i.e., 0), and α_{ij}^{kl} approaches its maximum value (i.e., 1) (steps 9, 10). Otherwise, c_{ij}^{kl} approaches its maximum value (i.e., ∞), and α_{ij}^{kl} approaches its minimum value (i.e., 0). Ψ is a predefined threshold that lies in $[0,1]$. The remaining execution length of s_i^k (i.e., δ_i^k) is used to calculate α (step 11). If s_i^k still has much work to do, then δ_i^k approaches $len(s_i^k)$ and α approaches 0. Otherwise, α approaches 1. If $len(s_i^k)$ is much longer than $len(s_j^l)$, or s_i^k still has much work to do when interfered by s_j^l, then α tends to be smaller than α_{ij}^{kl}. Consequently, s_i^k aborts in favour of s_j^l. When s_i^k aborts upon a conflict with s_j^l on object θ_{ij}^{kl}, then checkpoints in s_i^k recorded after $cp_i^k(\theta_{ij}^{kl})$ are removed (step 14). Prior checkpoints to $cp_i^k(\theta_{ij}^{kl})$ remain the same. Also, if s_j^l aborts in favour of s_i^k, then all checkpoints in s_j^l recorded after $cp_j^l(\theta_{ij}^{kl})$ are removed (steps 7, 17).

8.2.2 CP-FBLT

Algorithm 7 illustrates FBLT integrated with checkpointing to give CP-FBLT. A new checkpoint is recorded for each newly accessed object θ by any transaction s_h^u (step 2). Checkpoint is recorded when θ is accessed for the first time because any further changes to θ will be discarded upon conflict. Each transaction s_i^k can be aborted during T_i for at most Ω_i^k times before s_i^k becomes a non-preemptive transaction. η_i^k records the number of times s_i^k has already been aborted up to now. If s_i^k and s_j^l have not joined the m_set yet, then they are preemptive transactions. Preemptive transactions resolve conflicts using CPLCM (step 5). Thus, CP-FBLT defaults to CPLCM when the conflicting transactions (s_i^k and s_j^l) have not reached their Ωs (Ω_i^k and Ω_j^l). η_i^k is incremented each time s_i^k is aborted as long as $\eta_i^k < \Omega_i^k$ (steps 8 and 22). Otherwise, s_i^k is added to the $m\_$ set and priority of s_i^k is increased to m_prio (steps 10 to 12 and 24 to 26). When the priority of s_i^k is increased to m_prio, s_i^k becomes a non-preemptive transaction. Non-preemptive transactions cannot be aborted by other preemptive transactions, nor by any other real-time job (steps 18 to 30). Non-preemptive transactions can conflict with each other. The m_set can hold at most m concurrent transactions because there are m processors in the system. $r(s_i^k)$ records the time s_i^k joined the m_set (steps 11 and 25). When non-preemptive transactions conflict together (step 31), the transaction that joined m_set first becomes the transaction that commits first (steps 33 and 36). When s_i^k aborts due to a conflict on θ_{ij}^{kl} with s_j^l, then s_i^k retreats to $cp_i^k(\theta_{ij}^{kl})$. All checkpoints recorded after $cp_i^k(\theta_{ij}^{kl})$ are removed (steps 20, and 37). Similarly, s_j^l removes all checkpoints recorded after $cp_j^l(\theta_{ij}^{kl})$ if aborted by s_i^k (step 34).

8.3 CP-FBLT Retry Cost

In the following Claims, it is assumed that t_{cp}^c is the maximum time taken to construct a single checkpoint. t_{cp}^r is the maximum time to remove a single checkpoint. Let initial access of s_i^k to objects $\theta 1, \theta 2, ..., \theta g, ..., \theta z$ be in that order. $\Theta_i^k(\theta_g) \subseteq \Theta_i^k$ is the set of distinct objects accessed by s_i^k for the first time after s_i^k's first access to θ_g. If all objects before θg are not

Algorithm 7: CP-FBLT

Data:
s_i^k: interfered transaction.
s_j^l: interfering transaction.
Ω_i^k: maximum number of times s_i^k can be aborted during T_i.
η_i^k: number of times s_i^k has already been aborted up to now.
$m_$set: contains at most m non-preemptive transactions. m is number of processors.
m_prio: priority of any transaction in $m_$set. m_prio is higher than any priority of any real-time task.
$r(s_i^k)$: time point at which s_i^k joined $m_$set.
$cp_h^u(\theta)$: recorded checkpoint in transaction s_h^u for newly accessed object θ
Result: which transaction, s_i^k or s_j^l, aborts

1 **foreach** *newly accessed* θ *requested by any transaction* s_h^u **do**
2   Add a checkpoint $cp_h^u(\theta)$
3 **end**
4 **if** s_i^k, $s_j^l \notin m\_set$ **then**
5   Apply CPLCM (Algorithm 6);
6   **if** s_i^k *is aborted* **then**
7   **if** $\eta_i^k < \Omega_i^k$ **then**
8   Increment η_i^k by 1;
9   **else**
10   Add s_i^k to $m_$set;
11   Record $r(s_i^k)$;
12   Increase priority of s_i^k to m_prio;
13   **end**
14   **else**
15   Swap s_i^k and s_j^l;
16   Go to Step 6;
17   **end**
18 **else if** $s_j^l \in m\_set, s_i^k \notin m\_set$ **then**
19   s_i^k aborts and retreats to $cp_i^k(\theta_{ij}^{kl})$;
20   Remove all checkpoints in s_i^k recorded after $cp_i^k(\theta_{ij}^{kl})$;
21   **if** $\eta_i^k < \Omega_i^k$ **then**
22   Increment η_i^k by 1;
23   **else**
24   Add s_i^k to $m_$set;
25   Record $r(s_i^k)$;
26   Increase priority of s_i^k to m_prio;
27   **end**
28 **else if** $s_i^k \in m\_set, s_j^l \notin m\_set$ **then**
29   Swap s_i^k and s_j^l;
30   Go to Step 18;
31 **else**
32   **if** $r(s_i^k) < r(s_j^l)$ **then**
33   s_j^l aborts and retreats to $cp_j^l(\theta_{ij}^{kl})$;
34   Remove all checkpoints in s_j^l recorded after $cp_j^l(\theta_{ij}^{kl})$;
35   **else**
36   s_i^k aborts and retreats to $cp_i^k(\theta_{ij}^{kl})$;
37   Remove all checkpoints in s_i^k recorded after $cp_i^k(\theta_{ij}^{kl})$;
38   **end**
39 **end**

shared between s_i^k and any other transaction, then ∇_{i*}^k is the time interval between start of s_i^k and the first access to θg by s_i^k. So, ∇_{i*}^k is the time interval between start of s_i^k and the first access to any shared object between s_i^k and any other transaction.

Claim 80. *Assume only two transaction s_i^k and s_j^l conflicting together. Let s_i^k begins at time $S\left(s_i^k\right)$ and s_j^l begins at time $S\left(s_j^l\right)$. Let $\triangle = S\left(s_j^l\right) - S\left(s_i^k\right)$. In the absence of checkpointing, retry cost of s_i^k due to s_j^l is given by*

$$BASE\_RC_{ij}^{kl} \leq \begin{cases} len\left(s_j^l\right) + \triangle & , -len\left(s_j^l\right) \leq \triangle \leq len\left(s_i^k\right) \\ 0 & , Otherwise \end{cases} \tag{8.1}$$

$BASE\_RC_{ij}^{kl}$ is upper bounded by

$$len\left(s_j^l + s_i^k\right) \tag{8.2}$$

which is the same upper bound given by Claim 4.

Proof. Due to absence of checkpointing, s_i^k aborts and retries from its beginning due to s_j^l. So, s_i^k retries for the period starting from $S\left(s_i^k\right)$ to the end of execution of s_j^l. s_j^l ends execution at $S\left(s_j^l\right) + len\left(s_j^l\right)$. If $S\left(s_j^l\right) < S\left(s_i^k\right) - len\left(s_j^l\right)$, then s_j^l finishes execution before start of s_i^k and there will be no conflict. Also, if $S\left(s_j^l\right) > S\left(s_i^k\right) + len\left(s_i^k\right)$, then s_j^l starts execution after s_i^k finishes execution and there will be no conflict. Thus, (8.1) follows. Equation (8.2) is derived by substitution of $\triangle$ by its maximum value (i.e., $len\left(s_i^k\right)$). Claim follows. □

Claim 81. *Assume only two transactions s_i^k and s_j^l conflicting on one object θ. Let ∇_j^l be the time interval between the start of s_j^l and the first access to θ. Similarly, let ∇_i^k be the time interval between the start of s_i^k and the first access to θ. Let $\triangle$ be the time difference between start of s_j^l to the start of s_i^k. So, $\triangle < 0$ if s_j^l starts before s_i^k. Under checkpointing, s_i^k aborts and retries due to s_j^l for*

$$RC0_{ij}^{kl} \leq \begin{cases} \begin{array}{l} len\left(s_j^l\right) - \nabla_i^k + \triangle \\ + \left(t_{cp}^r + t_{cp}^c\right) \left|\Theta_i^k(\theta)\right| \end{array} & , if \begin{array}{l} \triangle \geq \nabla_i^k - len\left(s_j^l\right) \\ \triangle \leq len\left(s_i^k\right) - \nabla_j^l \end{array} \\ 0 & , Otherwise \end{cases} \tag{8.3}$$

where $\left|\Theta_i^k(\theta)\right|$ is the number of objects in $\Theta_i^k(\theta)$. $RC0_{ij}^{kl}$ is upper bounded by

$$len\left(s_j^l + s_i^k\right) - \nabla_j^l - \nabla_i^k + \left(t_{cp}^r + t_{cp}^c\right) \left|\Theta_i^k(\theta)\right| \tag{8.4}$$

Proof. As s_i^k and s_j^l conflict only on one object θ, there will be no conflict before both s_i^k and s_j^l access θ. Retry cost of s_i^k due to s_j^l is derived by Claim 80 excluding parts of s_i^k and s_j^l before both transactions access θ. Thus, excluding the parts of s_i^k and s_j^l that do not cause conflict. So, $len\left(s_i^k\right)$ in Claim 80 is substituted by $len\left(s_i^k\right) - \nabla_i^k$. $len\left(s_j^l\right)$ is substituted by $len\left(s_j^l\right) - \nabla_j^l$. $\triangle$ in Claim 80 is substituted by $\triangle + \nabla_j^l - \nabla_i^k$. Claim follows. □

Claim 82. *Assume only two transactions s_i^k and s_j^l conflicting on a number of objects $\theta_1, \theta_2 \dots \theta_z$. Let ∇_{i*}^k be the time interval between start of s_i^k and the first access to the first object accessed by s_i^k and shared with s_j^l (e.g., θ_i). Let ∇_{j*}^l be the time interval between start of s_j^l and the first access to the first object accessed by s_j^l and shared with s_i^k (e.g., θ_j). θ_i and θ_j may not be the same. With checkpointing, retry cost of s_i^k due to s_j^l is upper bounded by*

$$RC1_{ij}^{kl} \leq len\left(s_i^k + s_j^l\right) - \nabla_{i*}^k - \nabla_{j*}^l + \left(t_{cp}^r + t_{cp}^c\right) |\Theta_i^k(\theta_i)| \tag{8.5}$$

Proof. Proof follows directly from Claim 81 by maximizing (8.4). $len\left(s_i^k\right)$, as well as, $len\left(s_j^l\right)$ in (8.4) cannot be changed. Thus, by choosing minimum values of ∇_i^k and ∇_j^l that correspond to shared objects between s_i^k and s_j^l, (8.4) is maximized. Claim follows. □

Claim 83. *If s_j^l is conflicting indirectly (through transitive retry) with s_i^k, then s_i^k is assumed to retreat to the first shared object between s_i^k and any other transaction in calculating the upper bound of retry cost of s_i^k due to s_j^l.*

Proof. If s_j^l is conflicting indirectly with s_i^k, then s_j^l is accessing an object θ that does not belong to Θ_i^k. In this case, to get an upper bound for the retry cost of s_i^k due to s_j^l, s_i^k is assumed to retreat to the first shared object between s_i^k and any other transaction as given by (8.5). Claim follows. □

Claim 84. *Assume only two non-preemptive transactions s_i^k and s_j^l under CP-FBLT. With checkpointing, retry cost of s_i^k due to direct or indirect conflict with s_j^l is upper bounded by*

$$RC2_{ij}^{kl} \leq len\left(s_j^l\right) - \nabla_{i*}^k + \left(t_{cp}^r + t_{cp}^c\right) |\Theta_i^k(\theta_i)| \tag{8.6}$$

where ∇_{i*}^k is the length between start of s_i^k and the first access to θ_i. θ_i is the first shared object between s_i^k and any other transaction.

Proof. Proof follows directly from Claims 81, 82 and 83 except that s_j^l must have become non-preemptive before s_i^k. So, s_j^l starts execution non-preemptively before s_i^k. Otherwise, by definition of CP-FBLT, s_j^l will not be able to abort s_i^k. Thus, $\triangle$ must not exceed 0. Claim follows. □

Claim 85. *Let s_i^k be a non-preemptive transaction under CP-FBLT. Let χ_i^k be the set of transactions conflicting (directly or indirectly) with s_i^k. Each transaction $s_j^l \in \chi_i^k$ belongs to a distinct task. Transactions in χ_i^k are organized in non-increasing order of $RC2_{ij}^{kl}$ for each $s_j^l \in \chi_i^k$. Total retry cost of non-preemptive transaction s_i^k due to other non-preemptive transactions is upper bounded by*

$$RC3_i^k \leq \sum_{a=1}^{a=min\left(|\chi_i^k|, m-1\right)} RC2_i^k\left(\chi_i^k(a)\right) \tag{8.7}$$

where $\chi_i^k(a)$ is the a^{th} transaction in χ_i^k. If $\chi_i^k(a) = s_j^l$, then $RC2_i^k\left(\chi_i^k(a)\right) = RC2_{ij}^{kl}$.

Proof. By definition of CP-FBLT, a transaction s_i^k can be preceded by at most $m-1$ non-preemptive transactions. As non-preemptive transactions are organized in the order they become non-preemptive, no two non-preemptive transactions can belong to the same task. Maximum retry cost of non-preemptive s_i^k occurs when: 1) s_i^k is preceded by at most $m-1$ transactions conflicting with s_i^k. 2) Each conflicting transaction s_j^l to s_i^k must have one of the highest $m-1$ values for $RC2_{ij}^{kl}$. 3) Non-preemptive transactions preceding s_i^k are executing sequentially. Thus, retry cost of non-preemptive s_i^k can be upper bounded by Claim 84 for at most the first $m-1$ transactions in χ_i^k. If the third condition is not satisfied, then (8.7) still gives a correct, but not tight, upper bound. Claim follows. □

Claim 86. *Under CP-FBLT, a preemptive transaction s_i^k aborts and retries for at most*

$$RC4_i^k \leq \Omega_i^k \left(len\left(s_i^k\right) - \nabla_{i*}^k + \left(t_{cp}^r + t_{cp}^c\right)|\Theta_i^k(\theta_i)|\right) \tag{8.8}$$

where ∇_{i*}^k is the length between start of s_i^k and the first access to θ_i. θ_i is the first shared object between s_i^k and any other transaction.

Proof. No transaction will make preemptive s_i^k aborts and retries before $min\left(\nabla_{i*}^k\right)$. By checkpointing, s_i^k will not retreat earlier than $min\left(\nabla_{i*}^k\right)$. By definition of CP-FBLT, preemptive s_i^k aborts for at most Ω_i^k times before it becomes non-preemptive. Claim follows. □

Claim 87. *The total retry cost of any job τ_i^x under CP-FBLT due to 1) conflicts with other transactions during an interval L. 2) release of higher priority jobs during execution of preemptive transactions is upper bounded by*

$$RC(L)_{to}^i = \sum_{\forall s_i^k}\left(\left(t_{cp}^r + t_{cp}^c\right)|\Theta_i^k| + RC4_i^k + RC3_i^k\right) + RC_{re}(L) \tag{8.9}$$

where $RC_{re}(L)$ is the retry cost resulting from the release of higher priority jobs during execution of preemptive transactions. $RC_{i_{re}}(L)$ is calculated by (4.11) for G-EDF, and (4.17) for G-RMA schedulers.

Proof. Following Claims 83, 85, 86 and 65, Claim follows. □

Any newly released task τ_i^x can be blocked by m lower priority non-preemptive transactions. Blocking time $D(\tau_i^x)$ of any job τ_i^x is independent of checkpointing. Thus, $D(\tau_i^x)$ is calculated by Claim 66. Claim 67 is used to calculate response time under CP-FBLT where $RC_{i_{to}}(T_i)$ is calculated by (8.9).

8.4 CP-FBLT versus FBLT

Claim 88. *Following notions in Section 4.3, total utilization of CP-FBLT is equal or better than FBLT's if for each transaction, s_i^k, time cost of constructing and removing each checkpoint divided by minimum distance between start of s_i^k and the first access to the first shared* *object between s_i^k and any other transaction s_j^l does not exceed*

$$\frac{\Omega_i^k + min\left(|\chi_i^k|, m-1\right)}{1 + \Omega_i^k + min\left(|\chi_i^k|, m-1\right)}$$

where χ_i^k is defined in Claim 85. If each transaction, s_i^k, conflicts with at least one other transaction, s_j^l, then the lower bound over time cost of constructing and removing each checkpoint divided by minimum distance between start of s_i^k and the first access to the first shared object between s_i^k and any other transaction s_j^l should not exceed 0.5 to achieve better total utilization for CP-FBLT than total utilization of FBLT.

Proof. Let upper bound on retry cost of any task τ_i^x during T_i under FBLT be denoted as RC_i^{ncp}. RC_i^{ncp} is calculated by Claim 1 in [56]. Let upper bound on retry cost of any task τ_i^x during T_i under CP-FBLT be denoted as RC_i^{cp}. RC_i^{cp} is calculated by (8.9). Let B_i be the upper bound on blocking time of any newly released task τ_i^x during T_i due to lower priority jobs. B_i is the same for both CP-FBLT and FBLT. B_i is calculated by Claim 2 in [56]. For CP-FBLT total utilization to be better than total utilization of FBLT:

$$\sum_{\forall\tau_i} \frac{c_i + RC_i^{cp} + B_i}{T_i} \leq \sum_{\forall\tau_i} \frac{c_i + RC_i^{ncp} + B_i}{T_i} \tag{8.10}$$

$\because$ B_i and c_i are the same for each τ_i under CP-FBLT and FBLT, then (8.10) holds if:

$$\forall\tau_i,\ RC_i^{cp} \leq RC_i^{ncp}$$

$$\begin{aligned}
& \textstyle\sum_{\forall s_i^k} \left(t_{cp}^r + t_{cp}^c\right) |\Theta_i^k| \\
+ \quad & \textstyle\sum_{\forall s_i^k} \Omega_i^k \left(len\left(s_i^k\right) - \nabla_{i*}^k + \left(t_{cp}^r + t_{cp}^c\right) |\Theta_i^k\left(\theta_i\right)|\right) \\
+ \quad & \textstyle\sum_{\forall s_i^k} \left(\sum_{a=1}^{min\left(|\chi_i^k|, m-1\right)} \left(len\left(\chi_i^k\left(a\right)\right) - \nabla_{i*}^k + \left(t_{cp}^r + t_{cp}^c\right) |\Theta_i^k\left(\theta_i\right)|\right)\right) \\
\leq \quad & \textstyle\sum_{\forall s_i^k} \left(\Omega_i^k len\left(s_i^k\right) + \sum_{a=1}^{min\left(|\nu_i^k|, m-1\right)} len\left(\nu_i^k\left(a\right)\right)\right)
\end{aligned} \tag{8.11}$$

where ν_i^k is the set of at most $m-1$ longest transactions conflicting directly or indirectly with s_i^k. Thus, transactions in ν_i^k are organized in non-increasing order of $len(s_j^l)_{\forall s_j^l \in \nu_i^k}$. $\because$ $\Theta_i^k\left(\theta_i\right) \subseteq \Theta_i^k$, $\therefore$ $\forall s_i^k$, $\left(t_{cp}^r + t_{cp}^c\right) |\Theta_i^k| \geq \left(t_{cp}^r + t_{cp}^c\right) |\Theta_i^k\left(\theta_i\right)|$. Thus, (8.11) holds if

$$\begin{aligned}
& \textstyle\sum_{\forall s_i^k} \left(t_{cp}^r + t_{cp}^c\right) |\Theta_i^k| \\
+ \quad & \textstyle\sum_{\forall s_i^k} \Omega_i^k \left(len\left(s_i^k\right) - \nabla_{i*}^k + \left(t_{cp}^r + t_{cp}^c\right) |\Theta_i^k|\right) \\
+ \quad & \textstyle\sum_{\forall s_i^k} \left(\sum_{a=1}^{min\left(|\chi_i^k|, m-1\right)} \left(len\left(\chi_i^k\left(a\right)\right) - \nabla_{i*}^k + \left(t_{cp}^r + t_{cp}^c\right) |\Theta_i^k|\right)\right) \\
\leq \quad & \textstyle\sum_{\forall s_i^k} \left(\Omega_i^k len\left(s_i^k\right) + \sum_{a=1}^{min\left(|\nu_i^k|, m-1\right)} len\left(\nu_i^k\left(a\right)\right)\right)
\end{aligned} \tag{8.12}$$

$\therefore$ (8.12) holds if for each s_i^k

$$\begin{aligned} & \left(t^r_{cp}+t^c_{cp}\right)|\Theta_i^k| \\ + \quad & \Omega_i^k\left(len\left(s_i^k\right)-\nabla_{i*}^k+\left(t^r_{cp}+t^c_{cp}\right)|\Theta_i^k|\right) \\ + \quad & \sum_{a=1}^{min\left(|\chi_i^k|,m-1\right)}\left(len\left(\chi_i^k\left(a\right)\right)-\nabla_{i*}^k+\left(t^r_{cp}+t^c_{cp}\right)|\Theta_i^k|\right) \\ \leq \quad & \Omega_i^k len\left(s_i^k\right)+\sum_{a=1}^{min\left(|\nu_i^k|,m-1\right)} len\left(\nu_i^k\left(a\right)\right) \end{aligned} \tag{8.13}$$

$$\begin{aligned} \therefore \quad & \left(1+\Omega_i^k+min\left(|\chi_i^k|,m-1\right)\right)\left(t^r_{cp}+t^c_{cp}\right)|\Theta_i^k|-\left(\Omega_i^k+min\left(|\chi_i^k|,m-1\right)\right)\nabla_{i*}^k \\ + \quad & \Omega_i^k len\left(s_i^k\right)+\sum_{a=1}^{min\left(|\chi_i^k|,m-1\right)} len\left(\chi_i^k\left(a\right)\right) \\ \leq \quad & \Omega_i^k len\left(s_i^k\right)+\sum_{a=1}^{min\left(|\nu_i^k|,m-1\right)} len\left(\nu_i^k\left(a\right)\right) \end{aligned} \tag{8.14}$$

By definition of χ_i^k and ν_i^k, $\nu_i^k(a)\geq\chi_i^k(a),\forall a$. Thus, (8.14) holds if

$$\left(1+\Omega_i^k+min\left(|\chi_i^k|,m-1\right)\right)\left(t^r_{cp}+t^c_{cp}\right)|\Theta_i^k|\leq\left(\Omega_i^k+min\left(|\chi_i^k|,m-1\right)\right)\nabla_{i*}^k$$

$$\therefore \frac{\left(t^r_{cp}+t^c_{cp}\right)|\Theta_i^k|}{\nabla_{i*}^k}\leq\frac{\Omega_i^k+min\left(|\chi_i^k|,m-1\right)}{1+\Omega_i^k+min\left(|\chi_i^k|,m-1\right)} \tag{8.15}$$

Let $\kappa_i^k=\Omega_i^k+min\left(|\chi_i^k|,m-1\right)$. $\therefore$ (8.15) becomes

$$\therefore \frac{\left(t^r_{cp}+t^c_{cp}\right)|\Theta_i^k|}{\nabla_{i*}^k}\leq\frac{1}{1+\frac{1}{\kappa_i^k}} \tag{8.16}$$

If $\kappa_i^k\rightarrow\infty$, then the upper bound over $\frac{\left(t^r_{cp}+t^c_{cp}\right)|\Theta_i^k|}{\nabla_{i*}^k}$ to achieve better total utilization for CP-FBLT over FBLT is 1. If each transaction s_i^k conflicts with at least another transaction, then the minimum value for $\kappa_i^k=1$ by substituting $\Omega_i^k=0$ (i.e., minimum value for Ω_i^k). Thus, the lower bound over $\frac{\left(t^r_{cp}+t^c_{cp}\right)|\Theta_i^k|}{\nabla_{i*}^k}$ to achieve better total utilization for CP-FBLT over FBLT is 0.5. Claim follows. □

8.5 Conclusion

Past research on real-time CMs focused on developing different conflict resolution strategies for transactions. Except for LCM (Chapter 5), no policy was made to reduce the length of conflicting transactions. In this Chapter, we analysed effect of checkpointing over FBLT CM. Analysis shows that CP-FBLT has equal or better total utilization than FBLT if, for each s_i^k, time cost of constructing and removing each checkpoint starting from the first shared object, θ_i, between s_i^k and any other transaction does not exceed half the length between start of s_i^k and the first access to θ_i. Some CMs make no use of checkpointing due to behaviour of that CM (e.g, under PNF, all non-preemptive transactions are non-conflicting).

Chapter 9

Implementation and Experimental Evaluations

Having established upper bounds for retry cost and response time of different contention managers, and the conditions under which each one is preferred. We now would like to understand how each CM retries in practice (i.e., on average) compared with that of competitor methods. Also, we would like to know the effect of each CM on response time of real-time jobs compared to lock-free and locking protocols. Since this can only be understood experimentally, we implement ECM, RCM, LCM, PNF, FBLT, OMLP, RNLP and lock-free and conduct experimental studies.

The rest of this Chapter is organized as follow: Section 9.1 outlines the used methodology to generate different tasksets and run experiments. Section 9.2 outlines properties of used task sets and atomic sections for comparing different contention managers, locking protocols and lock-free. Section 9.3 presents used metrics to evaluate performance of different synchronization techniques. Section 9.4 discusses experimental results.

9.1 Methodology

We used the ChronOS real-time Linux kernel [48] and the RSTM library [144]. We modified RSTM to include implementations of ECM, RCM, LCM, PNF and FBLT contention managers and to support checkpointing. We modified ChronOS to include implementations of G-EDF and G-RMA schedulers and Global OMLP [22, 29] and RNLP [149] locking protocols. For the retry-loop lock-free implementation, we used a loop that reads an object and attempts to write to the object using a compare-and-swap (CAS) instruction. The task retries until the CAS succeeds. We use an 8 core, 2GHz AMD Opteron platform.

Experiments ran over a set of tasksets. Each taskset consists of a number of tasks. Each task is represented by a single thread. Our system assumes sporadic task model. The least common multiplier of periods of all tasks in a single taskset is called *Hyperperiod* of this taskset. Each task τ_i in a single taskset runs a number of jobs(i.e., instances) equal to hyperperiod of the taskset divided by period of the task, T_i.

9.2 Tasksets

We collected properties of tasksets from [12, 20, 21, 23–25, 27–29, 96, 104, 115] with some modifications due to insertion of transactions into tasks. Each task's period, T_i, is an integer uniformly distributed from $[10ms, 100ms]$. Utilization of each task, u_i, is derived from three uniform distributions: $[0.001, 0.1]$ (light), $[0.1, 0.4]$ (medium), and $[0.5, 0.9]$(heavy). Worst case execution time for each task, e_i, is calculated as $e_i = u_i.T_i$. Total utilization of all tasks in a given specific taskset should not exceed $\hat{U}$. Different tasksets are generated for each $\hat{U} \in \{2, 4, 6, 8\}$ where 8 is the maximum number of cores on the tested platform as given in Section 9.1. Tasks are added to each taskset until $\hat{U}$ is reached. If last task makes total utilization exceeds $\hat{U}$, then the last task is removed.

Each task has a number of atomic sections(transactions). Atomic section properties are probabilistically controlled using three parameters: the maximum(max_{Tx}) and minimum(min_{Tx}) lengths of any atomic section within the task, and the total(to_{Tx}) length of atomic sections within any task. Each of the 3 parameters (min_{Tx},max_{Tx} and to_{Tx}) is derived from 3 uniform distributions: $[0, 0.3)$ (light), $[0.3, 0.6)$ (medium), $[0.6, 1]$ (heavy). Each value of min_{Tx},max_{Tx} and to_{Tx} is relative to e_i. Thus, each of min_{Tx},max_{Tx} and to_{Tx} does not exceed e_i. max_{Tx} is chosen such that $max_{Tx} \leq to_{Tx}$. Similarly, min_{Tx} is chosen such that $min_{Tx} \leq max_{Tx}$. Total number of shared objects, N^r, is either 5, 20 or 40. Number of objects per each atomic section, N_i^r, is chosen from 3 uniform distributions: $[0, 0.3)$ (light), $[0.3, 0.6)$ (medium), $[0.6, 1]$ (heavy). As lock-free cannot access more than one object in one atomic operation, tasks share one object per transaction when lock-free is included in comparison. Different parameters for tasks and transactions are summarized in Table 9.1. Appendix A presents properties of each taskset where: 1) "ID" is the taskset ID. 2) "no_tasks" is number of tasks within each taskset. 3) "total_tx_dis", "max_tx_dis" and "min_tx_dis" are the distributions for deriving to_{Tx}, max_{Tx} and min_{Tx}, respectively. 4) "total_no_obj" is total number of shared objects among all tasks (i.e., N^r). 5) "no_obj_tx_dis" is the distribution for deriving ratio of accessed objects per each transaction relative to total number of shared objects (i.e., N_i^r). 6) "u_i_dis" is the distribution to derive utilization of each task relative to total utilization (i.e., $u_i/\hat{U}$).

To simplify reading task properties and writing results, we used MySQL database to store properties of tasksets and results. Properties for tasksets and results are organized into 4 major tables: 1) First table has properties for each taskset. Properties include number of tasks under each dataset, utilization cap and tasks' utilization distribution. Properties also include distributions of total, maximum and minimum length of atomic sections within any task under each dataset. 2) Second table holds properties about each task of each taskset. Properties include worst case execution time, period and relative deadline. 3) Third table holds properties on the structure of each task of each dataset. Each task is organized into a number of portions. Each portion is either an atomic or non-atomic section. Each record in the table represents one portion of a specific task of a specific taskset. Properties for each portion include portion type (i.e., atomic or non-atomic), portion length and accessed objects in atomic sections. 4) Fourth table holds results. Each record represents a job of a specific task under a specific taskset. Results of each job include absolute start and end time for this job, retry cost under different CMs and lock-free and blocking time under locking protocols.

Table 9.1: Tasksets' and transactions' properties

$\hat{U}$	$\{2, 4, 6, 8\}$
T_i	uniformly chosen from $[10ms, 100ms]$
u_i	Uniformly chosen from $[0.001, 0.1]$ (light), $[0.1, 0.4]$ (medium), and $[0.5, 0.9]$(heavy). $\sum_{\forall i} u_i \leq \hat{U}$
e_i	$u_i.T_i$
to_{Tx}	Uniformly chosen from $[0, 0.3)$ (light), $[0.3, 0.6)$ (medium), $[0.6, 1]$ (heavy) relative to e_i
max_{Tx}	Uniformly chosen from $[0, 0.3)$ (light), $[0.3, 0.6)$ (medium), $[0.6, 1]$ (heavy) relative to e_i. $max_{Tx} \leq to_{Tx}$
min_{Tx}	Uniformly chosen from $[0, 0.3)$ (light), $[0.3, 0.6)$ (medium), $[0.6, 1]$ (heavy) relative to e_i. $min_{Tx} \leq max_{Tx}$
N^r	5, 20, 40
N_i^r	Uniformly chosen from $(0, 0.3)$ (light), $[0.3, 0.6)$ (medium), $[0.6, 1]$ (heavy)

9.3 Performance Measurements

We record two measurements to compare different CMs against lock-free and locking protocols. Deadline Satisfaction Ratio (DSR) and Average Retry Cost (Avg_RC). Deadline Satisfaction Ratio (DSR) is calculated for each taskset as ratio between number of jobs that successfully met their deadlines to total number of jobs for the specified taskset under a specified synchronization technique. Thus, for taskset i

$$DSR = \frac{Deadlines\, met}{Total\, deadlines}$$

"DSR" shows contribution of different synchronization techniques to response time of each job. DSR_B^A measures how much DSR of synchronization technique A exceeds DSR of synchronization technique B, on average.

Retry cost of each job τ_k^l contributes to response time of τ_k^l, consequently to DSR. Thus, we measure Average Retry Cost (Avg_RC) for a each taskset. "Avg_RC" is the average value for retry cost of all jobs under the specified taskset under a specific CM. Thus, for taskset i

$$Avg\_RC = Average(Rertry\,cost\,for\,each\,job\,for\,taskset\,i)$$

"Avg_RC" for different CMs is compared against average retry cost for lock-free and blocking time for locking protocols. Blocking time for dataset i under locking protocols is the summation of time taken by each critical section in each job under taskset i to obtain all required locks.

9.4 Results

Appendixes B and C record "DSR" and "Avg_RC" for different CMs compared to lock-free and locking protocols. Atomic lock-free instruction accesses only object. Thus, lock-free is not applied to any taskset with multiple objects per critical section.

9.4.1 General results for DSR

1. DSR results for all tasksets are given in Appendix B. CP-FBLT has the highest DSR compared to other contention managers. $DSR_{ECM}^{CP-FBLT} = 31.3\%$. 77.2% of tasksets have higher DSR under CP-FBLT than ECM. 8.8% of tasksets have equal DSR under both CP-FBLT and ECM. $DSR_{LCM}^{CP-FBLT} = 31.2\%$. 76.7% of tasksets have higher DSR under CP-FBLT than LCM. 8.8% of tasksets have equal DSR under both CP-FBLT and LCM. $DSR_{PNF}^{CP-FBLT} = 8.8\%$. 51.9% of tasksets have higher DSR under CP-FBLT than PNF. 7.8% of tasksets have equal DSR under CP-FBLT and PNF. $DSR_{FBLT}^{CP-FBLT} = 4.6\%$. 54.1% of tasksets have higher DSR under CP-FBLT than FBLT. 9.5% of tasksets have equal DSR under CP-FBLT and FBLT.

2. On contrast to lock-free, proposed contention managers use different policies to resolve conflicts. Thus, more jobs meet their deadlines under STM than lock-free. $DSR_{LF}^{CP-FBLT} = 34.6\%$. 68.9% of tasksets have higher DSR under CP-FBLT than lock-free. 2.5% of tasksets have equal DSR under both CP-FBLT and lock-free. $DSR_{LF}^{FBLT} = 28.5\%$. 63.9% of tasksets have higher DSR under FBLT than lock-free. 7.4% of tasksets have equal DSR under both FBLT and lock-free. $DSR_{LF}^{PNF} = 32.4\%$. 61.5% of tasksets have higher DSR under PNF than lock-free. 8.2% of tasksets have equal DSR under both PNF and lock-free. $DSR_{LCM}^{LF} = 2.7\%$. 55.7% of tasksets have higher DSR under LCM than lock-free. 13.9% of tasksets have equal DSR under both LCM and lock-free. $DSR_{ECM}^{LF} = 5.3\%$. 49.2% of tasksets have higher DSR under ECM than lock-free. 16.4% of tasksets have equal DSR under both ECM and lock-free.

3. Generally, more jobs meet their deadlines under OMLP and RNLP than any contention manager by 12.4% and 13.7% on average, respectively. OMLP uses group locking that protects all required objects in the same atomic section by the same resource. Current implementation of RNLP requires a priori knowledge of requested objects per each atomic section. Thus, OMLP and RNLP have the advantage of a priori knowledge of requested objects per each atomic section. Only PNF has the same advantage. But PNF induces a lot of overhead because it is a centralized contention manager. To examine effect of a-priori knowledge of required objects, we modified FBLT to FBLT-P. Under FBLT-P, each transactions accesses all required objects at the beginning of the transaction. Thus, each transaction knows a priori what objects are going to be accessed. Results show that DSR of FBLT-P increased over DSR of CP-FBLT by 19.5%. Additionally, atomic sections under OMLP and RNLP do not have to retry, nor to make decisions upon a conflict in each retry. Under OMLP and RNLP, tasks suspend after making requests for acquiring specific locks. After obtaining all required locks, atomic sections can proceed without abortion upon a conflict. Thus, locking protocols make a decision only once regarding which atomic section to proceed, whereas a transaction can invoke the contention manager many times even if the contention manager is going to make the same decision. 71% of tasksets under FBLT-P have DSR lower than DSR of OMLP by at most $DSR^{OMLP}_{FBLT-P} = 14.6\%$. 71.9% of tasksets under FBLT-P have DSR lower than DSR of RNLP by at most $DSR^{RNLP}_{FBLT-P} = 15.9\%$. 65.6% of tasksets under CP-FBLT have DSR lower than DSR of OMLP by at most $DSR^{OMLP}_{CP-FBLT} = 33.2\%$. 66.2% of tasksets under CP-FBLT have DSR lower than DSR of RNLP by at most $DSR^{RNLP}_{CP-FBLT} = 34.5\%$. 58.6% of tasksets under FBLT have DSR lower than DSR of OMLP by at most $DSR^{OMLP}_{FBLT} = 37.8\%$. 58.8% of tasksets under FBLT have DSR lower than DSR of RNLP by at most $DSR^{RNLP}_{FBLT} = 39\%$. 55.3% of tasksets under PNF have DSR lower than DSR of OMLP by at most $DSR^{OMLP}_{PNF} = 42\%$. 55% of tasksets under PNF have DSR lower than DSR of RNLP by at most $DSR^{RNLP}_{PNF} = 43.3\%$. 39% of tasksets under LCM have DSR lower than DSR of OMLP by at most $DSR^{OMLP}_{LCM} = 64.4\%$. 38% of tasksets under LCM have DSR lower than DSR of RNLP by at most $DSR^{RNLP}_{LCM} = 65.7\%$. 39% of tasksets under ECM have DSR lower than DSR of OMLP by at most $DSR^{OMLP}_{ECM} = 64.5\%$. 38% of tasksets under ECM have DSR lower than DSR of RNLP by at most $DSR^{RNLP}_{ECM} = 65.8\%$.

4. Figure 9.1 is an example for tasksets [1,27], [541,567], [1081,1107] and [1621,1647] with total utilizations of 2,4,6 and 8 respectively. Generally, Different CMs show similar DSR performance to each other with no single optimal CM (i.e., no single CM alawys shows the best DSR). DSR performance of different CMs ranges from 85% to 100%. Under specified tasksets, number and length of transactions per each task is small. Thus, contention is low. Generally, different CMs show equal or higher DSR than Lock-free.

5. Figure 9.2 is an example of tasksets [28,36], [568,576], [1108,1116] and [1648,1656] with total utilizations 2, 4, 6 and 8, respectively. Total transactional length per each task

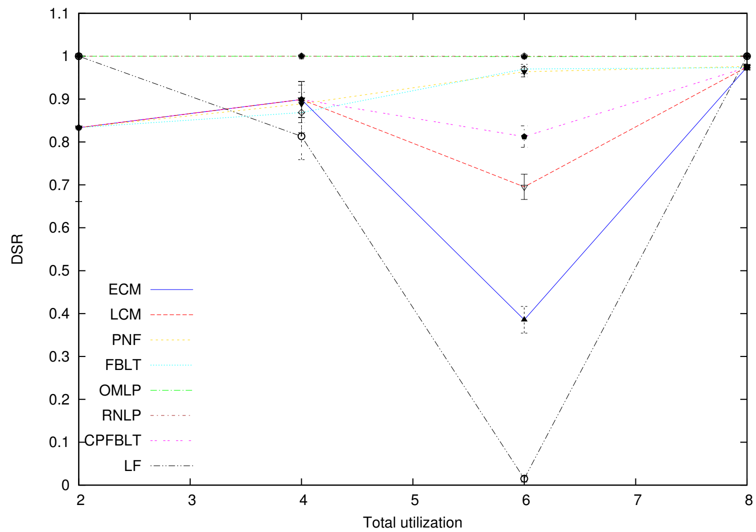


Figure 9.1: DSR for Tasksets 2, 542, 1082 and 1622

lies between 33% and 66% of each task length. Total number of objects is 5 which increases contention. However, maximum transactional length does not exceed 33% of each task length. Thus, each task contains at least 2 transactions. With large number of transactions and small length of each transaction, different CMs- except for CP-FBLT, show poor DSR performance at low utilizations (up to 4). However, as utilization increases, DSR for PNF and FBLT increases. DSR of ECM and LCM degrades to 0 mostly after total utilization of 4. Generally, CP-FBLT shows the best DSR among CMs. Lock-free shows poor DSR compared to other synchronization techniques. However, lock-free's DSR starts to increase after total utilization of 6 to coincide with DSR of locking protocols at total utilization of 8.

6. Figure 9.3 is an example for tasksets [37,54], [577,594], [1657,1134] and [1648,1674] with total utilizations 2, 4, 6 and 8, respectively. Tasksets have the same properties as in point 5 except than total number of objects is increased to 20 and 40. Thus, more objects are accessed by each transaction. In conrast to point 5, mostly at low utilizations, ECM and LCM show equal or better DSR compared to FBLT and PNF. However, DSR of ECM and LCM mostly degrades after total utilization of 4. DSR for PNF and FBLT mostly increase after total utilization of 4 and coincide with CP-FBLT after total utilization of 6. PNF and FBLT show better DSR than ECM and LCM after total utilization of 5. CP-FBLT still shows the best DSR among CMs.

7. Figure 9.4 is an example of tasksets [55,66], [595,606], [1135,1146] and [1675,1686] with total utilizations 2, 4, 6 and 8, respectively. Maximum transactional length can reach total transactional length for any task. Thus, each task contains at least 1 transaction. Total number of objects is 5 and 20. CP-FBLT shows the best DSR among CMs. Mostly, DSR of ECM and LCM degrad continuously. DSR of PNF and

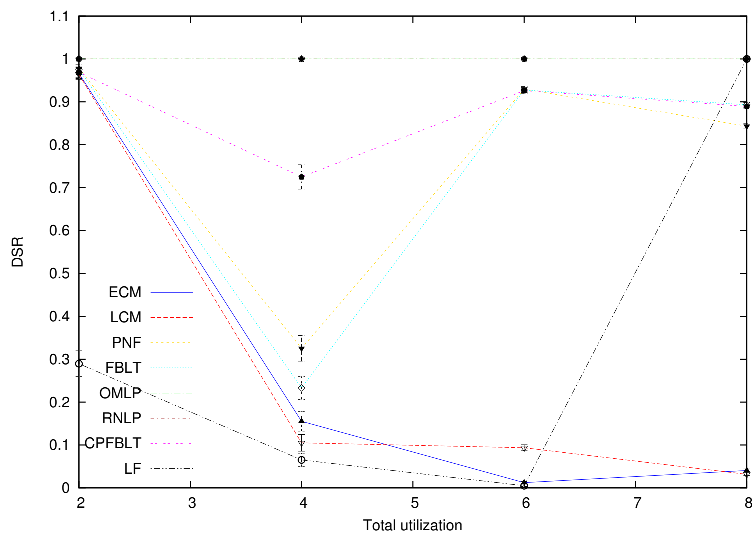


Figure 9.2: DSR for Tasksets 28, 568, 1108 and 1648

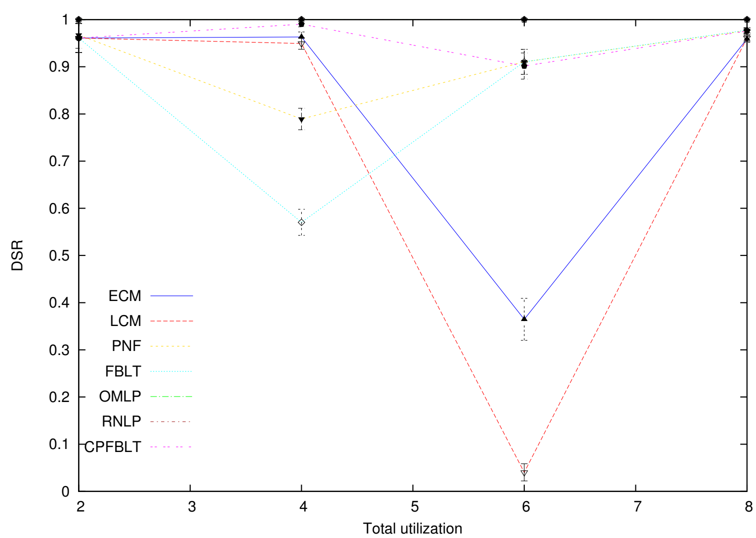


Figure 9.3: DSR for Tasksets 38, 578, 1118 and 1658

FBLT mostly increases after total utilization of 4. Lock-free shows poor DSR. After total utilization of 6, DSR of lock-free increases until it coincides with locking protocols at total utilization of 8.

8. Figure 9.5 is an example of tasksets [67,106], [607,646], [1147,1186] and [1687,1726] with total utilizations 2, 4, 6 and 8, respectively. Tasks have the same properties as tasks in point 7 except that more objects are accessed per each transaction. As more objects are accessed, additional overhead is introduced by CP-FBLT. Thus, CP-FBLT shows the best DSR up to total utilization of 6. Afterwards, DSR of CP-FBLT

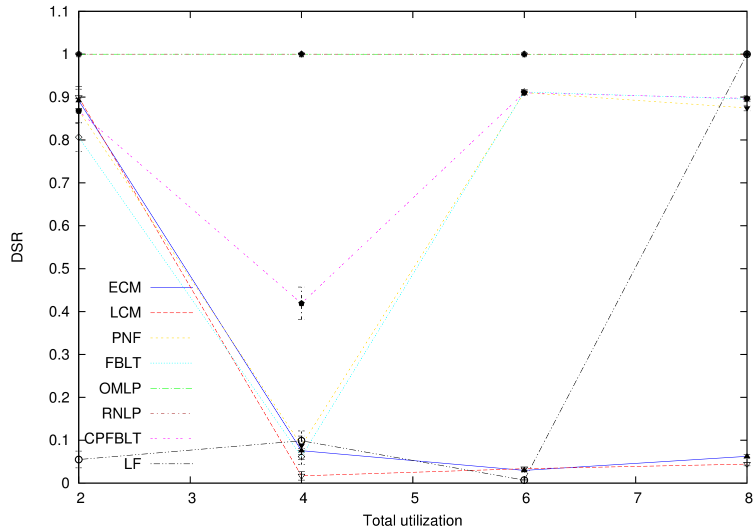


Figure 9.4: DSR for Tasksets 55, 595, 1135 and 1675

degrades and DSR of FBLT increases. Generally, between total utilizations of 6 and 8, either CP-FBLT or FBLT shows the best DSR. DSR of ECM and LCM degrades as total utilization increases. PNF shows equal or better DSR than ECM and LCM. Lock-free shows poor DSR. After total utilization of 6, DSR of lock-free increases until it coincides with locking protocols at total utilization of 8.

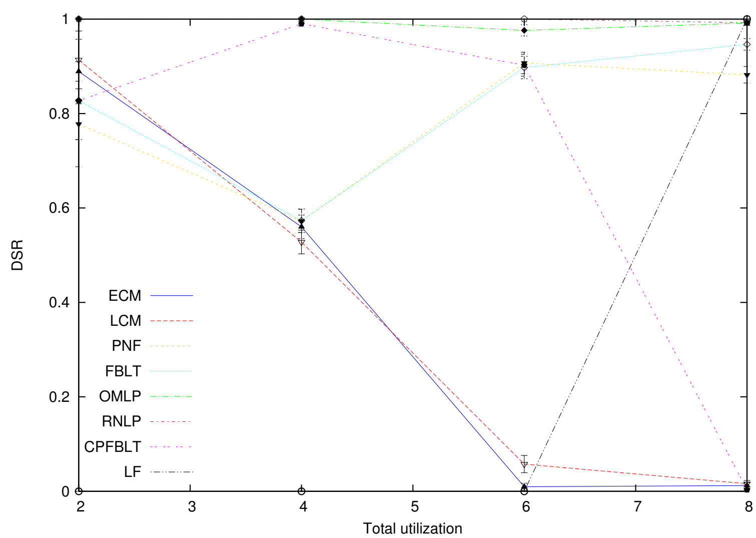


Figure 9.5: DSR for Tasksets 83, 623, 1163 and 1703

9. Figure 9.6 is an example for tasksets [107,108], [647,648], [1187,1188] and [1727,1728] with total utilizations 2, 4, 6 and 8, respectively. ECM and LCM show better DSR

than other CMs up to total utilization of 4.5. Then PNF and CP-FBLT show better DSR up to total utilization of 7.5. Finally, FBLT show better DSR up to 8.

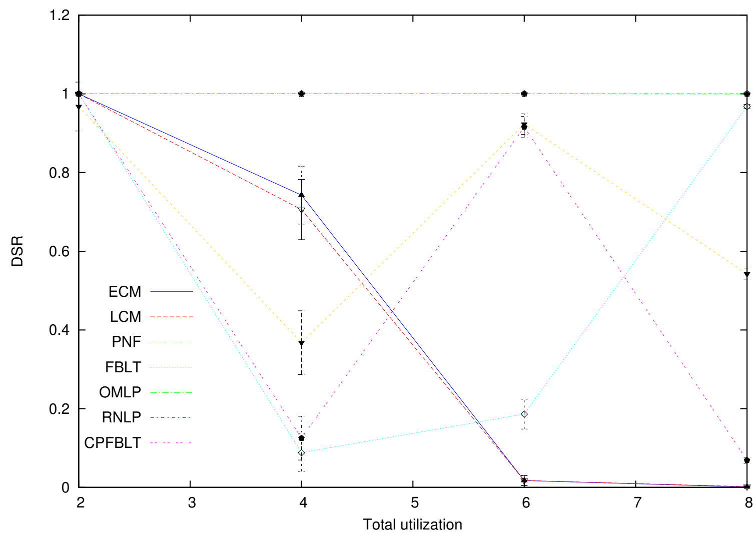


Figure 9.6: DSR for Tasksets 107, 647, 1187 and 1727

10. Figure 9.7 is an example for tasksets [109,137], [649,677], [1189,1217] and [1729,1757] with total utilizations 2, 4, 6 and 8, respectively. Each task contains a large number of small length transactions. DSR for OMLP and RNLP decrease after total utilization of 4 and 6, respectively. DSR for lock-free is poor but increases after total utilization of 6. Different CMs show varying DSRs up to total utilization of 4. Afterwards, PNF, FBLT and CP-FBLT show better DSR than ECM and LCM.

11. Figure 9.8 is an example of tasksets [138,155], [678,695], [1218,1235] and [1758,1775] with total utilizations 2, 4, 6 and 8, respectively. Tasks have the same properties as in point 10 except that maximum transactional length increases. Thus, total number of transactions per any task can be lower than that specified in point 10. PNF and CP-FBLT show equal or better DSR than other CMs starting from total utilization of 4.

12. Figure 9.9 is an example of tasksets [156,232], [696,772], [1236,1312] and [1776,1852] with total utilizations of 2, 4, 6 and 8, respectively. Generally, CP-FBLT shows better DSR than other CMs. After total utilization of 5 or 6, DSRs of FBLT and PNF sometimes outperform DSR of CP-FBLT. Lock-free shows poor DSR. After total utilization of 6, DSR of lock-free increases until it coincides with locking protocols at total utilization of 8.

13. Figure 9.10 is an example of tasksets [233,243], [773,783], [1313,1323] and [1853,1863] with total utilizations 2, 4, 6 and 8, respectively. Generally, CP-FBLT shows equal or

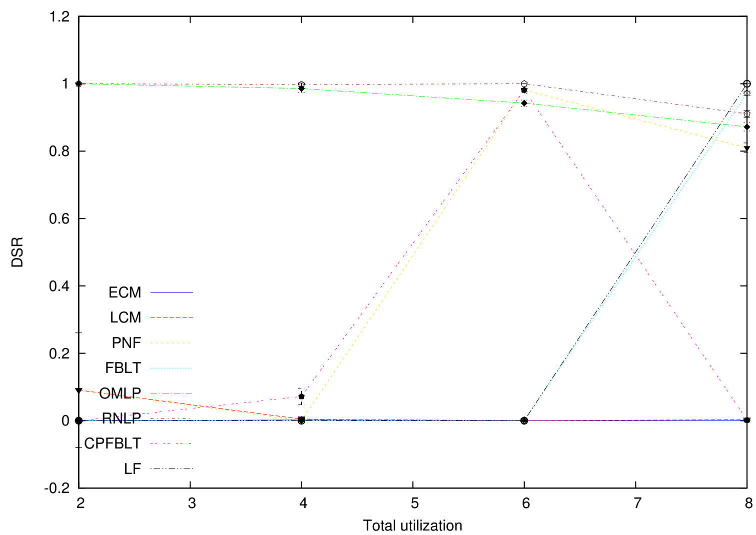


Figure 9.7: DSR for Tasksets 110, 650, 1190 and 1730

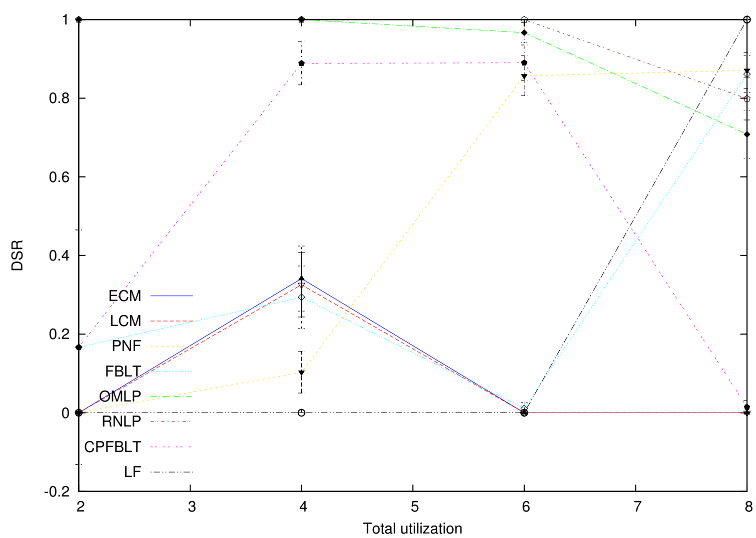


Figure 9.8: DSR for Tasksets 138, 678, 1218 and 1758

better DSR than other CMs. DSR of CP-FBLT degrades after total utilization of 4 and rises again after 6.

14. Figure 9.11 is an example of tasksets [244,255], [784,795], [1324,1335] and [1864,1875] with total utilizations 2, 4, 6 and 8, respectively. Each task contains, at most, 2 long transactions. CP-FBLT mostly shows equal or better DSR to other CMs in total utilizations of [2,5] and [7,8]. PNF shows best DSR among CMs in totoal utilizations of [5,7]. FBLT shows equal or better DSR to CP-FBLT after total utilization of 7. DSR of OMLP degrades after total utilization of 6. Lock-free shows poor DSR. After

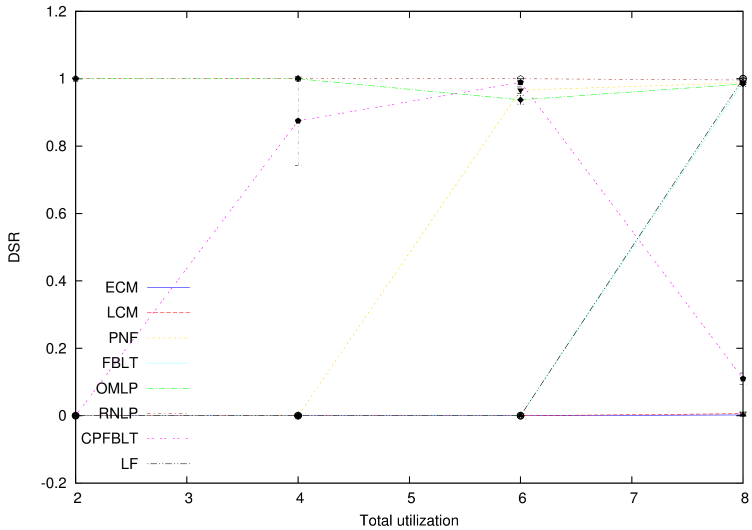


Figure 9.9: DSR for Tasksets 165, 705, 1245 and 1785

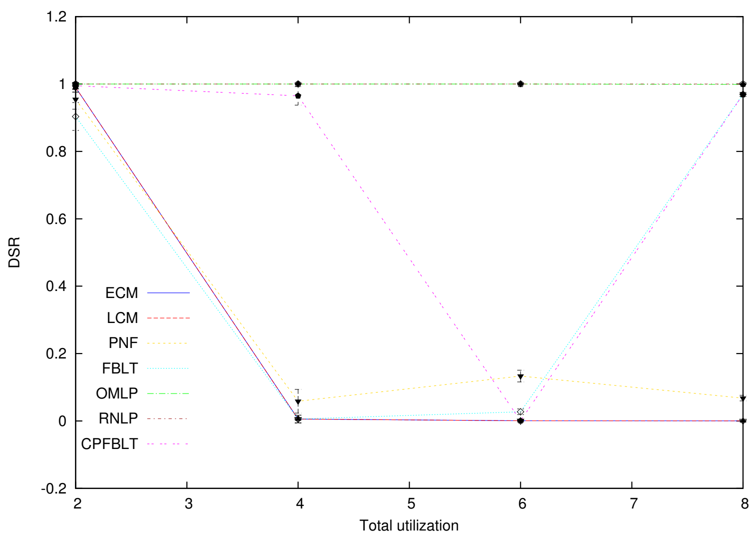


Figure 9.10: DSR for Tasksets 233, 773, 1313 and 1853

total utilization of 6, DSR of lock-free increases until it coincides with RNLP at total utilization of 8.

15. Figure 9.12 is an example of tasksets [256,261], [796,801], [1336,1341] and [1876,1881] with total utilizations 2, 4, 6 and 8, respectively. CMs show the same DSR pattern as in point 14 except that CP-FBLT and FBLT show equal or better DSR compared to other CMs for all utilizations.

16. Figure 9.13 is an example of tasksets [262,270], [802,810], [1342,1350] and [1882,1890].

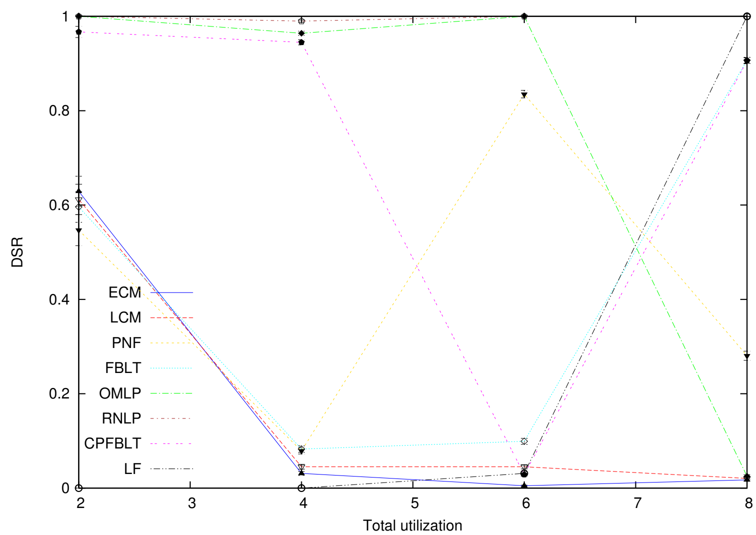


Figure 9.11: DSR for Tasksets 244, 784, 1324 and 1864

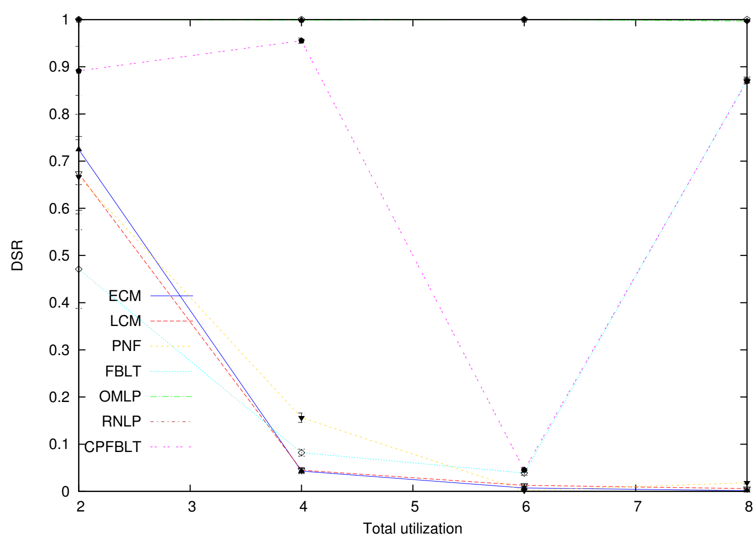


Figure 9.12: DSR for Tasksets 256, 796, 1336 and 1876

CMs show DSR pattern similar to points 14 and 15.

A closer look at CP-FBLT

1. DSR of CP-FBLT is at least 0.9 usually when summation of transactions' lengths per each task is at least 1/3 of each task's length, accessed objects per each transaction is at most 2/3 of total shared objects and total utilization is lower than 8. 48.7% of tasksets achieve DSR in [0.9,1[under CP-FBLT.

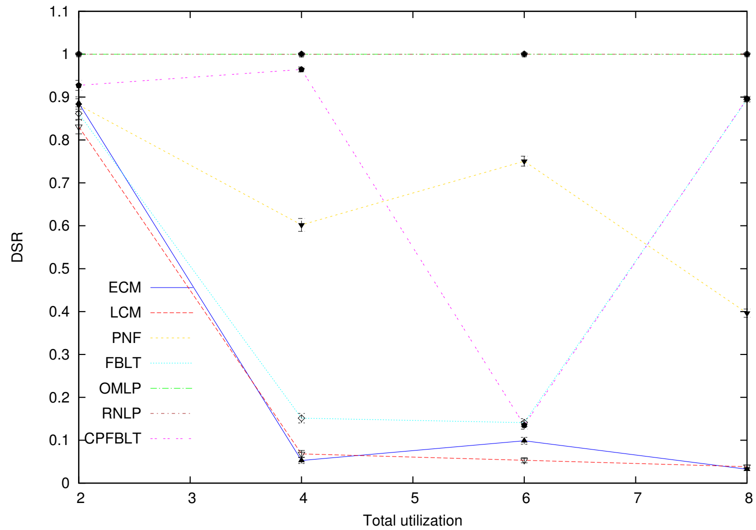


Figure 9.13: DSR for Tasksets 262, 802, 1342 and 1882

2. DSR of CP-FBLT is between 0.8 and 0.9 usually when summation of transactions' lengths per each task is at least 1/3 of each task's length, each transaction accesses a small portion of total shared objects and total utilization is lower than 6. 10.3% of tasksets achieve DSR in [0.8,0.9[under CP-FBLT.

3. DSR of CP-FBLT is between 0.7 and 0.8 usually when summation of transactions' lengths per each task is between 1/3 and 2/3 of each task's length, number of transactions per each task is small and total utilization does not exceed 4. 4.8% of tasksets achieve DSR in [0.7,0.8[under CP-FBLT.

4. DSR of CP-FBLT is between 0.6 and 0.7 usually when summation of transactions' lengths per each task is at least 1/3 of each task's length, number of transactions per each task is small and total utilization approaches 2 or 8. 2% of tasksets achieve DSR in [0.6,0.7[under CP-FBLT.

5. DSR of CP-FBLT is between 0.5 and 0.6 usually when summation of transactions' lengths per each task is at least 1/3 of each task's length and total utilization does not exceed 6. 1.3% of tasksets achieve DSR in [0.5,0.6[under CP-FBLT.

6. DSR of CP-FBLT is between 0.4 and 0.5 usually when summation of transactions' lengths per each task is at least 1/3 of each task's length and total utilization is close to 2 or 8. 1.4% of tasksets achieve DSR in [0.4,0.5[under CP-FBLT.

7. DSR of CP-FBLT is between 0.2 and 0.4 usually when summation of transactions' lengths per each task is at least 1/3 of each task's length and total utilization is at least 4. 2% of tasksets achieve DSR in [0.2,0.4[under CP-FBLT.

8. DSR of CP-FBLT is between 0.1 and 0.2 usually when summation of transactions' lengths per each task is at least 2/3 of each task's length and accessed objects per each transaction does not exceed 2/3 of total shared objects. 5% of tasksets achieve DSR in [0.1,0.2[under CP-FBLT.

9. DSR of CP-FBLT is at most 0.1 usually when summation of transactions' lengths per each task is at least 2/3 of each task's length, accessed objects per each transaction are at least 2/3 of total shared objects and total utilization is at least 6. 24.6% of tasksets achieve DSR in [0,0.1[under CP-FBLT.

10. CP-FBLT has the highest DSR among proposed contention managers. DSR comparison between DSR of CP-FBLT and other contention managers was given by Point 1 in Section 9.4.1.

A closer look at DSR of FBLT

1. DSR of FBLT is at least 0.9 usually when summation of transactions' lengths per each task is at least 1/3 of each task's length, number of transactions per each task is large, accessed objects per each transaction is at most 2/3 of total shared objects and total utilization is at least 6. 37.8% of tasksets achieve DSR in [0.9,1[under FBLT.

2. DSR of FBLT is between 0.8 and 0.9 usually when summation of transactions' lengths per each task is at least 1/3 of each task's length, accessed objects per each transaction is at most 2/3 of total shared objects and total utilization does not equal 4. 14.2% of tasksets achieve DSR in [0.8,0.9[under FBLT.

3. DSR of FBLT is between 0.7 and 0.8 usually when summation of transactions' lengths per each task is at least 2/3 of each task's length, number of transactions per each task is small, accessed objects per each transaction is at most 1/3 of total shared objects and total utilization does not equal 4. 4.26% of tasksets achieve DSR in [0.7,0.8[under FBLT.

4. DSR of FBLT is between 0.6 and 0.7 usually when summation of transactions' lengths per each task is at least 1/3 of each task's length, accessed objects per each transaction is at most 2/3 of total shared objects and total utilization is at most 6. 2.5% of tasksets achieve DSR in [0.6,0.7[under FBLT.

5. DSR of FBLT is between 0.4 and 0.6 usually when summation of transactions' lengths per each task is at least 1/3 of each task's length, accessed objects per each transaction is at least 1/3 of total shared objects and total utilization is at most 4. 4.5% of tasksets achieve DSR in [0.4,0.6[under FBLT.

6. DSR of FBLT is between 0.3 and 0.4 usually when summation of transactions' lengths per each task is at least 2/3 of each task's length, number of transactions per each task

is small, accessed objects per each transaction is at least 1/3 of total shared objects and total utilization lies between 4 and 6. 2.2% of tasksets achieve DSR in [0.3,0.4[under FBLT.

7. DSR of FBLT is between 0.2 and 0.3 usually when summation of transactions' lengths per each task is at least 2/3 of each task's length, number of transactions per each task is small, accessed objects per each transaction is at most 2/3 of total shared objects and total utilization is less than 8. 4.4% of tasksets achieve DSR in [0.2,0.3[under FBLT.

8. DSR of FBLT is between 0.1 and 0.2 usually when summation of transactions' lengths per each task is at least 2/3 of each task's length, number of transactions per each task is small, accessed objects per each transaction is at least 1/3 of total shared objects and total utilization is close to 4. 7.3% of tasksets achieve DSR in [0.1,0.2[under FBLT.

9. DSR of FBLT is less than 0.1 usually when summation of transactions' lengths per each task is at least 2/3 of each task's length, accessed objects per each transaction is at least 1/3 of total shared objects and total utilization is between 4 and 6. 22.7% of tasksets achieve DSR in [0,0.1[under FBLT.

10. $DSR^{FBLT}_{ECM} = 26.7\%$. 62.9% of tasksets have higher DSR under FBLT than ECM. 12.2% of tasksets have equal DSR under both FBLT and ECM. $DSR^{FBLT}_{LCM} = 26.5\%$. 62.8% of tasksets have higher DSR under FBLT than LCM. 12.8% of tasksets have equal DSR under both FBLT and LCM. $DSR^{FBLT}_{PNF} = 4.2\%$. 43.2% of tasksets have higher DSR under FBLT than PNF. 10.5% of tasksets have equal DSR under both FBLT and PNF.

A closer look at DSR of PNF

1. PNF can be implemented using locks, or lock-free or combination of both (cas and casX as defined in RSTM_R5). PNF is a centralized CM. Thus, there is a high contention on the main service of PNF from different transactions (even non-conflicting ones). Contention on the main service of PNF can be reduced by avoiding organization of retrying transactions in n_set according to priority. If any retrying transaction s_i^k finds no conflict with current non-preemptive transactions, s_i^k becomes a non-preemptive transaction even if n_set contains another transaction, s_j^l, with higher priority than s_i^k. The same analysis in Chapter 6 applies to the less restricted PNF.

2. DSR of PNF is at least 0.9 usually when summation of transactions' lengths per each task is at most 2/3 of each task's length, number of transactions per each task is small and total utilization does not equal 4. 26.8% of tasksets achieve DSR of at least 0.9 under PNF.

3. DSR of PNF lies within 0.8 and 0.9 usually when summation of transactions' lengths per each task is at least 1/3 of each task's length, number of transactions per each task is small, accessed objects per each transaction does not exceed 1/3 of shared objects and total utilization is at least 6. 13.87% of tasksets achieve DSR in [0.8,0.9[under PNF.

4. DSR of PNF lies within 0.6 and 0.8 usually when summation of transactions' lengths per each task is at least 1/3 of each task's length and number of transactions per each task is small. 12.2% of tasksets achieve DSR in [0.6,0.8[under PNF.

5. DSR of PNF lies within 0.5 and 0.6 usually when summation of transactions' lengths per each task is at least 2/3 of each task's length, number of transactions per each task is high and total utilization is less or greater than 6. 4% of tasksets achieve DSR in [0.5,0.6[under PNF.

6. DSR of PNF lies within 0.4 and 0.5 usually when summation of transactions' lengths per each task is at least 2/3 of each task's length, number of transactions per each task is small and accessed objects per each transaction is at most 2/3 of total shared objects. 3.7% of tasksets achieve DSR in [0.4,0.5[under PNF.

7. DSR of PNF lies within 0.3 and 0.4 usually when summation of transactions' lengths per each task is at least 2/3 of each task's length. 5% of tasksets achieve DSR in [0.3,0.4[under PNF.

8. DSR of PNF lies within 0.2 and 0.3 usually when summation of transactions' lengths per each task is at least 2/3 of each task's length and accessed objects per each transaction is at least 1/3 of total shared objects. 4.82 % of tasksets achieve DSR in [0.2,0.3[under PNF.

9. DSR of PNF lies within 0.1 and 0.2 mostly when summation of transactions' lengths per each task is at least 2/3 of each task's length, accessed objects per each transaction is at least 1/3 of total shared objects and total utilization is at least 4. 8.7% of tasksets achieve DSR in [0.1,0.2[under PNF.

10. DSR of PNF is at most 0.1 usually when summation of transactions' lengths per each task is at least 2/3 of each task's length, number of transactions per each task is small, accessed objects per each transaction is at least 2/3 of total shared objects and total utilization does not equal 6. 21% of tasksets achieve DSR in [0,0.1[under PNF.

11. $DSR_{ECM}^{PNF} = 22.5\%$. 69.1% of tasksets have higher DSR under PNF than ECM. 10% of tasksets have equal DSR under both PNF and ECM. $DSR_{LCM}^{PNF} = 22.4\%$. 68.4% of tasksets have higher DSR under PNF than LCM. 9.4% of tasksets have equal DSR under both PNF and LCM.

A closer look at DSR of LCM

1. DSR of LCM is at least 0.8 mostly when summation of transactions' lengths per each task is at most 2/3 of each task's length, number of transactions is small and total utilization is mostly low(< 4). 25.28% of tasksets achieve DSR of at least 0.8 under LCM.

2. DSR of LCM is between 0.2 and 0.8 mostly when summation of transactions' lengths per each task is at least 1/3 of each task's length, number of transactions is small, each transaction accesses at most 2/3 of shared objects and total utilization is usually at most 6. 14.2% of tasksets achieve DSR in [0.2,0.8[under LCM.

3. DSR of LCM is between 0.1 and 0.2 mostly when summation of transactions' lengths per each task is at least 1/3 of each task's length, each transaction accesses at least 1/3 of shared objects and total utilization is usually at most 6. 2.8% of tasksets achieve DSR in [0.1,0.2[under LCM.

4. DSR of LCM is at most 0.1 mostly when summation of transactions' lengths per each task is at least 1/3 of each task's length and total utilization is at least 6. 57.8% of tasksets achieve DSR in [0,0.1[under LCM.

5. $DSR_{ECM}^{LCM} = 0.13\%$. 38.6% of tasksets have higher DSR under LCM than ECM. 28% of tasksets have equal DSR under both LCM and ECM.

A closer look at DSR of ECM

1. DSR for ECM is at least 0.9 usually if summation of transactions' lengths per each task is at most 2/3 of each task's length, number of transactions per each task is small and total utilization is low(≤ 2). 18.7% of tasksets acheive DSR of at least 0.9 under ECM.

2. DSR for ECM is between 0.8 and 0.9 generally when each task contains at least one transaction of at most 2/3 of task's length at low total utilizations(≤ 4). As total utilization increases, number and length of transactions decrease to keep DSR between 0.8 and 0.9. 7.3% of tasksets acheive DSR in [0.8,0.9[under ECM.

3. DSR for ECM lies within 0.7 and 0.8 generally for small number of transactions per each task at low total utilizations(≤ 2). 3.6% of jobs acheive DSR in [0.7,0.8[under ECM.

4. DSR of ECM lies within 0.4 and 0.7 generally when summation of all transactional lengths per each task is not less than 1/3 of each task's length at low total utilization(≤ 4). DSR decreases as number of transactions per each task decreases and total utilization increases up to 8. 5.8% of tasksets achieve DSR in [0.4,0.7[under ECM.

5. DSR of ECM usually lies within 0.2 and 0.4 when summation of transactions' lengths is at least 1/3 of each task's length at all total utilizations and each task accesses at least 1/3 of objects and total utilization usually does not exceed 6. 4.54% of tasksets achieve DSR in [0.2,0.4[under ECM.

6. DSR of ECM usually lies within 0.1 and 0.2 when summation of transactions' lengths per each task is at least 2/3 of each task's length, number of transactions is small, each task accesses at least 2/3 of shared objects and total utilization is at least 4. 2.5% of tasksets achieve DSR in [0.1,0.2[under ECM.

7. DSR of ECM is usually at most 0.1 when summation of transactions' lengths per each task is at least 2/3 of each task's length and total utilization is usually high($\geq$ 6). 57.5% of tasksets achieve DSR of at most 0.1 under ECM.

9.4.2 General results for Avg_RC

1. Figure 9.14 is an example of tasksets [1,27], [541,567], [1081,1107] and [1621,1647] with total utilizations of 2, 4, 6 and 8, respectively. Average retry cost for different CMs is high compared to locking protocols and lock-free. PNF, FBLT and CP-FBLT generally show shorter average retry cost than ECM and LCM. FBLT and CP-FBLT show close average retry cost at total utilizations 2 and 8. Usually, average retry cost of FBLT is high compared to CP-FBLT and vice versa between total utilizations 2 and 8.

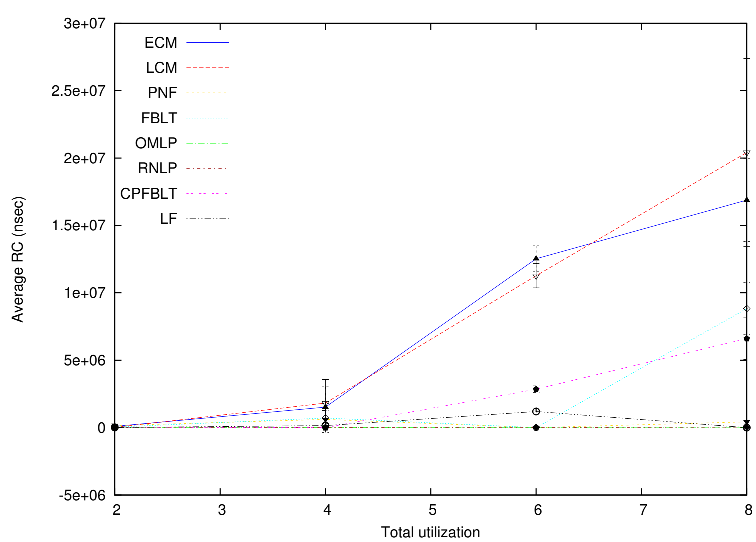


Figure 9.14: Average RC for Tasksets 3, 543, 1083 and 1623

2. Figure 9.15 is an example of tasksets [28,66], [568,606], [1108,1146] and [1648,1686] with total utilizations 2, 4, 6 and 8, respectively. Average retry cost for ECM and

LCM is high compared to locking protocols and lock-free. PNF, FBLT and CP-FBLT generally show shorter average retry cost than ECM and LCM. Average retry cost of ECM and LCM generally increases with increasing total utilization. Average retry cost of ECM and LCM reaches its maximum at total utilization of 6 or 8. Average retry cost of PNF and FBLT increases from total utilization 2 to 4 or 6 then decreases up to total utilization of 6 to be close to locking protocols and lock-free. Average retry cost of CP-FBLT decreases from total utilization 2 to 4 to be close to locking protocols and lock-free.

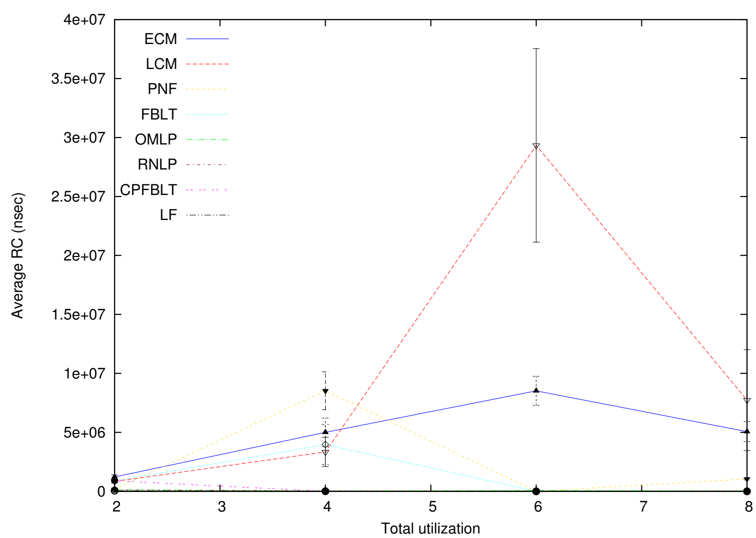


Figure 9.15: Average RC for Tasksets 28, 568, 1108 and 1648

3. Figure 9.16 is an example of tasksets [67,78], [607,618], [1147,1158] and [1687,1698] with total utilizations of 2, 4, 6 and 8, respectively. "Avg_RC" has the same pattern as in point 2 except that average retry cost of CP-FBLT increases after total utilization of 6. At total utilization of 8, average retry cost of CP-FBLT is close to average retry cost of ECM and LCM.

4. Figure 9.17 is an example of tasksets [79,96], [619,636], [1159,1176] and [1699,1716] with total utilizations of 2, 4, 6 and 8, respectively. "Avg_RC" shows the same pattern as in point 3 except that average retry cost of PNF decreases as total utilization approaches 6, then increases again until total utilization of 8.

5. Figure 9.18 is an example of tasksets [97,105], [637,645], [1177,1185] and [1717,1725] with total utilizations of 2, 4, 6 and 8, respectively. "Avg_RC" has the same pattern as in point 4 except that average retry cost of FBLT reaches its maximum at total utilization 4 or 6. Afterwards, average retry cost of FBLT decreases to be close to locking protocols at total utilization of 8.

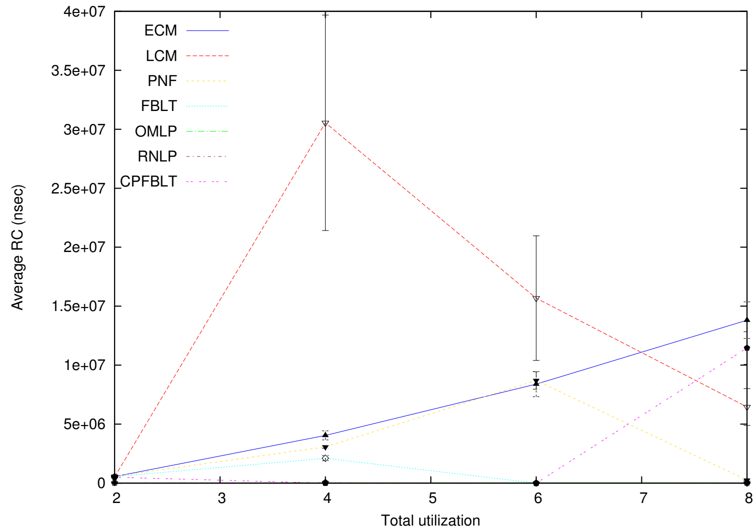


Figure 9.16: Average RC for Tasksets 67, 607, 1147 and 1687

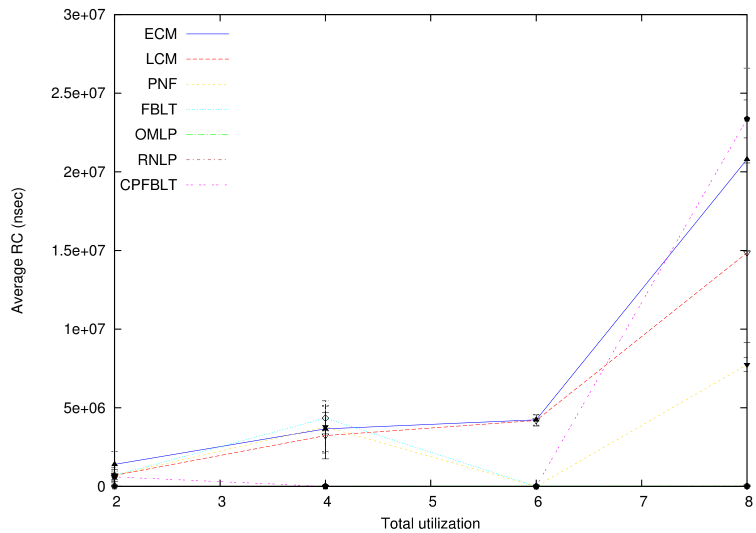


Figure 9.17: Average RC for Tasksets 81, 621, 1161 and 1701

6. Figure 9.19 is an example of tasksets [106,129], [646,669], [1186,1209] and [1721,1749] with total utilizations of 2, 4, 6 and 8, respectively. "Avg_RC" has the same pattern as in point 5 except that average retry cost of CP-FBLT reaches its reaches its minimum value at total utilization of 6. Afterwards, average retry cost of CP-FBLT increases.

7. Figure 9.20 is an example of tasksets [130,138], [670,678], [1210,1218] and [1750,1758] with total utilizations of 2, 4, 6 and 8, respectively. "Avg_RC" has the same patter as in point 6 except that average retry cost of PNF generally increases with increasing total utilization.

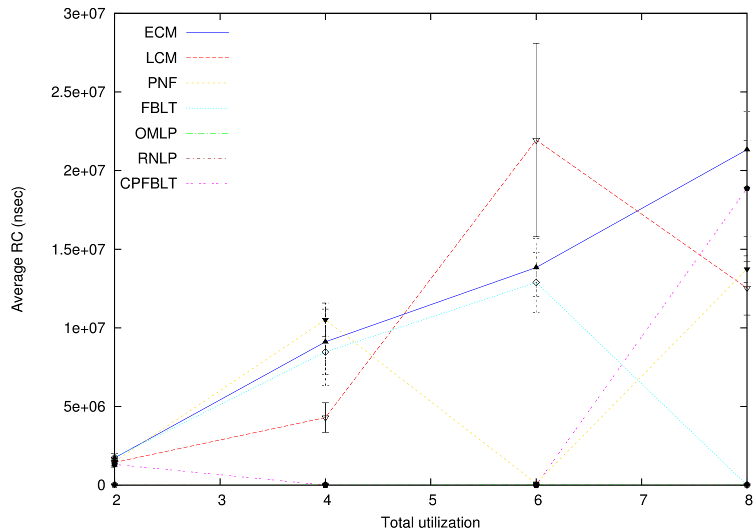


Figure 9.18: Average RC for Tasksets 97, 637, 1177 and 1717

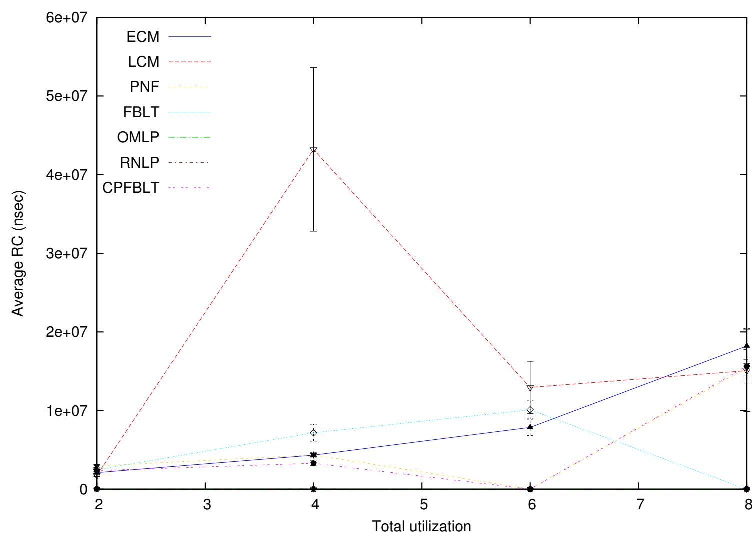


Figure 9.19: Average RC for Tasksets 106, 646, 1186 and 1726

8. Figure 9.21 is an example of tasksets [139,157], [679,697], [1219,1237] and [1759,1777] with total utilizations of 2, 4, 6 and 8, respectively. “Avg_RC” has the same pattern as in point 7 except that average retry cost of PNF generally reaches its maximum value at total utilizations 4 or 6.

9. Figure 9.22 is an example of tasksets [158,177], [698,717], [1238,1257] and [1778,1797] with total utilizations of 2, 4, 6 and 8, respectively. “Avg_RC” has the same pattern as in point 8 except that average retry cost of CP-FBLT decreases between total utilizations of 2 and 6, then increases.

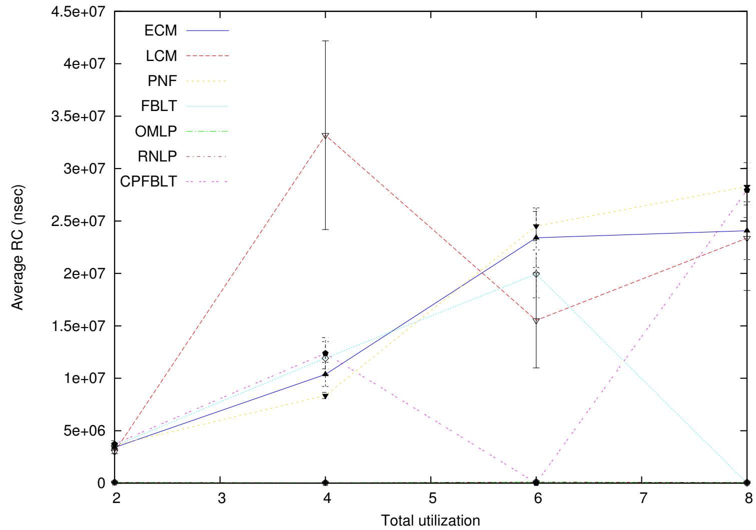


Figure 9.20: Average RC for Tasksets 130, 670, 1210 and 1750

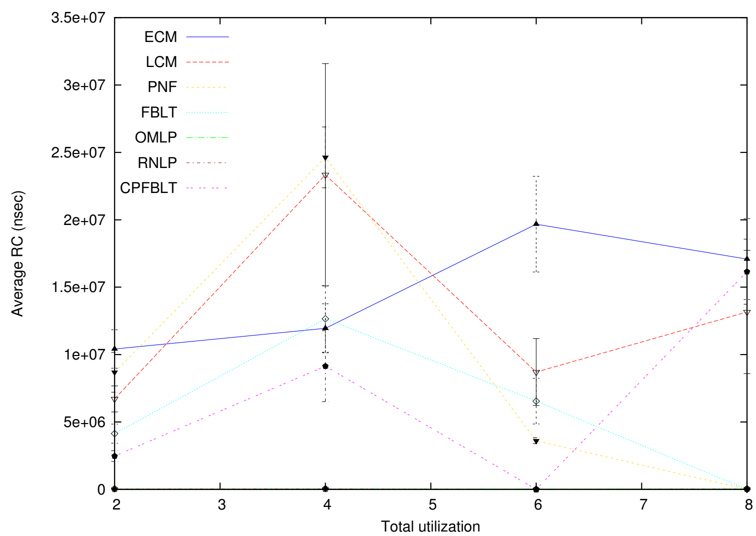


Figure 9.21: Average RC for Tasksets 139, 679, 1219 and 1759

10. Figure 9.23 is an example of tasksets [178,184], [718,724], [1258,1264] and [1798,1804] with total utilizations of 2, 4, 6 and 8, respectively. "Avg_RC" has the same pattern as in point 9 except that average retry cost of CP-FBLT reaches its minimum value within total utilizations 2 and 4.

11. Figure 9.24 is an example of tasksets [185,210], [725,750], [1265,1290] and [1805,1830] with total utilizations 2, 4, 6 and 8, respectively. "Avg_RC" has the same pattern as in point 10 except that average retry cost of FBLT reaches its maximum value at total utilization of 4, then decreases to be close to locking protocols at total utilization of 6.

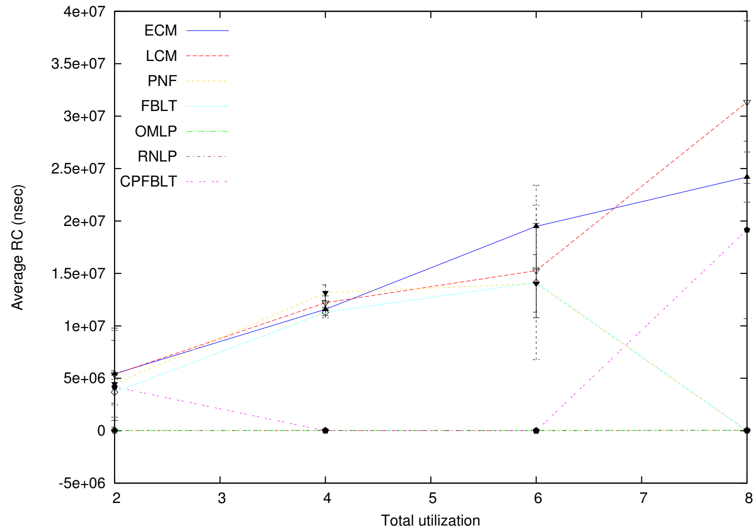


Figure 9.22: Average RC for Tasksets 158, 698, 1238 and 1778

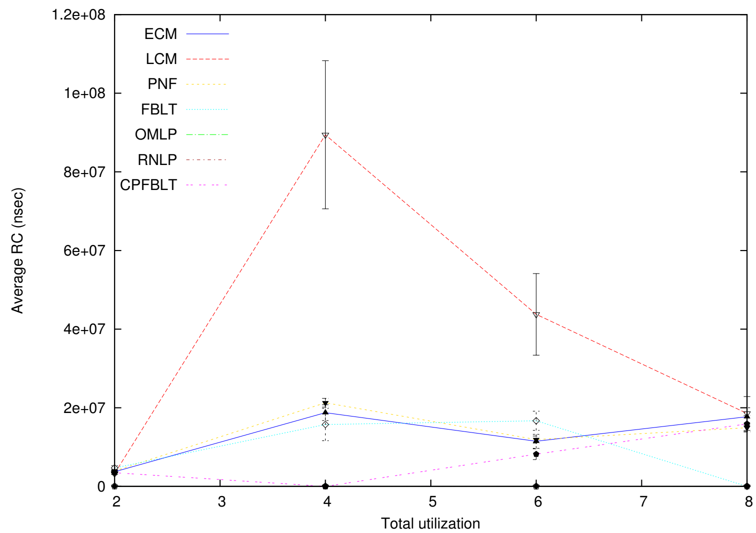


Figure 9.23: Average RC for Tasksets 178, 718, 1258 and 1798

12. Figure 9.25 is an example of tasksets [211,233], [751,773], [1291,1313] and [1831,1853]. "Avg_RC" has the same pattern as in point 11 except that average retry cost of PNF either increases with total utilization or reaches its maximum value at total utilization of 4 or 6.

13. Figure 9.26 is an example of tasksets [234,270], [774,810], [1314,1350] and [1854,1890] with total utilizations of 2, 4, 6 and 8, respectively. "Avg_RC" has the same pattern

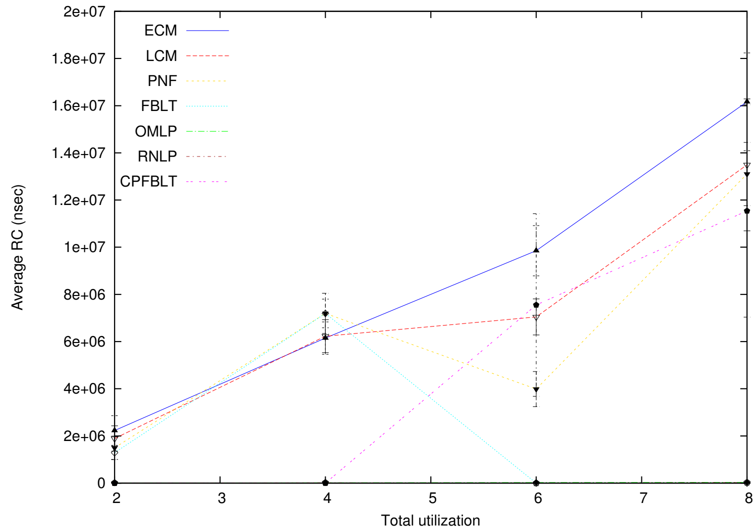


Figure 9.24: Average RC for Tasksets 185, 725, 1265 and 1805

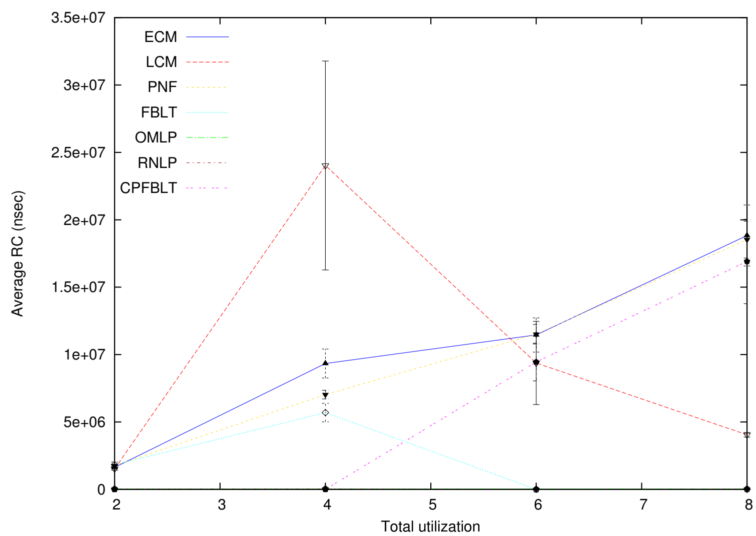


Figure 9.25: Average RC for Tasksets 211, 751, 1291 and 1831

as in point 12 except that average retry cost of FBLT reaches its maximum value at total utilization of 4 or 6. Average retry cost of CP-FBLT reaches its minimum value within total utilizations 2 and 4, and at total utilization of 8.

Closer look at Avg_RC of different CMs

14. CP-FBLT shows shortest Avg_RC among contention managers. 89% of tasksets have shorter Avg_RC under CP-FBLT than ECM. 2.3% of tasksets have equal Avg_RC

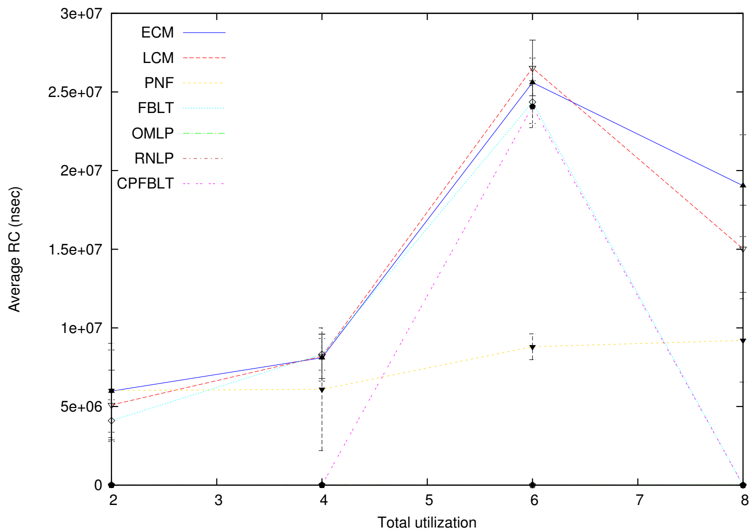


Figure 9.26: Average RC for Tasksets 234, 774, 1314 and 1854

under CP-FBLT and ECM. 80.7% of tasksets have shorter Avg_RC under CP-FBLT than LCM. 2.8% of tasksets have equal Avg_RC under CP-FBLT and LCM. 61.1% of tasksets have shorter Avg_RC under CP-FBLT than PNF. 9.4% of tasksets have equal Avg_RC under CP-FBLT and PNF. 53.1% of tasksets have shorter Avg_RC under CP-FBLT than FBLT. 16.9% of tasksets have equal Avg_RC under CP-FBLT and FBLT. Average retry cost of CP-FBLT is equal or shorter than average retry cost of other CMs when summation of transactions' lengths per each task is at least 1/3 of task's length and number of transactions per each task is small (i.e.,≤ 2) for any total utilization, especially at 4. 52.4% of tasksets have Avg_RC of CP-FBLT shorter or equal to blocking time of each of OMLP and RNLP. 55% of tasksets have equal or shorter Avg_RC of CP-FBLT compared to Avg_RC of lock-free.

15. 86% of tasksets have shorter Avg_RC under FBLT than ECM. 2.3% of tasksets have equal Avg_RC under FBLT and ECM. 77.5% of tasksets have shorter Avg_RC under FBLT than LCM. 2.2% of tasksets have equal Avg_RC under FBLT and LCM. 63.3% of tasksets have shorter Avg_RC under FBLT than PNF. 11.2% of tasksets have equal Avg_RC under FBLT and PNF. Average retry cost of FBLT is equal or shorter than average retry cost of other CMs when summation of transactions' lengths per each task is at least 1/3 of task's length, number of transactions per each task is small (i.e.,≤ 2) and total utilization is at least 8. 39.7% of tasksets have Avg_RC of FBLT shorter or equal to blocking time of each of OMLP and RNLP. 46% of tasksets have equal or shorter Avg_RC of FBLT compared to Avg_RC of lock-free.

16. 75.3% of tasksets have shorter Avg_RC under PNF than ECM. 2% of tasksets have equal Avg_RC under PNF and ECM. 67.5% of tasksets have shorter Avg_RC under PNF than LCM. 1.9% of tasksets have equal Avg_RC under PNF and LCM. . Average

retry cost of PNF is equal or shorter than average retry cost of other CMs when summation of transactions' lengths per each task is at least 1/3 of task's length, number of transactions per each task is small (i.e.,≤ 2) and total utilization is at least 6. 15.8% of tasksets have Avg_RC of PNF shorter or equal to blocking time of each of OMLP and RNLP. 32% of tasksets have equal or shorter Avg_RC of PNF compared to Avg_RC of lock-free.

17. 69.5% of tasksets have shorter Avg_RC under LCM than ECM. 2.3% have equal Avg_RC under LCM and ECM. Average retry cost of LCM is equal or better than average retry cost of other CMs when summation of transactions' lengths per each task is at least 2/3 of task's length, number of transactions per each task is small (i.e.,≤ 3) and total utilization does not exceed 6. 3.5% of tasksets have Avg_RC of LCM shorter or equal to blocking time of each of OMLP and RNLP. Avg_RC of LCM is equal or longer than Avg_RC of lock-free. 8.2% of tasksets have equal or shorter Avg_RC of LCM compared to Avg_RC of lock-free.

18. Average retry cost of ECM is equal or better than average retry cost of other CMs when summation of transactions' lengths per each task is at least 2/3 of task's length, number of transactions per each task is small (i.e.,≤ 3), number of accessed objects per each task is small and total utilization is 2. 2.7% of tasksets have Avg_RC of ECM shorter or equal to blocking time of each of OMLP and RNLP. Avg_RC of ECM is equal or longer than Avg_RC of lock-free. 7.4% of tasksets have equal Avg_RC of ECM compared to Avg_RC of lock-free.

9.5 Results Summary

1. Experiments show that CP-FBLT has the highest DSR among contention managers. Superiority of CP-FBLT results from combining benefits of PNF and LCM into design of FBLT (the base of CP-FBLT). Besides, checkpointing reduces response time of CP-FBLT compared to FBLT. More jobs meet their deadlines under STM than lock-free because of conflict resolution policies. DSR comparison within contention managers and against lock-free is summarized in Table 9.2. "Avg%" is the average increase of DSR of each contention manager in the left column to DSR of each contention manager and lock-free in the first row. "TS(>)%" is percentage of tasksets with a higher DSR under each contention manager in the first column than contention managers and lock-free in the first row. "TS(=)%" is percentage of tasksets that has the same DSR under both synchronization techniques in the first column and first row. Blank cell in Table 9.2 indicate that synchronization technique is compared to itself, or the two synchronization techniques have already been compared.

2. More jobs meet their deadlines under OMLP and RNLP than any contention manager by 12.4% and 13.7% on average, respectively. OMLP uses group locking that protects

Table 9.2: DSR comparison within CMs and with lock-free

		FBLT	PNF	LCM	ECM	LF
CP-FBLT	Avg%	4.6	8.8	31.2	31.3	34.6
	TS(>)%	54.1	51.9	76.7	77.2	68.9
	TS(=)%	9.5	7.8	8.8	8.8	2.5
FBLT	Avg%		4.2	26.5	26.7	28.5
	TS(>)%		43.2	62.8	62.9	63.9
	TS(=)%		10.5	12.8	12.2	7.4
PNF	Avg%			22.4	22.5	32.4
	TS(>)%			68.4	69.1	61.5
	TS(=)%			9.4	10	8.2
LCM	Avg%				0.13	-2.7
	TS(>)%				38.6	55.7
	TS(=)%				28	13.9
ECM	Avg%					-5.3
	TS(>)%					49.2
	TS(=)%					16.4

all required objects in the same atomic section by the same resource. Current implementation of RNLP requires a priori knowledge of requested objects per each atomic section. Thus, OMLP and RNLP have the advantage of a priori knowledge of requested objects per each atomic section. Only PNF has the same advantage. But PNF induces a lot of overhead because it is a centralized contention manager. To examine effect of a-priori knowledge of required objects, we modified FBLT to FBLT-P. Under FBLT-P, each transactions accesses all required objects at the beginning of the transaction. Thus, each transaction knows a priori what objects are going to be accessed. Results show that DSR of FBLT-P increased over DSR of CP-FBLT by 19.5%. Additionally, atomic sections under OMLP and RNLP do not have to retry, nor to make decisions upon a conflict in each retry. Under OMLP and RNLP, tasks suspend after making requests for acquiring specific locks. After obtaining all required locks, atomic sections can proceed without abortion upon a conflict. Thus, locking protocols make a decision only once regarding which atomic section to proceed, whereas a transaction can invoke the contention manager many times even if the contention manager is going to make the same decision. DSR comparison between locking protocols and contention managers is summarized in Table 9.3. "Avg%" is the average increase of DSR of each locking protocol in first column to DSR of each contention manager in the first row. "TS(≥)%" is percentage of tasksets, under each contention manager, with DSR lower than DSR of locking protocol by at most "Avg%".

3. CP-FBLT shows the shortest Avg_RC among proposed contention managers. Avg_RC comparison between proposed contention managers, lock-free and locking protocols is

Table 9.3: DSR comparison between CMs and locking protocols

		FBLT-P	CP-FBLT	FBLT	PNF	LCM	ECM
OMLP	Avg%	14.6	33.2	37.8	42	64.4	64.5
	TS(≥)%	71	65.6	58.6	55.3	39	39
RNLP	Avg%	15.9	34.5	39	43.3	65.7	65.8
	TS(≥)%	71.9	66.2	58.8	55	38	38

summarized in Table 9.4. “TS(>)%” is percentage of tasksets, under each contention manager in first column of Table 9.4, that has shorter Avg_RC than each synchronization technique in first row of Table 9.4. “TS(=)%” is percentage of tasksets, under each contention manager in first column of Table 9.4, that has equal Avg_RC with each synchronization technique in first row of Table 9.4. Blank cell in Table 9.4 indicate that contention manager is compared to itself, or the two synchronization techniques have already been compared.

Table 9.4: Avg_RC comparison between CMs, locking protocols and lock-free

		FBLT	PNF	LCM	ECM	OMLP	RNLP	LF
CP-FBLT	TS(>)%	53.1	61.1	80.7	88.9	52.4	38	40.2
	TS(=)%	16.9	9.4	2.8	2.3	0.2	14.5	14.8
FBLT	TS(>)%		63.3	77.5	86	38.6	23.1	17.2
	TS(=)%		11.2	2.2	2.3	1.2	16.7	28.7
PNF	TS(>)%			67.5	75.3	16.5	9.2	18
	TS(=)%			1.9	2	0.3	5.8	13.9
LCM	TS(>)%				69.5	3.6	3.1	0.8
	TS(=)%				2.3	0.1	0.4	7.4
ECM	TS(>)%					2.7	2.6	0
	TS(=)%					0	0	7.4

Chapter 10

Qualitative Comparison Between STM, Locking Protocols and Lock-Free

We compared proposed contention managers against retry-loop lock-free [49] and locking protocols (i.e., Global OMLP [22, 29] and RNLP [149]) analytically in Chapters 4 to 7 and quantitatively in Chapter 9. In this Chapter, we compare the proposed contention managers against retry-loop lock-free and locking protocols qualitatively.

The rest of this Chapter is organized as follows, Section 10.1 compares compositionality for the three synchronization techniques. Section 10.2 compares priority inversion and its bonuds under the synchronization techniques. The ability to access multi-objects per each atomic section (i.e., nesting) is compared in Section 10.3. Convoying is compared in Section 10.4. Deadlocks and livelocks are compared in Section 10.5. Section 10.6 compares dependence of different synchronization techniques on the underlying platform and how this dependence affects implementation. Section 10.7 compares the amount of a priori knowledge required by each synchronization technique. Complexity to upper bounds retry cost, blocking time and response time for the three synchronization techniques is compared in Section 10.8. We conclude the Chapter in Section 10.10.

10.1 Compositionality

Compositionality means when an object in the system satisfies a specific property, then the system as a whole satisfies this property [73]. Compositionality is important because it allows building the system in a modular way. Each module (or component) distinguishes between its implementation and its interface. As long as the component is composable, then programmers need not know its implementation. Programmers can rely only on the properties provided by the interface to build a system that satisfies these properties.

Locking protocols (including OMLP and RNLP) are not composable. To access multiple objects under OMLP and RNLP, either these objects must be protected by a global lock as in OMLP, or each object exposes its own lock as in RNLP. Thus, it is up to the programmer to decide how to use locks to satisfy correctness of the system. Lock-free objects are not composable as each object has different design alternatives. Thus, programmers must know implementation details of these alternatives to ensure correctness of the system. On contrast to locking and lock-free, STM allows compositionality as all objects are accessed in one transaction. If there is a conflict on any object inside the atomic block, the whole transaction restarts. Compositionality for different synchronization techniques is summarized in Table 10.1.

Table 10.1: Compositionality comparison

	Locking	STM	Lock-free
Compositionality	No	Yes	No

10.2 Priority Inversion

Priority inversion occurs when a higher priority job is not allowed to run because it needs a resource locked by a lower priority job [32, 101]. Priority inversion can be sometimes useful to prevent starvation of lower priority jobs. On the other hand, response times of higher priority jobs are elongated. Thus, priority inversion must be bounded.

Locking protocols bound priority inversion using priority inheritance [32,51,130] and priority ceiling [32, 38, 51, 89, 96, 118, 119, 130]. Global OMLP [22, 29] and RNLP [149] use priority inheritance. Under retry-loop lock-free, there is no guarantee that a higher priority task will get access to a required object before a conflicting lower priority task. Thus, priority inversion can occur under lock-free synchronization. Due to nature of hard real-time systems, a higher priority task can conflict with a bounded number of lower priority tasks. Thus, priority inversion under lock-free is bounded. Usually under STM, priority inversion is simply avoided by having the higher priority task aborting the lower priority one (i.e., ECM and RCM). According to design of contention manager, priority inversion is bounded as given by Claims 29, 52 and 66 for LCM, PNF and FBLT. As CP-FBLT is based on FBLT, CP-FBLT also bounds priority inversion. Priority inversion for different synchronization techniques is summarized in Table 10.2.

Table 10.2: Priority inversion comparison

	Locking	STM	Lock-free
Priority inversion	Bounded	Avoided or bounded	Bounded

10.3 Nesting

Nesting means the ability to access multiple objects in the same atomic section individually [149]. Nesting is important as it allows the design of fine-grained objects and increases concurrency (i.e., synchronization technique accesses only the needed parts of the object, not the whole object).

Lock-free does not allow nesting as lock-free primitives access only one object. OMLP does not allow nesting because all objects that are accessed in the same atomic section are protected by the same lock. RNLP and STM allow nesting. STM not only allows access of multiple object per transaction, but also allows nesting of transactions inside each other. Transactions can be nested *linearly*, where each transaction has at most one pending transaction [114]. Nesting can also be done in *parallel*, where transactions execute concurrently within the same parent [147]. Linear nesting can be *1) flat:* If a child transaction aborts, then the parent transaction also aborts. If a child commits, no effect is taken until the parent commits. Modifications made by the child transaction are only visible to the parent until the parent commits, after which they are externally visible. *2) Closed [88]:* Similar to *flat nesting*, except that if a child transaction conflicts, it is aborted and retried, without aborting the parent, potentially improving concurrency over flat nesting. *3) Open [87]:* If a child transaction commits, its modifications are immediately externally visible, releasing memory isolation of objects used by the child, thereby potentially improving concurrency over closed nesting. However, if the parent conflicts after the child commits, then compensating actions are executed to undo the actions of the child, before retrying the parent and the child. Nesting for different synchronization techniques is summarized in Table 10.3

Table 10.3: Nesting comparison

	Locking	STM	Lock-free
Nesting	OMLP: No	Yes	No
	RNLP: Yes		

10.4 Convoying

Convoying occurs when descheduling a task holding a lock [73]. Other tasks waiting for the lock are queued, waiting for the lock and unable to progress. Thus, convoying avoidance is important to ensure progress of the system.

Global OMLP and RNLP avoid lock convoying by priority inheritance. STM and lock-free objects do not suffer from convoying as they do not hold locks. Besides, all objects accessed by a preempted transaction under STM are available to all other transactions. Convoying for different synchronization techniques is summarized in Table 10.4

Table 10.4: Convoying comparison

	Locking	STM	Lock-free
Convoying	No	No	No

10.5 Deadlock and Livelock

Deadlock occurs if two or more tasks holding objects needed by each other. Each task is waiting for the others to finish and no task releases its objects [31, 73]. Thus, no task progresses.

For multiple objects, locking protocols avoid deadlock by accessing objects in order as assumed in RNLP. It is programmer's responsibility to impose order on objects. If requested objects in each critical section are known a priori, then objects can be requested at once. Thus, no need for objects' order. OMLP avoids deadlocks by group locking. Under group locking, all objects accessed in the same critical section are protected by the same lock. Lock-free objects use atomic primitives that access only one object. So, there is no chance for deadlock under lock-free objects. Deadlocks cannot occur under STM because contention manager allows only one transaction to proceed and aborts the others. Thus, objects can be accessed in any order under STM. Livelocks [73] are similar to deadlocks, except that tasks are changing their status with regard to each other with no progress (i.e., each task releases its objects for the other task). By definition of OMLP, RNLP, STM and lock-free, livelocks do not occur. Deadlock and livelock for different synchronization techniques are summarized in Table 10.5

Table 10.5: Deadlock/Livelock comparison

<table>
<tr><th></th><th>Locking</th><th>STM</th><th>Lock-free</th></tr>
<tr><td rowspan="2">Deadlock</td><td>OMLP: No</td><td rowspan="2">No</td><td rowspan="2">No</td></tr>
<tr><td>RNLP: Programmer dependent</td></tr>
<tr><td>Livelock</td><td>No</td><td>No</td><td>No</td></tr>
</table>

10.6 Platform Dependence and Implementation Complexity

By "Platform Dependence", we mean how much a specific synchronization technique is related to the underlying system (e.g., operating system, virtual machine and scheduler). Platform dependence is important for synchronization technique portability. It is more useful for legacy applications to use libraries than to rebuild the whole system. "Implementation Complexity" is related to "Platform Dependence". If the implementation requires detailed knowledge of the underlying platform (e.g., data structures and schedulers' details), then implementation becomes complex. If implementations requires moderate knowledge of the underlying platform (e.g., APIs), then implementation is at most of medium complexity.

STM, in general, is built as user-space library with a specific programming language (e.g., RSTM [144],TinySTM [122] and HyflowCPP [146] for C and C++. Deuce [90] and HyFlow [145] for Java). Using products, like Jini [52], different languages can communicate with each other. Thus, STM libraries can be independent on programming languages. Implementation of proposed contention managers was simply done by addition of header files to RSTM [144]. Proposed contention managers do not require modification of ChronOS. In contrast to STM, OMLP and RNLP are implemented inside kernel of ChronOS. OMLP and RNLP used "FIFO" and "Priority" queues inside kernel to organize requests by tasks to different objects. ChronOS structures and schedulers have been modified to enable suspension of tasks requesting locks, non-preemptive execution of tasks holding locks and priority inheritance. Thus, OMLP and RNLP are very dependent on ChronOS. Retry-loop lock-free uses atomic primitives such as CAS operations. Thus, retry-loop lock-free is independent on the underlying system. So, implementation of OMLP and RNLP is the most complex among synchronization techniques because OMLP and RNLP are tailored inside kernel. Implementations of proposed contention managers and lock-free are less complicated than OMLP and RNLP. Platform dependence and implementation complexity for different synchronization techniques are summarized in Table 10.6

Table 10.6: Platform dependence/Implementation complexity comparison

	Locking	STM	Lock-free
Platform dependence	Dependent	Independent	Independent
Implementation complexity	Hard	Simple	Simple

10.7 Transparency

By transparency we mean how much information about real-time tasks and shared objects must be provided a priori by programmer to the synchronization technique [31]. With less information, the synchronization technique becomes more flexible and more applicable to commercial applications.

OMLP must know a priori all objects accessed by each atomic section in each task. This information is important to form groups of objects and assignment of a distinct lock to each group (i.e., group locking). Despite a priori knowledge of required object per each atomic section is beneficial to RNLP, this information is not necessary for RNLP to work. Nevertheless, if objects are not know a priori for RNLP, objects must be accessed in order to prevent deadlock. Imposing order on object access is a programmer responsibility. Except for PNF (Chapter 6), all proposed contention managers require no previous knowledge about required object per each transaction. In contrast to OMLP, PNF requires knowledge only about required object for the current transaction when transaction starts. OMLP requires a priori knowledge about all objects accessed by each atomic section in all tasks to compute group locks. Retry-loop lock-free does not need any information about any objects. Thus, retry-loop lock-free is the most transparent synchronization technique. Transparency for different synchronization techniques is summarized in Table 10.7.

Table 10.7: Transparency comparison

<table>
<tr><th></th><th>Locking</th><th>STM</th><th>Lock-free</th></tr>
<tr><td rowspan="2">Transparency</td><td>OMLP: No</td><td rowspan="2">Yes</td><td rowspan="2">Yes</td></tr>
<tr><td>RNLP: Yes</td></tr>
</table>

10.8 Upper Bounds Complexity

By “Upper Bound Complexity” we mean how long does it take to calculate theoretical upper bounds over retry cost, blocking and response time under different synchronization techniques. Theoretical upper bounds are used to determine feasibility of a given taskset under a specific synchronization technique. As complexity of upper bounds’ calculation decreases, it will be faster to determine feasibility of the taskset. Schedulers can use the theoretical upper bounds to test effect of addition of new tasks (i.e., whether the addition of a new task will render the taskset to be infeasible). Complexity of upper bounds’ calculation depends on amount of required information about real-time tasks and accessed objects. To get tighter upper bounds, more information is needed which increases complexity.

As given in Chapters 4 to 7, upper bounds on retry cost and response time depend on accessed objects by each transaction, number and lengths of transactions per each task, periods and deadlines of each job, number of tasks and number of processors. Eq(4.38) upper bounds blocking time under OMLP. Eq(4.38) is given by Lemma 15 in [29] and tightened by Theorem 4 in [29]. Upper bound of blocking time under RNLP, given in [149], can be extended by the same analysis in [29]. Thus, OMLP and RNLP will need the same information required by proposed contention managers to calculate upper bounds over blocking time. Despite retry-loop lock access only one object for each atomic primitive, retry-loop lock-free still needs the same information by proposed contention managers to upper bound retry cost and response time. Thus, the three synchronization techniques have equal complexity to calculate upper bounds for retry cost, blocking and response time. Upper bound complexity for different synchronization techniques is summarized in Table 10.8.

Table 10.8: Upper bound complexity comparison

	Locking	STM	Lock-free
Upper bound complexity	Equal	Equal	Equal

10.9 Memory

We compare STM against lock-free and locking protocols in terms of required memory space and memory management. Without reasonable maximum amount of memory space, and without lightweight memory management, performance of STM can degrade. Proposed contention managers are integrated into object-based non-blocking RSTM [144]. As described in [110], each writer makes its own copy of the object data while keeping a pointer to the old object data. Writer makes modifications to its copy of the object data. Object header has a pointer to the most recent transaction that acquired the object, known as *Owner*. Owner has a pointer to the old object data. Upon commit, the new object data becomes the current object data. Upon abort, the old object data is the current object data. When a writer acquires an object, it invokes the contention manager to resolve any conflicts. Thus, object copies of aborted transactions become obsolete and marked as “retired” to be reclaimed later. Readers do not have to make copies of the object data. Instead each object has a list of pointers to visible readers. So, for n tasks and N_r objects, the maximum required memory space occurs when all the n tasks are executing transactions concurrently and each transaction tries to modify each of the N_r objects. At any time instant before reclaiming obsolete object data copies of aborted transactions, there can be at most $n.N_r$ copies of the objects. Proposed contention managers do not add any memory requirements except for PNF. PNF requires at most 3 lists: 1) A list of pointers to non-preemptive transactions. 2) A list of pointers to retrying transactions. 3) A list of pointers to used objects by non-preemptive transactions.

Similar to STM, retry-loop lock-free [49] needs at least a copy of the old data as a verification upon update. In contrast to STM, retry-loop lock-free uses atomic operations that need a copy of a single word, not the whole object. To compare STM and lock-free in terms of memory space, it is assumed that each object consumes a single word and each transaction accesses only one object. Thus, for n concurrent transactions released by n tasks, both STM and lock-free need n copies of objects at most. STM will need additional space for metadata (e.g., owner and descriptor fields). Besides, lock-free does not need memory management to reclaim obsolete data by aborted transactions.

In OMLP [22, 29] under G-EDF, each lock has two queues. A FIFO queue of m size and a priority queue of $max(n-m,0)$ size. Thus, the required space for each lock is $max(n,m)$. OMLP uses "group locking" that collects all objects that can be accessed with the same atomic section into the same group. Each group is protected by one lock. Thus, maximum memory requirements occurs when each object belongs to a distinct group. In this case, there are N_r locks for N_r objects. Consequently, OMLP needs $N_r.max(n,m)$ slots for all queues. Each slot contains a pointer to a requesting task. Thus, if $n > m$, then OMLP needs $n.N_r$ slots for pointers, whereas STM needs $n.N_r$ copies of objects. Thus, OMLP needs equal or less space than STM. If $n \leq m$, then STM may need less space than OMLP if the size for any object does not exceed m/n multiplied by the size for any pointer. In contrast to STM, OMLP does not need memory management to reclaim obsolete data. OMLP does not record metadata as in STM.

In RNLP [148–150] under G-EDF, each object has a queue of length m. Despite objects can be grouped into "dynamic groups" as presented by [148], the maximum space requirement occurs when each object is accessed individually. Thus, each object is protected by a separate lock. For any task to request an object, it must first obtain a token. The token queue is a FIFO queue of length m. Additional tasks requesting tokens are organized in a priority queue of length $max(n-m,0)$. Consequently, RNLP needs $m.N_r + max(n,m)$ slots for all queues. Each slot contains a pointer to a requesting task. If $m \geq n$, then RNLP needs $m(N_r+1)$ slots for pointers, whereas STM needs $n.N_r$ copies of objects. Thus, STM may require less space than RNLP if the size for any object does not exceed $\frac{m(N_r+1)}{n.N_r}$ multiplied by size of any pointer. Otherwise, if $n < m$, then STM may require less space than RNLP if the size for any object does not exceed $\frac{m.N_r+n}{n.N_r}$ multiplied by size of any pointer. In contrast to STM, RNLP does not need memory management to reclaim obsolete data. RNLP does not record metadata as in STM. Memory comparison between different synchronization techniques is summarized in Table 10.9

Table 10.9: Memory comparison between locking, STM and lock-free

		Locking	STM	Lock-free
Memory	Space compared to STM	Dependent	-	Less
	Memory management	Low	High	Low

10.10 Conclusion

We compared proposed contention managers against retry-loop lock-free [49] and locking protocols (i.e., Global OMLP [22, 29] and RNLP [149]) analytically in Chapters 4 to 7 and quantitatively in Chapter 9. In this Chapter, we compared the proposed contention managers against retry-loop lock-free and locking protocols qualitatively. The complete qualitative comparison between the three synchronization techniques is give in Table 10.10.

Table 10.10: Qualitative comparison between locking, STM and lock-free

<table>
<tr><th colspan="2"></th><th>Locking</th><th>STM</th><th>Lock-free</th></tr>
<tr><td colspan="2">Compositionality</td><td>No</td><td>Yes</td><td>No</td></tr>
<tr><td colspan="2">Priority inversion</td><td>Bounded</td><td>Avoided or bounded</td><td>Bounded</td></tr>
<tr><td colspan="2" rowspan="2">Nesting</td><td>OMLP: No</td><td rowspan="2">Yes</td><td rowspan="2">No</td></tr>
<tr><td>RNLP: Yes</td></tr>
<tr><td colspan="2">Convoying</td><td>No</td><td>No</td><td>No</td></tr>
<tr><td colspan="2" rowspan="2">Deadlock</td><td>OMLP: No</td><td rowspan="2">No</td><td rowspan="2">No</td></tr>
<tr><td>RNLP: Programmer dependent</td></tr>
<tr><td colspan="2">Livelock</td><td>No</td><td>No</td><td>No</td></tr>
<tr><td colspan="2">Platform dependence</td><td>Dependent</td><td>Independent</td><td>Independent</td></tr>
<tr><td colspan="2">Implementation complexity</td><td>Hard</td><td>Simple</td><td>Hard</td></tr>
<tr><td colspan="2" rowspan="2">Transparency</td><td>OMLP: No</td><td rowspan="2">Yes</td><td rowspan="2">Yes</td></tr>
<tr><td>RNLP: Yes</td></tr>
<tr><td colspan="2">Upper bound complexity</td><td>Equal</td><td>Equal</td><td>Equal</td></tr>
<tr><td rowspan="2">Memory</td><td>Space compared to STM</td><td>Dependent</td><td>-</td><td>Less</td></tr>
<tr><td>Memory management</td><td>Low</td><td>High</td><td>Low</td></tr>
</table>

Chapter 11

Conclusions and Future Work

11.1 Conclusions

In this dissertation, we designed, analyzed, and experimentally evaluated six real-time CMs. Designing real-time CMs is straightforward. The simplest logic is to use the same rationale as that of the underlying real-time scheduler. This was shown in the design of ECM and RCM. ECM allows the transaction with the earliest absolute deadline (i.e., dynamic priority) to commit first. RCM allows the transaction with the smallest period (i.e., fixed priority) to commit first. We established upper bounds for retry costs and response times under ECM and RCM, and identified sufficient conditions under which they have equal or better performance than lock-free and locking protocols in terms of total utilization.

As each transaction can access multiple objects, a transaction may abort indirectly due to another transaction with no shared objects between them. The indirect retrial is denoted as transitive retry. Under both ECM and RCM, a task incurs at most $2s_{max}$ retry cost for each of its atomic sections due to a conflict with another task's atomic section. Transactions can also retry due to release of higher priority jobs that preempt a transaction in a lower priority job.

The s_{max}/r_{max} ratio is a sufficient condition to determine whether STM is better or as good as lock-free. ECM and RCM have equal or better total utilization than retry-loop lock-free if s_{max} does not exceed one half of r_{max}. s_{max} can exceed r_{max} with equal periods between conflicting tasks, and large access times to the same object within the same transaction. Performance of ECM and RCM were compared against real-time locking protocols (i.e., OMLP and RNLP) in terms of total utilization. As number of atomic sections in each task increases, all tasks have equal periods and equal number of atomic sections, and number of processors is at least equal to number of tasks, then s_{max} can be at least equal to L_{max} with total utilization of ECM equal or better than total utilization of OMLP and RNLP. The same results apply to total utilization comparison between RCM and locking protocols except that number of processors should be at least double number of tasks.

In ECM and RCM, a task incurs at most $2s_{max}$ retry cost for each of its atomic section due to conflict with another task's atomic section. With LCM, this retry cost is reduced to $(1+\alpha_{max})s_{max}$ for each aborted atomic section. In ECM and RCM, higher priority tasks are not blocked due to lower priority tasks, whereas in LCM, they are. In LCM/G-EDF, blocking due to a lower priority job is encountered only from a task τ_j's last job instance during τ_i's period. Contribution of a transaction s_j^l to the retry cost of a lower priority transaction is higher than blocking caused by s_j^l to a higher priority transaction. Thus, under LCM/G-EDF, each transaction is assumed to contribute in the abort and retry of a lower priority transaction. Hence, blocking of higher priority transactions due to lower priority transactions is ignored under LCM/G-EDF. This is not the case with LCM/G-RMA, because of fixed priority under G-RMA. Blocking time under LCM is bounded.

Total utilization of LCM/G-EDF is always equal or better than ECM's. Whereas, total utilization of LCM/G-RMA is equal or better than RCM's depending on α_{min} and α_{max}. Total utilization of LCM (with G-EDF and G-RMA) is equal or better than total utilization of retry-loop lock-free if s_{max} does not exceed $r_{max}/\left(1+\alpha_{max}\right)$. With high number of object access within each transaction, s_{max} can be much larger than r_{max} with equal or better total utilization for LCM (with G-EDF and G-RMA) than total utilization of retry-loop lock-free. Total utilization of LCM was compared against real-time locking protocols (i.e., OMLP and RNLP) under G-EDF and G-RMA. As number of atomic sections in each task increases, all tasks have equal periods and equal number of atomic sections, α_{max} approaches 0, and number of processors is at least equal to half number of tasks, then s_{max} can be at least equal to L_{max} with total utilization of LCM/G-EDF equal or better than total utilization of OMLP and RNLP. The same results apply to total utilization comparison between LCM/G-RMA and locking protocols except that α_{max} approaches α_{min} and number of processors should be at least double number of tasks.

Transitive retry increases transactional retry cost under ECM, RCM, and LCM. PNF avoids transitive retry by avoiding transactional preemptions. PNF reduces the priority of aborted transactions to enable other tasks to execute, increasing processor usage. Executing transactions are not preempted due to the release of higher priority jobs. On the negative side of PNF, higher priority jobs can be blocked by executing transactions of lower priority jobs. PNF/G-EDF's total utilization is equal or better than ECM's if, for each task τ_i, total number of transactions in any task $\tau_j \neq \tau_i$ - that has no direct conflict with any transaction in τ_i - divided by number of processors is not greater than maximum number of higher priority jobs than current job of τ_i that can be released during T_i. Similar condition holds for the total utilization comparison between PNF/G-EDF and LCM/G-EDF, in addition to maintain a lower bound of $1/\alpha_{max}$ over maximum number of higher priority jobs of τ_j that can exist during T_i and have direct conflict with any transaction in τ_i. Total utilization of PNF/G-RMA is equal or better than RCM's if, for each task τ_i, total number of transactions in tasks with lower priority than p_i does not exceed one half of maximum number of jobs with higher priority than p_i that can be released during T_i. Total utilization of PNF/G-RMA is equal or better than LCM/G-RMA's if $\alpha_{min} \rightarrow 0$ and, for each task τ_i, total number of transactions in tasks with lower priority than p_i and have no direct conflict with any transaction in τ_i divided by number of processors does not exceed one half of maximum number of jobs with higher priority than p_i that can be released during T_i. Total utilization of PNF under G-EDF and G-RMA is equal or better than total utilization of retry-loop lock-free [49] with $s_{max}/r_{max} \geq 1$ if, for each task τ_i, maximum number of higher priority jobs than current job of τ_i - that can be released during T_i - is not less than maximum number of lower priority transactions in any task $\tau_j \neq \tau_i$ that has no direct conflict with any transaction in τ_i. Total utilization of PNF was compared against real-time locking protocols (i.e., OMLP and RNLP) under G-EDF and G-RMA. As all tasks have equal periods and equal number of atomic sections, and number of processors exceeds number of tasks, then s_{max} can be at least equal to L_{max} with total utilization of PNF equal or better than total utilization of OMLP and RNLP under G-EDF and G-RMA.

PNF requires a priori knowledge about objects accessed by each transaction. Besides, PNF is a centralized CM. This is incompatible with dynamic STM implementations. Thus, we introduce the FBLT contention manager. Under FBLT, each transaction is allowed to abort for no larger than a specified number of times. Afterwards, the transaction becomes non-preemptive. Non-preemptive transactions have higher priorities than other preemptive transactions and real-time jobs. Non-preemptive transactions resolve their conflicts according to the order they become non-preemptive (i.e., FBLT aborts the later non-preemptive transaction in favour of the earlier non-preemptive transaction).

By proper adjustment of the maximum abort number for any preemptive transaction of any task τ_i (i.e., Ω_i^{max}), FBLT's total utilization is equal to or better than total utilization of other CMs. Ratio between s_{max} for FBLT on one side and r_{max} for lock-free and L_{max} for locking protocols on the other side also depends on Ω_i^{max}. As Ω_i^{max} decreases, s_{max}/r_{max} and s_{max}/L_{max} increase. As number of atomic sections in each task increases, all tasks have equal number of atomic sections, and number of processors is not less than maximum Ω_i^{max} for any τ_i, then s_{max} can be at least equal to L_{max} with total utilization of FBLT equal or better than total utilization of OMLP and RNLP under G-EDF and G-RMA. In any case, s_{max} should not exceed $2.L_{max}$.

Past research on real-time CMs focused on developing different conflict resolution strategies for transactions. Except for LCM (Chapter 5), no policy was made to reduce the length of conflicting transactions. We analysed effect of checkpointing over FBLT CM and identified conditions under which CP-FBLT can have reduced response time than FBLT. Some CMs make no use of checkpointing due to behaviour of that CM (e.g, under PNF, all non-preemptive transactions are non-conflicting).

We also implemented previous CMs, lock-free, OMLP and RNLP locking protocols in ChronOS real-time operating system and conducted experimental studies. We compared different synchronization techniques in term of Deadline Satsifaction Ratio (DSR) and Average Retry Cost (Avg_RC). Our experimental studies revealed, among the contention managers, CP-

FBLT performs the best. DSR for CP-FBLT exceeds DSR of ECM, LCM, PNF and FBLT by 31.3%, 31.2%, 8.8% and 4.6% on average, respectively. Percentages of tasksets, under CP-FBLT, that have DSR at least equal to DSR of ECM, LCM, PNF and FBLT are 86%, 85.5%, 59.6% and 63.6% on average, respectively. CP-FBLT's higher performance is due to the fact that PNF's and LCM's advantages are combined into the design of FBLT, which is the base of CP-FBLT. Moreover, checkpointing improves task response time.

Contention managers show equal or better performance than lock-free: More jobs meet their deadlines using CP-FBLT, FBLT and PNF than lock-free synchronization by 34.5%, 28.4% and 32.4% (on average), respectively. Average percentage of jobs that meet their deadlines using ECM and LCM are slightly lower than lock-free by 5.2% and 2.6%, respectively. Superiority of contention managers to lock-free results from conflict resolution policy of contention managers. STM allows access of multiple objects per transaction, while lock-free do not.

Generally, more jobs meet their deadlines under OMLP and RNLP than any contention manager by 12.4% and 13.7% on average, respectively. 66% of tasksets, under CP-FBLT, have lower DSR than OMLP and RNLP by at most 33.5% on average. 58.7% of tasksets, under FBLT, have lower DSR than OMLP and RNLP by at most 38% on average. 55% of tasksets, under PNF, have lower DSR than OMLP and RNLP by at most 43% on average. 38.5% of tasksets, under each of ECM and LCM, have lower DSR than OMLP and RNLP by at most 65% on average. Higher DSR of OMLP and RNLP results from priori knowledge of required object per each critical section. Thus, priority inversion is reduced. Only PNF has the same advantage. But PNF induces a lot of overhead because it is a centralized contention manager. Additionally, atomic sections under OMLP and RNLP do not have to retry, nor to make decisions upon a conflict in each retry. Whereas a contention manager has to be invoked on each conflict, even if it will make the same decision. However, the contention managers have numerous *qualitative* advantages over locking protocols. Locks do not compose, whereas STM transactions do. Support for nested critical sections is generally complicated for locking protocols, whereas it is trivial with STM. To allow multiple objects to be accessed in a critical section, OMLP assigns objects to non-conflicting groups, where each group is protected by a distinct lock. RNLP assumes that objects are accessed in a specific order to prevent deadlocks. In contrast, STM allows multiple objects to be accessed in a transaction in any order, while guaranteeing deadlock-freedom, which significantly increases programmability. From a systems programmer's perspective, OMLP and RNLP are relatively difficult to implement, whereas proposed contention managers are easy to implement. From an application programmer's perspective, OMLP is not transparent as it requires the description of additional information (i.e., what objects will be needed in each critical section). For RNLP to avoid order on object access, RNLP needs to know required objects for each critical section a priori. In contrast, no such extra information is needed for using proposed contention managers (except for PNF), which significantly increases programmability. STM offers platform independence: the proposed contention managers can be entirely implemented in the user-space as a library. In contrast, OMLP and RNLP must be supported by the underlying platform (i.e., operating system or virtual machine).

11.2 Future Work

Based on dissertation's results, we propose the following directions for future research.

1. One way to imitate suspension behaviour of locking protocols is to delay transactions upon conflict instead of abortion. Delaying transactions is implemented in contention managers such as Kindergarten, Timestamp, Randomized, Greedy, Polka [67–69, 126, 127, 135]. A transaction waits for different amount of times according to contention manager. To delay a transaction under proposed contention managers for real-time systems, delay time should not increase response time nor average retry cost compared to contention mangers with no delay of transactions. Analysis of upper bounds over average retry cost, response time and schedulability should take delay time into account.

2. Results showed that CP-FBLT achieved higher DSR than other CMs. Thus, one direction for future research is to extend all CMs with chechpointing and checkpointing scheduling policy. Current implementation of CP-FBLT records a new checkpoint for each newly accessed object. Checkpointing scheduling looks for the best location of checkpoints to reduce retry cost and improve DSR. Also, checkpointing scheduling must take into account the overhead of creation and removal of checkpoints.

3. PNF, FBLT and CP-FBLT showed mixed DSR patterns depending on different parameters (e.g., total utilization, number of objects, number and length of transactions in each task). It will be useful to develop a hybrid CM that adaptively chooses the suitable CM depending on these parameters. The developed CM should be able to adapt new values for the underlying parameters of the CM (e.g., Ω_i^k in FBLT and Ψ in LCM) to reduce retry cost and increase DSR. If transactions can wait as given in point 1, then the developed CM should be able to determine the proper delay value for each transaction according to situation in hand.

4. Extending proposed CMs into soft real-time systems. Under soft real-time systems, tasks can miss their deadlines within a bounded time interval. Depending on the current dissertation's results, soft real-time systems can make good use of STM CMs.

5. Development of new CMs to further reduce retry cost and increase DSR. Extending experiments' scale to total utilization higher than 8. Development of standard benchmarks devoted to test effect of synchronization techniques on hard and/or soft real-time systems.

Bibliography

[1] Advanced Micro Devices, Inc. Advanced Synchronization Facility – Proposed Architectural Specification, 2.1 edition. Available `http://developer.amd.com/assets/45432-ASF_Spec_2.1.pdf`, 2009.

[2] S. Acharya and R.N. Mahapatra. A dynamic slack management technique for real-time distributed embedded systems. *Computers, IEEE Transactions on*, 57(2):215–230, 2008.

[3] S. Agrawal, R.S. Yadav, and N. Das. Checkpointing based fault tolerance patterns for systems with arbitrary deadlines. In *Advanced Computing and Communications, 2007. ADCOM 2007. International Conference on*, pages 694–699, 2007.

[4] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie. Unbounded transactional memory. In *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, pages 316 – 327, feb. 2005.

[5] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. volume 15, pages 134–165. ACM, May 1997.

[6] J.H. Anderson and P. Holman. Efficient pure-buffer algorithms for real-time systems. In *Proceedings of Seventh International Conference on Real-Time Computing Systems and Applications*, pages 57 –64, 2000.

[7] J.H. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *17th IEEE Real-Time Systems Symposium*, pages 94 –105, dec 1996.

[8] J.H. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. Lock-free transactions for real-time systems. In *Real-Time Databases: Issues and Applications*, pages 215–234. Kluwer, 1997.

[9] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management as̨ a̧ non-clairvoyant scheduling problem. *Algorithmica*, 57:44–61, 2010. 10.1007/s00453-008-9195-x.

[10] Tian Bai, YunSheng Liu, and Yong Hu. Timestamp vector based optimistic concurrency control protocol for real-time databases. In *4th International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM)*, pages 1 –4, oct. 2008.

[11] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3:67–99, 1991.

[12] Theodore P. Baker. A comparison of global and partitioned edf schedulability tests for multiprocessors. Technical report, In International Conf. on Real-Time and Network Systems, 2005.

[13] A. Barros and L.M. Pinho. Managing contention of software transactional memory in real-time systems. In *IEEE RTSS, Work-In-Progress*, 2011.

[14] A. Barros and L.M. Pinho. Software transactional memory as a building block for parallel embedded real-time systems. In *37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 251 –255, 2011.

[15] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *RTSS*, pages 119–128, 2007.

[16] M. Behnam, F. Nemati, T. Nolte, and H. Grahn. Towards an efficient approach for resource sharing in real-time multiprocessor systems. In *6th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 99 –102, june 2011.

[17] C. Belwal and A.M.K. Cheng. Lazy versus eager conflict detection in software transactional memory: A real-time schedulability perspective. *IEEE Embedded Systems Letters*, 3(1):37 –41, march 2011.

[18] Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *RTSS*, pages 149–160, 2007.

[19] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Proactive transaction scheduling for contention management. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 156–167. ACM, 2009.

[20] A. Block, H. Leontyev, B.B. Brandenburg, and J.H. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 47 –56, aug. 2007.

[21] B.B. Brandenburg and J.H. Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 184 –193, july 2009.

[22] B.B. Brandenburg and J.H. Anderson. Optimality results for multiprocessor real-time locking. In *IEEE 31st Real-Time Systems Symposium (RTSS)*, pages 49 –60, 30 2010-dec. 3 2010.

[23] B.B. Brandenburg and J.H. Anderson. Real-time resource-sharing under clustered scheduling: mutex, reader-writer, and k-exclusion locks. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 69 –78, oct. 2011.

[24] B.B. Brandenburg, J.M. Calandrino, and J.H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Real-Time Systems Symposium*, pages 157 –169, 30 2008-dec. 3 2008.

[25] B.B. Brandenburg, J.M. Calandrino, A. Block, H. Leontyev, and J.H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 342 –353, april 2008.

[26] Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[27] Bjrn Brandenburg and James Anderson. A comparison of the m-pcp, d-pcp, and fmlp on litmus rt. In Theodore Baker, Alain Bui, and Sbastien Tixeuil, editors, *Principles of Distributed Systems*, volume 5401, pages 105–124, 2008.

[28] Bjrn Brandenburg and James Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46:25–87, 2010. 10.1007/s11241-010-9097-2.

[29] Bjrn Brandenburg and James Anderson. The omlp family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, pages 1–66, 2012.

[30] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In *Principles of Distributed Systems*, volume 7109, pages 207–221. Springer Berlin-Heidelberg, 2011.

[31] G.C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer-Verlag New York Inc, 2005.

[32] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems COMMENTseries)*. Springer-Verlag TELOS, 2004.

[33] J.M. Calandrino, J.H. Anderson, and D.P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *19th Euromicro Conference on Real-Time Systems. ECRTS '07.*, pages 247–258, 2007.

[34] J.M. Calandrino, H. Leontyev, A. Block, U.C. Devi, and J.H. Anderson. LITMUS-RT : A testbed for empirically comparing real-time multiprocessor schedulers. In *27th IEEE International Real-Time Systems Symposium. RTSS '06.*, pages 111–126, 2006.

[35] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238. IEEE Computer Society, 2006.

[36] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance SPARC CMT processor. *Micro, IEEE*, 29(2):6–16, 2009.

[37] Jing Chen. A loop-free asynchronous data sharing mechanism in multiprocessor real-time systems based on timing properties. In *Proceedings of 23rd International Conference on Distributed Computing Systems Workshops*, pages 184 – 190, May 2003.

[38] Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 2:325–346, 1990.

[39] Hyeonjoong Cho, B. Ravindran, and E.D. Jensen. Synchronization for an optimal real-time scheduling algorithm on multiprocessors. In *International Symposium on Industrial Embedded Systems (SIES)*, pages 9 –16, july 2007.

[40] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. On utility accrual processor scheduling with wait-free synchronization for embedded real-time software. In *Proceedings of the ACM symposium on Applied computing*, pages 918–922. ACM, 2006.

[41] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *27th IEEE International Real-Time Systems Symposium (RTSS)*, pages 101 –110, dec. 2006.

[42] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. Lock-free synchronization for dynamic embedded real-time systems. *ACM Trans. Embed. Comput. Syst.*, 9(3):23:1–23:28, March 2010.

[43] Hyeonjoong Cho, Binoy Ravindran, and E.D. Jensen. A space-optimal wait-free real-time synchronization protocol. In *Proceedings of 17th Euromicro Conference on Real-Time Systems*, pages 79 – 88, July 2005.

[44] D. Christie, J.W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, et al. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, pages 27–40. ACM, 2010.

[45] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346. ACM, 2006.

[46] A. Datta, S.H. Son, and V. Kumar. Is a bird in the hand worth more than two in the bush? limitations of priority cognizance in conflict resolution for firm real-time database systems. *IEEE Transactions on Computers*, 49(5):482 –502, may 2000.

[47] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[48] Matthew Dellinger, Piyush Garyali, and Binoy Ravindran. ChronOS Linux: a best-effort real-time multiprocessor linux kernel. In *Proceedings of the 48th DAC*, pages 474–479. ACM, 2011.

[49] U.M.C. Devi, H. Leontyev, and J.H. Anderson. Efficient synchronization under global edf scheduling on multiprocessors. In *18th Euromicro Conference on Real-Time Systems*, pages 10 pp. –84, 0-0 2006.

[50] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 125–134. ACM, 2008.

[51] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 377 –386, dec. 2009.

[52] W. Keith Edwards. *Core JINI*. Prentice Hall Professional Technical Reference, 2nd edition, 2000.

[53] Mohammed El-Shambakey and Binoy Ravindran. STM concurrency control for embedded real-time software with tighter time bounds. In *Proceedings of the 49th DAC*, pages 437–446. ACM, 2012.

[54] Mohammed El-Shambakey and Binoy Ravindran. STM concurrency control for multicore embedded real-time software: time bounds and tradeoffs. In *Proceedings of the 27th SAC*, pages 1602–1609. ACM, 2012.

[55] G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-gpu systems. *19th RTNS*, 2011.

[56] Mohammed Elshambakey and Binoy Ravindran. Fblt: A real-time contention manager with improved schedulability. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1325–1330, 2013.

[57] Mohammed Elshambakey and Binoy Ravindran. On real-time stm concurrency control for embedded software with improved schedulability. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 47–52, 2013.

[58] J.R. Engdahl and Dukki Chung. Lock-free data structure for multi-core processors. In *International Conference on Control Automation and Systems (ICCAS)*, pages 984 –989, oct. 2010.

[59] A. Ermedahl, H. Hansson, M. Papatriantafilou, and P. Tsigas. Wait-free snapshots in real-time systems: algorithms and performance. In *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, pages 257 –266, oct 1998.

[60] S. Fahmy and B. Ravindran. On STM concurrency control for multicore embedded real-time software. In *International Conference on Embedded Computer Systems (SAMOS)*, pages 1 –8, July 2011.

[61] S.F. Fahmy, B. Ravindran, and E. D. Jensen. On bounding response times under software transactional memory in distributed multiprocessor real-time systems. In *DATE*, pages 688–693, 2009.

[62] Y.M.P. Fernandes, A. Perkusich, P.F.R. Neto, and M.L.B. Perkusich. Implementation of transactions scheduling for real-time database management. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 6, pages 5136 – 5141 vol.6, oct. 2004.

[63] K. Fraser. *Practical lock-freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

[64] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 189 – 198, may 2003.

[65] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of 22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 73 – 83, dec. 2001.

[66] J. Gottschlich and D.A. Connors. Extending contention managers for user-defined priority-based transactions. In *Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM), Boston, MA*. Citeseer, 2008.

[67] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In Pierre Fraigniaud, editor, *Distributed Computing*, volume 3724, pages 303–323. 2005.

[68] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC*, pages 258–264, 2005.

[69] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Towards a theory of transactional contention managers. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 316–317. ACM, 2006.

[70] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, pages 102–. IEEE Computer Society, 2004.

[71] M. Herlihy, Y. Lev, and N. Shavit. A lock-free concurrent skiplist with wait-free search. In *Unpublished Manuscript*. Sun Microsystems Laboratories, Burlington, Massachusetts, 2007.

[72] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[73] Maurice Herlihy. The art of multiprocessor programming. In *PODC*, 2006.

[74] Maurice Herlihy et al. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd PODC*, pages 92–101. ACM, 2003.

[75] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[76] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 289–300. ACM, 1993.

[77] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch Hashing. In Gadi Taubenfeld, editor, *Distributed Computing*, volume 5218, pages 350–364. Springer Berlin / Heidelberg, 2008.

[78] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 82–91. ACM, 2006.

[79] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, 2001.

[80] P. Holman and J.H. Anderson. Locking in pfair-scheduled multiprocessor systems. In *23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 149 – 158, 2002.

[81] P. Holman and J.H. Anderson. Supporting lock-free synchronization in Pfair-scheduled real-time systems. *Journal of Parallel and Distributed Computing*, 66(1):47–67, 2006.

[82] Philip L. Holman. *On the implementation of pfair-scheduled multiprocessor systems*. PhD thesis, University of North Carolina, Chapel Hill, 2004.

[83] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M. `http://www.intel.com/Assets/en_US/PDF/manual/253666.pdf`, 2007.

[84] Intel Corporation. Intel Itanium Architecture Software Developers Manual Volume 3: Instruction Set Reference. `http://download.intel.com/design/Itanium/manuals/24531905.pdf`, 2007.

[85] Intel Corporation. Intel C++ STM Compiler 4.0, Prototype Edition. `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`, 2009.

[86] J. Reinders. Transactional synchronization in Haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, 2013.

[87] Junwhan Kim, Roberto Palmieri, and Binoy Ravindran. Scheduling open-nested transactions in distributed transactional memory. In Rocco Nicola and Christine Julien, editors, *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 105–120. Springer Berlin Heidelberg, 2013.

[88] Junwhan Kim and B. Ravindran. Scheduling closed-nested transactions in distributed transactional memory. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, pages 179 –188, may 2012.

[89] D.K. Kiss. Intelligent priority ceiling protocol for scheduling. In *2011 3rd IEEE International Symposium on Logistics and Industrial Informatics*, pages 105 –110, aug. 2011.

[90] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *MULTIPROG*, 2010.

[91] Eric Koskinen and Maurice Herlihy. Checkpoints and continuations instead of nested transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 160–168, New York, NY, USA, 2008. ACM.

[92] Tei-Wei Kuo and Hsin-Chia Hsih. Concurrency control in a multiprocessor real-time database system. In *12th Euromicro Conference on Real-Time Systems (Euromicro RTS)*, pages 55 –62, 2000.

[93] Seong-Woo Kwak, Byung-Jae Choi, and Byung-Kook Kim. Checkpointing strategy for multiple real-time tasks. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 517–521, 2000.

[94] S.W. Kwak and J. M Yang. Schedulability and optimal checkpoint placement for real-time multi-tasks. In *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 778–782, 2010.

[95] Shouwen Lai, Binoy Ravindran, and Hyeonjoong Cho. On scheduling soft real-time tasks with lock-free synchronization for embedded devices. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1685–1686. ACM, 2009.

[96] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 469 –478, dec. 2009.

[97] Kam-Yiu Lam, Tei-Wei Kuo, and Wai-Hung Tsang. Concurrency control for real-time database systems with mixed transactions. In *Proceedings of Fourth International Workshop on Real-Time Computing Systems and Applications*, pages 96 –103, oct 1997.

[98] C.P.M. Lau and V.C.S. Lee. Real time concurrency control for data intensive applications. In *Proceedings of 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 337 – 342, aug. 2005.

[99] M.R. Lehr, Young-Kuk Kim, and S.H. Son. Managing contention and timing constraints in a real-time database system. In *Proceedings of 16th IEEE Real-Time Systems Symposium*, pages 332 –341, dec 1995.

[100] Yossi Lev and Jan-Willem Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 197–206. ACM, 2008.

[101] Gertrude Levine. Priority inversion with fungible resources. *Ada Lett.*, 31(2):9–14, February 2012.

[102] Guohui Li, Fangxiao Hu, and Ling Yuan. An energy-efficient fault-tolerant scheduling scheme for aperiodic tasks in embedded real-time systems. In *Multimedia and Ubiquitous Engineering, 2009. MUE '09. Third International Conference on*, pages 369–376, 2009.

[103] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, MIT, 2004.

[104] Aaron C Lindsay. *LWFG: A Cache-Aware Multi-core Real-Time Scheduling Algorithm.* PhD thesis, Virginia Polytechnic Institute and State University, 2012.

[105] A. Lotfi, A. Bayat, and S. Safari. Architectural vulnerability aware checkpoint placement in a multicore processor. In *On-Line Testing Symposium (IOLTS), 2012 IEEE 18th International*, pages 118–120, 2012.

[106] G. Macariu and V. Cretu. Limited blocking resource sharing for global multiprocessor scheduling. In *23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 262 –271, july 2011.

[107] W. Maldonado, P. Marlier, P. Felber, J. Lawall, G. Muller, and E. Riviere. Deadline-aware scheduling for software transactional memory. In *41st International Conference on Dependable Systems Networks (DSN)*, pages 257 –268, june 2011.

[108] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–90. ACM, 2010.

[109] J. Manson, J. Baker, et al. Preemptible atomic regions for real-time Java. In *RTSS*, pages 10–71, 2006.

[110] Virendra J Marathe, Michael F Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N Scherer III, and Michael L Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[111] Fadi Meawad, Martin Schoeberl, Karthik Iyer, and Jan Vitek. Real-time wait-free queues using micro-transactions. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 1–10. ACM, 2011.

[112] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[113] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: log-based transactional memory. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 254 – 265, feb. 2006.

[114] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186 – 201, 2006.

[115] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 251 –261, july 2011.

[116] Wu Peng and Pang Zilong. Research on the improvement of the concurrency control protocol for real-time transactions. In *International Conference on Machine Vision and Human-Machine Interface (MVHI)*, pages 146 –148, april 2010.

[117] Sathya Peri and Krishnamurthy Vidyasankar. Correctness of concurrent executions of closed nested transactions in transactional memory systems. In *Proceedings of the 12th international conference on Distributed computing and networking*, pages 95–106. Springer-Verlag, 2011.

[118] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS*, pages 116–123, 2002.

[119] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[120] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of 32nd International Symposium on Computer Architecture (ISCA)*, pages 494 – 505, june 2005.

[121] M. Raynal. Wait-free objects for real-time systems? In *Proceedings of Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 413 –420, 2002.

[122] T. Riegel, P. Felber, and C. Fetzer. TinySTM. `http://tmware.org/tinystm`, 2010.

[123] Mohamed M. Saad and Binoy Ravindran. Hyflow: a high performance distributed software transactional memory framework. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 265–266, New York, NY, USA, 2011. ACM.

[124] Bratin Saha, Ali-Reza Adl-Tabatabai, et al. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, pages 187–197, 2006.

[125] T. Sarni, A. Queudet, and P. Valduriez. Real-time support for software transactional memory. In *RTCSA*, pages 477–485, 2009.

[126] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM, 2005.

[127] W.N. Scherer III and M.L. Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java programs*, pages 70–79, 2004.

[128] M. Schoeberl, F. Brandner, and J. Vitek. RTTM: Real-time transactional memory. In *ACM SAC*, pages 326–333, 2010.

[129] M. Schoeberl and P. Hilber. Design and implementation of real-time transactional memory. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 279 –284, 31 2010-sept. 2 2010.

[130] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175 –1185, sep 1990.

[131] L. Sha, R. Rajkumar, S.H. Son, and C.-H. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793 –800, jul 1991.

[132] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[133] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 104–115. ACM, 2007.

[134] Richard L. Sites. Alpha AXP architecture. *Commun. ACM*, 36:33–44, February 1993.

[135] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 141–150. ACM, 2009.

[136] J.M. Stone, H.S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(4):58 –71, nov 1993.

[137] H. Sundell and P. Tsigas. Space efficient wait-free buffer sharing in multiprocessor real-time systems based on timing information. In *Proceedings of Seventh International Conference on Real-Time Computing Systems and Applications*, pages 433 –440, 2000.

[138] TM Specication Drafting Group. Draft specification of transactional language constructs for c++, version 1.1, 2012.

[139] Marc Tremblay. Transactional memory for a modern microprocessor. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 1–1, 2007.

[140] P. Tsigas and Yi Zhang. Non-blocking data sharing in multiprocessor real-time systems. In *Sixth International Conference on Real-Time Computing Systems and Applications*, pages 247 –254, 1999.

[141] P. Tsigas, Yi Zhang, D. Cederman, and T. Dellsen. Wait-free queue algorithms for the real-time java specification. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 373 – 383, april 2006.

[142] A. Turcu, B. Ravindran, and M.M. Saad. On closed nesting in distributed transactional memory. In *Seventh ACM SIGPLAN workshop on Transactional Computing*, 2012.

[143] Alexandru Turcu. *On Improving Distributed Transactional Memory Through Nesting and Data Partitioning*. Phd proposal, Virginia Tech, 2012. Available as `http://www.ssrg.ece.vt.edu/theses/PhdProposal_Turcu.pdf`.

[144] University of Rochester. Rochester Software Transactional Memory. `http://www.cs.rochester.edu/research/synchronization/rstm/index.shtml`, `http://code.google.com/p/rstm`, 2006.

[145] University of Virginia Polytechnique Institue. Hyflow. `http://www.hyflow.org/hyflow`.

[146] University of Virginia Polytechnique Institue. HyflowCPP. `http://www.hyflow.org/hyflow/wiki/HyflowCPP`.

[147] H. Volos, A. Welc, A.R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. Nepaltm: design and implementation of nested parallelism for transactional memory systems. *ECOOP 2009–Object-Oriented Programming*, pages 123–147, 2009.

[148] B Ward and J Anderson. Nested multiprocessor real-time locking with improved blocking. In *Proceedings of the 24th Euromicro conference on real-time systems*, 2012.

[149] B.C. Ward and J.H. Anderson. Supporting nested locking in multiprocessor real-time systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 223 –232, july 2012. longer version available at `http://www.cs.unc.edu/~anderson/papers.html`.

[150] B.C. Ward, G.A. Elliott, and J.H. Anderson. Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *IEEE 18th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 280 –289, aug. 2012.

[151] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Volos, M.D. Hill, M.M. Swift, and D.A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 261 –272, feb. 2007.

[152] Kam yiu Lam, Tei-Wei Kuo, and T.S.H. Lee. Designing inter-class concurrency control strategies for real-time database systems with mixed transactions. In *12th Euromicro Conference on Real-Time Systems (Euromicro RTS)*, pages 47 –54, 2000.

[153] P.S. Yu, Kun-Lung Wu, Kwei-Jay Lin, and S.H. Son. On real-time databases: concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140 –157, jan 1994.

Appendix A

Tasksets' Properties

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
1	11	ul	0.1	ul	0.08	ul	0.05	5	ul	2	ul
2	3	ul	0.11	ul	0.09	ul	0.05	5	ul	2	um
3	2	ul	0.06	ul	0.01	ul	0.01	5	ul	2	uh
4	12	ul	0.26	ul	0.05	ul	0.03	5	um	2	ul
5	3	ul	0.27	ul	0.19	ul	0.17	5	um	2	um
6	2	ul	0.03	ul	0.02	ul	0.01	5	um	2	uh
7	12	ul	0.08	ul	0.06	ul	0.04	5	uh	2	ul
8	4	ul	0.01	ul	0.01	ul	0.01	5	uh	2	um
9	2	ul	0.07	ul	0.07	ul	0.06	5	uh	2	uh
10	9	ul	0.17	ul	0.16	ul	0.14	20	ul	2	ul
11	4	ul	0.29	ul	0.25	ul	0.16	20	ul	2	um
12	2	ul	0.25	ul	0.23	ul	0.22	20	ul	2	uh
13	11	ul	0.12	ul	0.12	ul	0.09	20	um	2	ul
14	3	ul	0.26	ul	0.09	ul	0.09	20	um	2	um
15	2	ul	0.16	ul	0.08	ul	0.03	20	um	2	uh
16	12	ul	0.06	ul	0.03	ul	0.01	20	uh	2	ul
17	4	ul	0.26	ul	0.04	ul	0.04	20	uh	2	um
18	2	ul	0.19	ul	0.06	ul	0.05	20	uh	2	uh
19	10	ul	0.29	ul	0.27	ul	0.12	40	ul	2	ul
20	4	ul	0.08	ul	0.01	ul	0.01	40	ul	2	um
21	2	ul	0.03	ul	0.02	ul	0.01	40	ul	2	uh
22	12	ul	0.04	ul	0.02	ul	0.02	40	um	2	ul
23	3	ul	0.21	ul	0.12	ul	0.07	40	um	2	um
24	2	ul	0.06	ul	0.03	ul	0.03	40	um	2	uh
25	12	ul	0.33	ul	0.11	ul	0.1	40	uh	2	ul
26	4	ul	0.31	ul	0.07	ul	0.01	40	uh	2	um

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
27	2	ul	0.22	ul	0.13	ul	0.02	40	uh	2	uh
28	8	um	0.55	ul	0.04	ul	0.02	5	ul	2	ul
29	3	um	0.56	ul	0.28	ul	0.15	5	ul	2	um
30	2	um	0.42	ul	0.1	ul	0.04	5	ul	2	uh
31	11	um	0.57	ul	0.03	ul	0.01	5	um	2	ul
32	4	um	0.62	ul	0.07	ul	0.05	5	um	2	um
33	2	um	0.56	ul	0.26	ul	0.13	5	um	2	uh
34	14	um	0.63	ul	0.3	ul	0.01	5	uh	2	ul
35	3	um	0.61	ul	0.31	ul	0.18	5	uh	2	um
36	2	um	0.57	ul	0.04	ul	0.01	5	uh	2	uh
37	10	um	0.34	ul	0.18	ul	0.18	20	ul	2	ul
38	3	um	0.35	ul	0.17	ul	0.17	20	ul	2	um
39	2	um	0.47	ul	0.08	ul	0.02	20	ul	2	uh
40	11	um	0.35	ul	0.18	ul	0.15	20	um	2	ul
41	3	um	0.6	ul	0.25	ul	0.1	20	um	2	um
42	2	um	0.36	ul	0.33	ul	0.29	20	um	2	uh
43	12	um	0.65	ul	0.07	ul	0.04	20	uh	2	ul
44	4	um	0.49	ul	0.17	ul	0.02	20	uh	2	um
45	2	um	0.53	ul	0.18	ul	0.05	20	uh	2	uh
46	13	um	0.49	ul	0.33	ul	0.1	40	ul	2	ul
47	4	um	0.58	ul	0.24	ul	0.15	40	ul	2	um
48	2	um	0.49	ul	0.07	ul	0.04	40	ul	2	uh
49	12	um	0.51	ul	0.34	ul	0.11	40	um	2	ul
50	3	um	0.66	ul	0.12	ul	0.03	40	um	2	um
51	2	um	0.52	ul	0.04	ul	0.03	40	um	2	uh
52	13	um	0.58	ul	0.27	ul	0.26	40	uh	2	ul
53	4	um	0.35	ul	0.16	ul	0.08	40	uh	2	um
54	2	um	0.61	ul	0.17	ul	0.01	40	uh	2	uh
55	10	um	0.65	um	0.57	ul	0.27	5	ul	2	ul
56	3	um	0.4	um	0.38	ul	0.01	5	ul	2	um
57	2	um	0.51	um	0.35	ul	0.28	5	ul	2	uh
58	13	um	0.49	um	0.45	ul	0.02	5	um	2	ul
59	4	um	0.6	um	0.5	ul	0.2	5	um	2	um
60	2	um	0.6	um	0.43	ul	0.19	5	um	2	uh
61	10	um	0.59	um	0.44	ul	0.26	5	uh	2	ul
62	3	um	0.55	um	0.43	ul	0.34	5	uh	2	um
63	2	um	0.6	um	0.57	ul	0.21	5	uh	2	uh
64	7	um	0.46	um	0.36	ul	0.05	20	ul	2	ul
65	3	um	0.49	um	0.4	ul	0.28	20	ul	2	um
66	2	um	0.64	um	0.58	ul	0.04	20	ul	2	uh

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
67	14	um	0.36	um	0.36	ul	0.12	20	um	2	ul
68	3	um	0.41	um	0.37	ul	0.08	20	um	2	um
69	2	um	0.6	um	0.47	ul	0.08	20	um	2	uh
70	9	um	0.45	um	0.35	ul	0.18	20	uh	2	ul
71	4	um	0.66	um	0.55	ul	0.02	20	uh	2	um
72	2	um	0.42	um	0.41	ul	0.1	20	uh	2	uh
73	9	um	0.54	um	0.41	ul	0.24	40	ul	2	ul
74	4	um	0.35	um	0.35	ul	0.26	40	ul	2	um
75	2	um	0.4	um	0.4	ul	0.12	40	ul	2	uh
76	10	um	0.34	um	0.34	ul	0.16	40	um	2	ul
77	4	um	0.65	um	0.58	ul	0.06	40	um	2	um
78	2	um	0.44	um	0.4	ul	0.1	40	um	2	uh
79	8	um	0.63	um	0.62	ul	0.03	40	uh	2	ul
80	4	um	0.44	um	0.37	ul	0.28	40	uh	2	um
81	2	um	0.41	um	0.38	ul	0.16	40	uh	2	uh
82	12	um	0.49	um	0.41	um	0.37	5	ul	2	ul
83	4	um	0.56	um	0.48	um	0.41	5	ul	2	um
84	2	um	0.38	um	0.38	um	0.34	5	ul	2	uh
85	10	um	0.37	um	0.34	um	0.34	5	um	2	ul
86	4	um	0.36	um	0.34	um	0.34	5	um	2	um
87	2	um	0.66	um	0.38	um	0.34	5	um	2	uh
88	12	um	0.5	um	0.35	um	0.34	5	uh	2	ul
89	4	um	0.4	um	0.4	um	0.34	5	uh	2	um
90	2	um	0.5	um	0.48	um	0.35	5	uh	2	uh
91	11	um	0.59	um	0.35	um	0.34	20	ul	2	ul
92	3	um	0.49	um	0.4	um	0.39	20	ul	2	um
93	2	um	0.56	um	0.43	um	0.39	20	ul	2	uh
94	11	um	0.5	um	0.39	um	0.39	20	um	2	ul
95	4	um	0.35	um	0.35	um	0.34	20	um	2	um
96	2	um	0.34	um	0.34	um	0.34	20	um	2	uh
97	10	um	0.56	um	0.4	um	0.35	20	uh	2	ul
98	4	um	0.56	um	0.34	um	0.34	20	uh	2	um
99	2	um	0.61	um	0.47	um	0.37	20	uh	2	uh
100	11	um	0.62	um	0.46	um	0.44	40	ul	2	ul
101	3	um	0.47	um	0.42	um	0.35	40	ul	2	um
102	2	um	0.36	um	0.36	um	0.35	40	ul	2	uh
103	13	um	0.62	um	0.47	um	0.41	40	um	2	ul
104	3	um	0.51	um	0.37	um	0.34	40	um	2	um
105	2	um	0.39	um	0.39	um	0.38	40	um	2	uh
106	16	um	0.6	um	0.59	um	0.55	40	uh	2	ul

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
107	4	um	0.41	um	0.36	um	0.36	40	uh	2	um
108	2	um	0.54	um	0.53	um	0.34	40	uh	2	uh
109	14	uh	0.77	ul	0.11	ul	0.02	5	ul	2	ul
110	3	uh	0.76	ul	0.18	ul	0.04	5	ul	2	um
111	2	uh	0.68	ul	0.22	ul	0.02	5	ul	2	uh
112	10	uh	0.95	ul	0.23	ul	0.03	5	um	2	ul
113	3	uh	0.92	ul	0.3	ul	0.18	5	um	2	um
114	2	uh	0.71	ul	0.13	ul	0.02	5	um	2	uh
115	11	uh	0.74	ul	0.24	ul	0.14	5	uh	2	ul
116	3	uh	0.84	ul	0.07	ul	0.07	5	uh	2	um
117	2	uh	0.94	ul	0.17	ul	0.05	5	uh	2	uh
118	8	uh	0.98	ul	0.27	ul	0.04	20	ul	2	ul
119	4	uh	0.69	ul	0.17	ul	0.1	20	ul	2	um
120	2	uh	0.71	ul	0.06	ul	0.03	20	ul	2	uh
121	11	uh	0.92	ul	0.05	ul	0.05	20	um	2	ul
122	3	uh	0.89	ul	0.3	ul	0.03	20	um	2	um
123	2	uh	0.91	ul	0.14	ul	0.04	20	um	2	uh
124	13	uh	0.85	ul	0.12	ul	0.08	20	uh	2	ul
125	3	uh	0.96	ul	0.29	ul	0.19	20	uh	2	um
126	2	uh	0.71	ul	0.11	ul	0.09	20	uh	2	uh
127	11	uh	0.93	ul	0.31	ul	0.25	40	ul	2	ul
128	4	uh	0.69	ul	0.04	ul	0.01	40	ul	2	um
129	2	uh	0.95	ul	0.04	ul	0.04	40	ul	2	uh
130	12	uh	0.78	ul	0.1	ul	0.03	40	um	2	ul
131	3	uh	0.86	ul	0.15	ul	0.03	40	um	2	um
132	2	uh	0.86	ul	0.02	ul	0.02	40	um	2	uh
133	10	uh	0.95	ul	0.02	ul	0.02	40	uh	2	ul
134	3	uh	0.84	ul	0.1	ul	0.02	40	uh	2	um
135	2	uh	0.94	ul	0.17	ul	0.03	40	uh	2	uh
136	11	uh	0.76	um	0.51	ul	0.28	5	ul	2	ul
137	3	uh	0.72	um	0.49	ul	0.13	5	ul	2	um
138	2	uh	0.75	um	0.46	ul	0.1	5	ul	2	uh
139	10	uh	0.84	um	0.53	ul	0.12	5	um	2	ul
140	4	uh	0.79	um	0.5	ul	0.2	5	um	2	um
141	2	uh	0.99	um	0.59	ul	0.3	5	um	2	uh
142	11	uh	0.95	um	0.53	ul	0.26	5	uh	2	ul
143	3	uh	0.71	um	0.58	ul	0.08	5	uh	2	um
144	2	uh	0.83	um	0.59	ul	0.24	5	uh	2	uh
145	11	uh	0.75	um	0.42	ul	0.06	20	ul	2	ul
146	4	uh	0.88	um	0.45	ul	0.19	20	ul	2	um

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
147	2	uh	0.7	um	0.56	ul	0.29	20	ul	2	uh
148	10	uh	0.8	um	0.49	ul	0.31	20	um	2	ul
149	4	uh	0.68	um	0.65	ul	0.13	20	um	2	um
150	2	uh	0.97	um	0.5	ul	0.29	20	um	2	uh
151	10	uh	0.83	um	0.54	ul	0.33	20	uh	2	ul
152	3	uh	0.93	um	0.48	ul	0.13	20	uh	2	um
153	2	uh	0.81	um	0.37	ul	0.25	20	uh	2	uh
154	12	uh	0.87	um	0.62	ul	0.27	40	ul	2	ul
155	4	uh	0.79	um	0.52	ul	0.17	40	ul	2	um
156	2	uh	0.83	um	0.67	ul	0.15	40	ul	2	uh
157	13	uh	0.85	um	0.59	ul	0.24	40	um	2	ul
158	3	uh	0.9	um	0.44	ul	0.13	40	um	2	um
159	2	uh	0.78	um	0.36	ul	0.19	40	um	2	uh
160	11	uh	0.67	um	0.53	ul	0.02	40	uh	2	ul
161	3	uh	0.68	um	0.65	ul	0.18	40	uh	2	um
162	2	uh	0.69	um	0.35	ul	0.17	40	uh	2	uh
163	14	uh	0.96	um	0.54	um	0.5	5	ul	2	ul
164	4	uh	0.82	um	0.39	um	0.34	5	ul	2	um
165	2	uh	0.82	um	0.47	um	0.34	5	ul	2	uh
166	10	uh	0.72	um	0.61	um	0.53	5	um	2	ul
167	4	uh	0.91	um	0.49	um	0.34	5	um	2	um
168	2	uh	0.73	um	0.54	um	0.47	5	um	2	uh
169	9	uh	0.76	um	0.51	um	0.39	5	uh	2	ul
170	3	uh	0.8	um	0.48	um	0.37	5	uh	2	um
171	2	uh	0.79	um	0.55	um	0.36	5	uh	2	uh
172	12	uh	0.87	um	0.52	um	0.36	20	ul	2	ul
173	4	uh	0.78	um	0.54	um	0.48	20	ul	2	um
174	2	uh	0.87	um	0.55	um	0.47	20	ul	2	uh
175	10	uh	0.85	um	0.53	um	0.37	20	um	2	ul
176	4	uh	0.88	um	0.39	um	0.35	20	um	2	um
177	2	uh	0.94	um	0.6	um	0.38	20	um	2	uh
178	15	uh	0.81	um	0.34	um	0.34	20	uh	2	ul
179	4	uh	0.73	um	0.58	um	0.45	20	uh	2	um
180	2	uh	0.92	um	0.65	um	0.46	20	uh	2	uh
181	11	uh	0.96	um	0.48	um	0.35	40	ul	2	ul
182	4	uh	0.88	um	0.43	um	0.39	40	ul	2	um
183	2	uh	0.96	um	0.36	um	0.34	40	ul	2	uh
184	14	uh	0.79	um	0.38	um	0.36	40	um	2	ul
185	3	uh	0.75	um	0.62	um	0.35	40	um	2	um
186	2	uh	0.79	um	0.58	um	0.46	40	um	2	uh

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
187	11	uh	0.89	um	0.5	um	0.46	40	uh	2	ul
188	3	uh	0.85	um	0.44	um	0.35	40	uh	2	um
189	2	uh	0.75	um	0.5	um	0.36	40	uh	2	uh
190	12	uh	0.75	uh	0.68	ul	0.33	5	ul	2	ul
191	4	uh	0.7	uh	0.68	ul	0.1	5	ul	2	um
192	2	uh	0.76	uh	0.69	ul	0.25	5	ul	2	uh
193	10	uh	0.93	uh	0.68	ul	0.26	5	um	2	ul
194	3	uh	0.77	uh	0.68	ul	0.08	5	um	2	um
195	2	uh	0.96	uh	0.7	ul	0.22	5	um	2	uh
196	10	uh	0.73	uh	0.7	ul	0.01	5	uh	2	ul
197	4	uh	0.94	uh	0.91	ul	0.24	5	uh	2	um
198	2	uh	0.97	uh	0.93	ul	0.14	5	uh	2	uh
199	11	uh	0.9	uh	0.78	ul	0.03	20	ul	2	ul
200	3	uh	0.71	uh	0.71	ul	0.33	20	ul	2	um
201	2	uh	0.78	uh	0.74	ul	0.14	20	ul	2	uh
202	13	uh	0.74	uh	0.74	ul	0.27	20	um	2	ul
203	4	uh	0.83	uh	0.77	ul	0.17	20	um	2	um
204	2	uh	0.72	uh	0.67	ul	0.11	20	um	2	uh
205	11	uh	0.81	uh	0.72	ul	0.13	20	uh	2	ul
206	3	uh	0.73	uh	0.71	ul	0.28	20	uh	2	um
207	2	uh	0.79	uh	0.77	ul	0.26	20	uh	2	uh
208	12	uh	0.71	uh	0.68	ul	0.11	40	ul	2	ul
209	3	uh	0.7	uh	0.7	ul	0.07	40	ul	2	um
210	2	uh	0.76	uh	0.74	ul	0.12	40	ul	2	uh
211	13	uh	0.76	uh	0.71	ul	0.18	40	um	2	ul
212	3	uh	0.69	uh	0.68	ul	0.03	40	um	2	um
213	2	uh	0.78	uh	0.75	ul	0.04	40	um	2	uh
214	9	uh	0.83	uh	0.78	ul	0.07	40	uh	2	ul
215	3	uh	0.93	uh	0.68	ul	0.34	40	uh	2	um
216	2	uh	0.96	uh	0.84	ul	0.11	40	uh	2	uh
217	13	uh	0.85	uh	0.7	um	0.42	5	ul	2	ul
218	4	uh	0.85	uh	0.69	um	0.35	5	ul	2	um
219	2	uh	0.88	uh	0.71	um	0.56	5	ul	2	uh
220	11	uh	0.81	uh	0.79	um	0.55	5	um	2	ul
221	3	uh	0.82	uh	0.79	um	0.35	5	um	2	um
222	2	uh	0.96	uh	0.94	um	0.35	5	um	2	uh
223	12	uh	0.91	uh	0.82	um	0.51	5	uh	2	ul
224	4	uh	0.76	uh	0.72	um	0.4	5	uh	2	um
225	2	uh	0.71	uh	0.67	um	0.36	5	uh	2	uh
226	10	uh	0.82	uh	0.73	um	0.51	20	ul	2	ul

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
227	3	uh	0.73	uh	0.71	um	0.38	20	ul	2	um
228	2	uh	0.9	uh	0.71	um	0.38	20	ul	2	uh
229	10	uh	0.8	uh	0.73	um	0.55	20	um	2	ul
230	4	uh	0.8	uh	0.69	um	0.44	20	um	2	um
231	2	uh	0.75	uh	0.67	um	0.35	20	um	2	uh
232	9	uh	0.76	uh	0.68	um	0.34	20	uh	2	ul
233	4	uh	0.78	uh	0.74	um	0.41	20	uh	2	um
234	2	uh	0.88	uh	0.81	um	0.49	20	uh	2	uh
235	16	uh	0.77	uh	0.7	um	0.51	40	ul	2	ul
236	4	uh	0.82	uh	0.8	um	0.51	40	ul	2	um
237	2	uh	0.96	uh	0.75	um	0.48	40	ul	2	uh
238	12	uh	0.94	uh	0.81	um	0.58	40	um	2	ul
239	3	uh	0.83	uh	0.71	um	0.52	40	um	2	um
240	2	uh	0.93	uh	0.91	um	0.56	40	um	2	uh
241	11	uh	0.87	uh	0.7	um	0.46	40	uh	2	ul
242	3	uh	0.94	uh	0.76	um	0.6	40	uh	2	um
243	2	uh	0.85	uh	0.79	um	0.64	40	uh	2	uh
244	11	uh	0.87	uh	0.78	uh	0.71	5	ul	2	ul
245	3	uh	0.88	uh	0.75	uh	0.74	5	ul	2	um
246	2	uh	0.83	uh	0.69	uh	0.69	5	ul	2	uh
247	13	uh	0.93	uh	0.7	uh	0.67	5	um	2	ul
248	4	uh	0.75	uh	0.68	uh	0.67	5	um	2	um
249	2	uh	0.74	uh	0.72	uh	0.71	5	um	2	uh
250	12	uh	0.95	uh	0.77	uh	0.75	5	uh	2	ul
251	3	uh	0.76	uh	0.73	uh	0.68	5	uh	2	um
252	2	uh	0.87	uh	0.81	uh	0.68	5	uh	2	uh
253	11	uh	0.97	uh	0.76	uh	0.76	20	ul	2	ul
254	3	uh	0.8	uh	0.68	uh	0.67	20	ul	2	um
255	2	uh	0.68	uh	0.68	uh	0.67	20	ul	2	uh
256	8	uh	0.73	uh	0.68	uh	0.67	20	um	2	ul
257	4	uh	0.85	uh	0.85	uh	0.76	20	um	2	um
258	2	uh	0.9	uh	0.87	uh	0.85	20	um	2	uh
259	14	uh	0.9	uh	0.83	uh	0.82	20	uh	2	ul
260	3	uh	0.88	uh	0.78	uh	0.73	20	uh	2	um
261	2	uh	0.81	uh	0.72	uh	0.69	20	uh	2	uh
262	18	uh	0.8	uh	0.76	uh	0.75	40	ul	2	ul
263	4	uh	0.7	uh	0.7	uh	0.67	40	ul	2	um
264	2	uh	0.81	uh	0.81	uh	0.69	40	ul	2	uh
265	9	uh	0.79	uh	0.73	uh	0.73	40	um	2	ul
266	4	uh	0.82	uh	0.74	uh	0.73	40	um	2	um

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
267	2	uh	0.68	uh	0.67	uh	0.67	40	um	2	uh
268	10	uh	0.79	uh	0.75	uh	0.7	40	uh	2	ul
269	4	uh	0.91	uh	0.7	uh	0.67	40	uh	2	um
270	2	uh	0.95	uh	0.82	uh	0.78	40	uh	2	uh
271	22	ul	0.03	ul	0.02	ul	0.02	5	ul	4	ul
272	7	ul	0.2	ul	0.12	ul	0.1	5	ul	4	um
273	4	ul	0.21	ul	0.19	ul	0.09	5	ul	4	uh
274	21	ul	0.13	ul	0.04	ul	0.03	5	um	4	ul
275	8	ul	0.01	ul	0.01	ul	0.01	5	um	4	um
276	4	ul	0.21	ul	0.18	ul	0.11	5	um	4	uh
277	22	ul	0.04	ul	0.01	ul	0.01	5	uh	4	ul
278	7	ul	0.11	ul	0.11	ul	0.04	5	uh	4	um
279	4	ul	0.28	ul	0.24	ul	0.16	5	uh	4	uh
280	25	ul	0.08	ul	0.07	ul	0.07	20	ul	4	ul
281	8	ul	0.32	ul	0.01	ul	0.01	20	ul	4	um
282	4	ul	0.09	ul	0.07	ul	0.06	20	ul	4	uh
283	19	ul	0.01	ul	0.01	ul	0.01	20	um	4	ul
284	7	ul	0.33	ul	0.2	ul	0.15	20	um	4	um
285	4	ul	0.16	ul	0.02	ul	0.01	20	um	4	uh
286	23	ul	0.29	ul	0.28	ul	0.06	20	uh	4	ul
287	7	ul	0.07	ul	0.03	ul	0.03	20	uh	4	um
288	4	ul	0.13	ul	0.07	ul	0.05	20	uh	4	uh
289	25	ul	0.02	ul	0.02	ul	0.01	40	ul	4	ul
290	8	ul	0.05	ul	0.02	ul	0.01	40	ul	4	um
291	5	ul	0.32	ul	0.23	ul	0.23	40	ul	4	uh
292	23	ul	0.28	ul	0.27	ul	0.23	40	um	4	ul
293	7	ul	0.14	ul	0.04	ul	0.03	40	um	4	um
294	5	ul	0.09	ul	0.05	ul	0.05	40	um	4	uh
295	22	ul	0.21	ul	0.07	ul	0.06	40	uh	4	ul
296	8	ul	0.14	ul	0.08	ul	0.03	40	uh	4	um
297	4	ul	0.16	ul	0.11	ul	0.1	40	uh	4	uh
298	26	um	0.37	ul	0.17	ul	0.1	5	ul	4	ul
299	8	um	0.51	ul	0.18	ul	0.1	5	ul	4	um
300	4	um	0.65	ul	0.21	ul	0.11	5	ul	4	uh
301	22	um	0.49	ul	0.32	ul	0.32	5	um	4	ul
302	8	um	0.42	ul	0.04	ul	0.01	5	um	4	um
303	4	um	0.41	ul	0.03	ul	0.03	5	um	4	uh
304	21	um	0.66	ul	0.23	ul	0.1	5	uh	4	ul
305	7	um	0.41	ul	0.29	ul	0.12	5	uh	4	um
306	4	um	0.63	ul	0.31	ul	0.06	5	uh	4	uh

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
307	26	um	0.36	ul	0.27	ul	0.14	20	ul	4	ul
308	7	um	0.57	ul	0.32	ul	0.03	20	ul	4	um
309	4	um	0.53	ul	0.32	ul	0.14	20	ul	4	uh
310	21	um	0.56	ul	0.19	ul	0.03	20	um	4	ul
311	8	um	0.64	ul	0.12	ul	0.02	20	um	4	um
312	4	um	0.59	ul	0.29	ul	0.16	20	um	4	uh
313	21	um	0.34	ul	0.23	ul	0.13	20	uh	4	ul
314	8	um	0.58	ul	0.04	ul	0.03	20	uh	4	um
315	5	um	0.5	ul	0.04	ul	0.02	20	uh	4	uh
316	23	um	0.45	ul	0.26	ul	0.18	40	ul	4	ul
317	9	um	0.5	ul	0.24	ul	0.06	40	ul	4	um
318	4	um	0.54	ul	0.08	ul	0.02	40	ul	4	uh
319	24	um	0.47	ul	0.02	ul	0.02	40	um	4	ul
320	7	um	0.57	ul	0.32	ul	0.17	40	um	4	um
321	5	um	0.64	ul	0.16	ul	0.13	40	um	4	uh
322	23	um	0.46	ul	0.25	ul	0.18	40	uh	4	ul
323	7	um	0.42	ul	0.25	ul	0.22	40	uh	4	um
324	4	um	0.55	ul	0.07	ul	0.04	40	uh	4	uh
325	27	um	0.42	um	0.35	ul	0.32	5	ul	4	ul
326	8	um	0.64	um	0.64	ul	0.31	5	ul	4	um
327	4	um	0.48	um	0.4	ul	0.21	5	ul	4	uh
328	24	um	0.48	um	0.38	ul	0.28	5	um	4	ul
329	7	um	0.49	um	0.46	ul	0.33	5	um	4	um
330	4	um	0.39	um	0.37	ul	0.19	5	um	4	uh
331	24	um	0.45	um	0.38	ul	0.18	5	uh	4	ul
332	8	um	0.45	um	0.36	ul	0.05	5	uh	4	um
333	5	um	0.62	um	0.5	ul	0.07	5	uh	4	uh
334	26	um	0.6	um	0.45	ul	0.19	20	ul	4	ul
335	8	um	0.49	um	0.35	ul	0.25	20	ul	4	um
336	4	um	0.37	um	0.35	ul	0.04	20	ul	4	uh
337	29	um	0.52	um	0.45	ul	0.04	20	um	4	ul
338	7	um	0.36	um	0.36	ul	0.23	20	um	4	um
339	4	um	0.64	um	0.38	ul	0.03	20	um	4	uh
340	26	um	0.39	um	0.34	ul	0.19	20	uh	4	ul
341	8	um	0.38	um	0.37	ul	0.1	20	uh	4	um
342	5	um	0.43	um	0.34	ul	0.21	20	uh	4	uh
343	23	um	0.65	um	0.39	ul	0.3	40	ul	4	ul
344	8	um	0.59	um	0.47	ul	0.28	40	ul	4	um
345	4	um	0.45	um	0.39	ul	0.12	40	ul	4	uh
346	21	um	0.36	um	0.36	ul	0.08	40	um	4	ul

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
347	8	um	0.58	um	0.54	ul	0.24	40	um	4	um
348	4	um	0.43	um	0.42	ul	0.29	40	um	4	uh
349	20	um	0.35	um	0.34	ul	0.15	40	uh	4	ul
350	8	um	0.66	um	0.63	ul	0.33	40	uh	4	um
351	4	um	0.59	um	0.52	ul	0.23	40	uh	4	uh
352	19	um	0.64	um	0.39	um	0.39	5	ul	4	ul
353	8	um	0.48	um	0.38	um	0.34	5	ul	4	um
354	5	um	0.34	um	0.34	um	0.34	5	ul	4	uh
355	20	um	0.41	um	0.38	um	0.37	5	um	4	ul
356	7	um	0.56	um	0.39	um	0.34	5	um	4	um
357	4	um	0.57	um	0.35	um	0.35	5	um	4	uh
358	25	um	0.36	um	0.35	um	0.35	5	uh	4	ul
359	7	um	0.6	um	0.42	um	0.35	5	uh	4	um
360	4	um	0.56	um	0.52	um	0.46	5	uh	4	uh
361	17	um	0.55	um	0.43	um	0.41	20	ul	4	ul
362	7	um	0.47	um	0.45	um	0.4	20	ul	4	um
363	4	um	0.5	um	0.48	um	0.36	20	ul	4	uh
364	17	um	0.43	um	0.43	um	0.34	20	um	4	ul
365	7	um	0.56	um	0.42	um	0.41	20	um	4	um
366	4	um	0.38	um	0.38	um	0.38	20	um	4	uh
367	28	um	0.62	um	0.37	um	0.35	20	uh	4	ul
368	7	um	0.38	um	0.34	um	0.34	20	uh	4	um
369	5	um	0.65	um	0.51	um	0.48	20	uh	4	uh
370	24	um	0.54	um	0.42	um	0.35	40	ul	4	ul
371	8	um	0.49	um	0.34	um	0.34	40	ul	4	um
372	4	um	0.5	um	0.45	um	0.39	40	ul	4	uh
373	26	um	0.53	um	0.39	um	0.34	40	um	4	ul
374	8	um	0.6	um	0.53	um	0.51	40	um	4	um
375	5	um	0.52	um	0.45	um	0.37	40	um	4	uh
376	24	um	0.41	um	0.39	um	0.39	40	uh	4	ul
377	8	um	0.37	um	0.37	um	0.35	40	uh	4	um
378	5	um	0.35	um	0.34	um	0.34	40	uh	4	uh
379	20	uh	0.77	ul	0.22	ul	0.17	5	ul	4	ul
380	7	uh	0.72	ul	0.2	ul	0.08	5	ul	4	um
381	4	uh	0.74	ul	0.27	ul	0.08	5	ul	4	uh
382	21	uh	0.87	ul	0.14	ul	0.09	5	um	4	ul
383	7	uh	0.93	ul	0.32	ul	0.04	5	um	4	um
384	5	uh	0.96	ul	0.01	ul	0.01	5	um	4	uh
385	29	uh	0.73	ul	0.26	ul	0.02	5	uh	4	ul
386	9	uh	0.86	ul	0.13	ul	0.12	5	uh	4	um

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
387	4	uh	0.71	ul	0.28	ul	0.09	5	uh	4	uh
388	25	uh	0.67	ul	0.18	ul	0.04	20	ul	4	ul
389	8	uh	0.68	ul	0.06	ul	0.06	20	ul	4	um
390	4	uh	0.85	ul	0.06	ul	0.03	20	ul	4	uh
391	19	uh	0.69	ul	0.07	ul	0.06	20	um	4	ul
392	9	uh	0.78	ul	0.27	ul	0.18	20	um	4	um
393	4	uh	0.72	ul	0.34	ul	0.2	20	um	4	uh
394	28	uh	0.83	ul	0.19	ul	0.06	20	uh	4	ul
395	7	uh	0.94	ul	0.13	ul	0.07	20	uh	4	um
396	4	uh	0.69	ul	0.25	ul	0.2	20	uh	4	uh
397	23	uh	0.76	ul	0.08	ul	0.04	40	ul	4	ul
398	8	uh	0.78	ul	0.1	ul	0.02	40	ul	4	um
399	4	uh	0.78	ul	0.3	ul	0.2	40	ul	4	uh
400	26	uh	0.73	ul	0.31	ul	0.17	40	um	4	ul
401	7	uh	0.7	ul	0.19	ul	0.03	40	um	4	um
402	4	uh	0.8	ul	0.29	ul	0.04	40	um	4	uh
403	23	uh	0.77	ul	0.12	ul	0.09	40	uh	4	ul
404	8	uh	0.82	ul	0.07	ul	0.07	40	uh	4	um
405	4	uh	0.68	ul	0.26	ul	0.23	40	uh	4	uh
406	25	uh	0.89	um	0.36	ul	0.07	5	ul	4	ul
407	9	uh	0.93	um	0.51	ul	0.21	5	ul	4	um
408	5	uh	0.69	um	0.47	ul	0.3	5	ul	4	uh
409	22	uh	0.97	um	0.64	ul	0.05	5	um	4	ul
410	8	uh	0.99	um	0.48	ul	0.04	5	um	4	um
411	4	uh	0.72	um	0.64	ul	0.14	5	um	4	uh
412	32	uh	0.69	um	0.43	ul	0.12	5	uh	4	ul
413	8	uh	0.92	um	0.62	ul	0.03	5	uh	4	um
414	4	uh	0.69	um	0.5	ul	0.28	5	uh	4	uh
415	26	uh	0.84	um	0.49	ul	0.08	20	ul	4	ul
416	8	uh	0.87	um	0.64	ul	0.24	20	ul	4	um
417	4	uh	0.88	um	0.43	ul	0.18	20	ul	4	uh
418	25	uh	0.68	um	0.35	ul	0.29	20	um	4	ul
419	7	uh	0.9	um	0.49	ul	0.3	20	um	4	um
420	5	uh	0.79	um	0.53	ul	0.11	20	um	4	uh
421	24	uh	0.84	um	0.42	ul	0.25	20	uh	4	ul
422	8	uh	0.87	um	0.38	ul	0.19	20	uh	4	um
423	5	uh	0.71	um	0.38	ul	0.27	20	uh	4	uh
424	27	uh	0.93	um	0.39	ul	0.34	40	ul	4	ul
425	8	uh	0.7	um	0.37	ul	0.3	40	ul	4	um
426	5	uh	0.89	um	0.59	ul	0.04	40	ul	4	uh

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
427	25	uh	0.84	um	0.42	ul	0.2	40	um	4	ul
428	8	uh	0.91	um	0.6	ul	0.21	40	um	4	um
429	4	uh	0.77	um	0.59	ul	0.16	40	um	4	uh
430	24	uh	0.91	um	0.56	ul	0.15	40	uh	4	ul
431	9	uh	0.93	um	0.49	ul	0.23	40	uh	4	um
432	4	uh	0.78	um	0.67	ul	0.19	40	uh	4	uh
433	23	uh	0.97	um	0.45	um	0.37	5	ul	4	ul
434	8	uh	0.94	um	0.36	um	0.34	5	ul	4	um
435	4	uh	0.93	um	0.54	um	0.34	5	ul	4	uh
436	22	uh	0.73	um	0.66	um	0.59	5	um	4	ul
437	7	uh	0.73	um	0.55	um	0.35	5	um	4	um
438	4	uh	0.95	um	0.46	um	0.44	5	um	4	uh
439	20	uh	0.91	um	0.37	um	0.37	5	uh	4	ul
440	7	uh	0.99	um	0.55	um	0.42	5	uh	4	um
441	4	uh	0.93	um	0.54	um	0.45	5	uh	4	uh
442	24	uh	0.8	um	0.67	um	0.53	20	ul	4	ul
443	8	uh	0.71	um	0.37	um	0.36	20	ul	4	um
444	4	uh	0.98	um	0.47	um	0.34	20	ul	4	uh
445	28	uh	0.85	um	0.35	um	0.35	20	um	4	ul
446	8	uh	0.9	um	0.41	um	0.36	20	um	4	um
447	5	uh	0.69	um	0.55	um	0.44	20	um	4	uh
448	20	uh	0.76	um	0.38	um	0.35	20	uh	4	ul
449	7	uh	0.69	um	0.37	um	0.34	20	uh	4	um
450	4	uh	0.92	um	0.43	um	0.34	20	uh	4	uh
451	24	uh	0.94	um	0.44	um	0.38	40	ul	4	ul
452	8	uh	0.98	um	0.37	um	0.37	40	ul	4	um
453	4	uh	0.77	um	0.66	um	0.57	40	ul	4	uh
454	24	uh	0.73	um	0.41	um	0.34	40	um	4	ul
455	8	uh	0.89	um	0.59	um	0.51	40	um	4	um
456	4	uh	0.8	um	0.35	um	0.35	40	um	4	uh
457	20	uh	0.95	um	0.44	um	0.38	40	uh	4	ul
458	8	uh	0.87	um	0.59	um	0.43	40	uh	4	um
459	4	uh	0.93	um	0.55	um	0.41	40	uh	4	uh
460	31	uh	0.88	uh	0.76	ul	0.04	5	ul	4	ul
461	7	uh	0.87	uh	0.75	ul	0.15	5	ul	4	um
462	4	uh	0.89	uh	0.69	ul	0.22	5	ul	4	uh
463	20	uh	0.77	uh	0.71	ul	0.01	5	um	4	ul
464	8	uh	0.72	uh	0.67	ul	0.04	5	um	4	um
465	4	uh	0.97	uh	0.69	ul	0.22	5	um	4	uh
466	27	uh	0.81	uh	0.7	ul	0.21	5	uh	4	ul

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
467	7	uh	0.78	uh	0.67	ul	0.3	5	uh	4	um
468	4	uh	0.96	uh	0.84	ul	0.16	5	uh	4	uh
469	25	uh	0.88	uh	0.8	ul	0.31	20	ul	4	ul
470	8	uh	0.83	uh	0.67	ul	0.25	20	ul	4	um
471	5	uh	0.67	uh	0.67	ul	0.15	20	ul	4	uh
472	21	uh	0.78	uh	0.67	ul	0.09	20	um	4	ul
473	7	uh	0.71	uh	0.7	ul	0.07	20	um	4	um
474	5	uh	0.93	uh	0.71	ul	0.2	20	um	4	uh
475	20	uh	0.72	uh	0.7	ul	0.19	20	uh	4	ul
476	8	uh	0.7	uh	0.67	ul	0.04	20	uh	4	um
477	4	uh	0.9	uh	0.69	ul	0.21	20	uh	4	uh
478	21	uh	0.77	uh	0.67	ul	0.08	40	ul	4	ul
479	7	uh	0.73	uh	0.7	ul	0.11	40	ul	4	um
480	5	uh	0.99	uh	0.82	ul	0.19	40	ul	4	uh
481	25	uh	0.81	uh	0.73	ul	0.21	40	um	4	ul
482	7	uh	0.77	uh	0.67	ul	0.2	40	um	4	um
483	4	uh	0.79	uh	0.67	ul	0.1	40	um	4	uh
484	25	uh	0.96	uh	0.83	ul	0.17	40	uh	4	ul
485	8	uh	0.71	uh	0.67	ul	0.23	40	uh	4	um
486	5	uh	0.97	uh	0.84	ul	0.28	40	uh	4	uh
487	20	uh	0.79	uh	0.73	um	0.64	5	ul	4	ul
488	7	uh	0.98	uh	0.74	um	0.42	5	ul	4	um
489	5	uh	0.76	uh	0.75	um	0.38	5	ul	4	uh
490	23	uh	0.84	uh	0.67	um	0.64	5	um	4	ul
491	8	uh	0.8	uh	0.74	um	0.4	5	um	4	um
492	4	uh	0.77	uh	0.75	um	0.62	5	um	4	uh
493	22	uh	0.94	uh	0.87	um	0.6	5	uh	4	ul
494	8	uh	0.74	uh	0.73	um	0.4	5	uh	4	um
495	4	uh	0.75	uh	0.73	um	0.53	5	uh	4	uh
496	23	uh	0.86	uh	0.75	um	0.63	20	ul	4	ul
497	7	uh	0.83	uh	0.77	um	0.66	20	ul	4	um
498	5	uh	0.93	uh	0.9	um	0.65	20	ul	4	uh
499	26	uh	0.72	uh	0.7	um	0.64	20	um	4	ul
500	7	uh	0.74	uh	0.7	um	0.66	20	um	4	um
501	5	uh	0.67	uh	0.67	um	0.58	20	um	4	uh
502	22	uh	0.97	uh	0.87	um	0.37	20	uh	4	ul
503	7	uh	0.79	uh	0.76	um	0.59	20	uh	4	um
504	4	uh	0.88	uh	0.75	um	0.48	20	uh	4	uh
505	22	uh	0.9	uh	0.71	um	0.64	40	ul	4	ul
506	8	uh	0.91	uh	0.91	um	0.41	40	ul	4	um

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
507	4	uh	0.78	uh	0.67	um	0.54	40	ul	4	uh
508	25	uh	0.91	uh	0.8	um	0.5	40	um	4	ul
509	7	uh	0.87	uh	0.86	um	0.62	40	um	4	um
510	4	uh	0.78	uh	0.67	um	0.42	40	um	4	uh
511	21	uh	0.67	uh	0.67	um	0.6	40	uh	4	ul
512	7	uh	0.85	uh	0.68	um	0.55	40	uh	4	um
513	4	uh	0.9	uh	0.78	um	0.45	40	uh	4	uh
514	26	uh	0.96	uh	0.92	uh	0.76	5	ul	4	ul
515	8	uh	0.97	uh	0.86	uh	0.79	5	ul	4	um
516	4	uh	0.82	uh	0.77	uh	0.69	5	ul	4	uh
517	24	uh	0.93	uh	0.81	uh	0.79	5	um	4	ul
518	8	uh	0.74	uh	0.72	uh	0.72	5	um	4	um
519	5	uh	0.88	uh	0.7	uh	0.7	5	um	4	uh
520	23	uh	0.83	uh	0.68	uh	0.68	5	uh	4	ul
521	7	uh	0.75	uh	0.73	uh	0.7	5	uh	4	um
522	4	uh	0.95	uh	0.85	uh	0.77	5	uh	4	uh
523	28	uh	0.83	uh	0.81	uh	0.75	20	ul	4	ul
524	8	uh	0.69	uh	0.69	uh	0.69	20	ul	4	um
525	4	uh	0.97	uh	0.75	uh	0.69	20	ul	4	uh
526	21	uh	0.81	uh	0.71	uh	0.7	20	um	4	ul
527	7	uh	0.69	uh	0.69	uh	0.67	20	um	4	um
528	4	uh	0.77	uh	0.74	uh	0.68	20	um	4	uh
529	25	uh	0.8	uh	0.75	uh	0.72	20	uh	4	ul
530	8	uh	0.81	uh	0.69	uh	0.69	20	uh	4	um
531	5	uh	0.83	uh	0.83	uh	0.77	20	uh	4	uh
532	19	uh	0.76	uh	0.75	uh	0.75	40	ul	4	ul
533	8	uh	0.9	uh	0.86	uh	0.83	40	ul	4	um
534	4	uh	0.94	uh	0.84	uh	0.67	40	ul	4	uh
535	23	uh	0.73	uh	0.72	uh	0.67	40	um	4	ul
536	7	uh	0.95	uh	0.9	uh	0.69	40	um	4	um
537	4	uh	0.86	uh	0.86	uh	0.83	40	um	4	uh
538	23	uh	0.78	uh	0.76	uh	0.68	40	uh	4	ul
539	7	uh	0.7	uh	0.7	uh	0.69	40	uh	4	um
540	5	uh	0.89	uh	0.8	uh	0.79	40	uh	4	uh
541	33	ul	0.02	ul	0.01	ul	0.01	5	ul	6	ul
542	11	ul	0.3	ul	0.29	ul	0.18	5	ul	6	um
543	7	ul	0.23	ul	0.07	ul	0.02	5	ul	6	uh
544	35	ul	0.01	ul	0.01	ul	0.01	5	um	6	ul
545	11	ul	0.14	ul	0.12	ul	0.02	5	um	6	um
546	7	ul	0.23	ul	0.04	ul	0.04	5	um	6	uh

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
547	31	ul	0.13	ul	0.11	ul	0.01	5	uh	6	ul
548	12	ul	0.3	ul	0.13	ul	0.06	5	uh	6	um
549	6	ul	0.03	ul	0.01	ul	0.01	5	uh	6	uh
550	45	ul	0.25	ul	0.12	ul	0.06	20	ul	6	ul
551	11	ul	0.21	ul	0.12	ul	0.07	20	ul	6	um
552	7	ul	0.13	ul	0.08	ul	0.03	20	ul	6	uh
553	31	ul	0.04	ul	0.02	ul	0.02	20	um	6	ul
554	13	ul	0.12	ul	0.06	ul	0.02	20	um	6	um
555	6	ul	0.32	ul	0.16	ul	0.09	20	um	6	uh
556	37	ul	0.32	ul	0.04	ul	0.01	20	uh	6	ul
557	12	ul	0.3	ul	0.07	ul	0.01	20	uh	6	um
558	7	ul	0.12	ul	0.04	ul	0.04	20	uh	6	uh
559	36	ul	0.32	ul	0.26	ul	0.25	40	ul	6	ul
560	12	ul	0.17	ul	0.16	ul	0.1	40	ul	6	um
561	6	ul	0.23	ul	0.06	ul	0.02	40	ul	6	uh
562	35	ul	0.03	ul	0.03	ul	0.03	40	um	6	ul
563	11	ul	0.18	ul	0.11	ul	0.04	40	um	6	um
564	6	ul	0.11	ul	0.05	ul	0.03	40	um	6	uh
565	35	ul	0.15	ul	0.14	ul	0.04	40	uh	6	ul
566	11	ul	0.04	ul	0.03	ul	0.03	40	uh	6	um
567	7	ul	0.13	ul	0.02	ul	0.01	40	uh	6	uh
568	34	um	0.41	ul	0.1	ul	0.05	5	ul	6	ul
569	13	um	0.62	ul	0.04	ul	0.01	5	ul	6	um
570	7	um	0.42	ul	0.14	ul	0.05	5	ul	6	uh
571	36	um	0.52	ul	0.26	ul	0.05	5	um	6	ul
572	11	um	0.53	ul	0.3	ul	0.18	5	um	6	um
573	7	um	0.35	ul	0.16	ul	0.13	5	um	6	uh
574	37	um	0.49	ul	0.33	ul	0.29	5	uh	6	ul
575	12	um	0.57	ul	0.03	ul	0.02	5	uh	6	um
576	7	um	0.39	ul	0.05	ul	0.05	5	uh	6	uh
577	36	um	0.38	ul	0.2	ul	0.04	20	ul	6	ul
578	11	um	0.46	ul	0.06	ul	0.03	20	ul	6	um
579	7	um	0.59	ul	0.27	ul	0.08	20	ul	6	uh
580	35	um	0.55	ul	0.09	ul	0.04	20	um	6	ul
581	11	um	0.55	ul	0.09	ul	0.06	20	um	6	um
582	6	um	0.55	ul	0.26	ul	0.14	20	um	6	uh
583	35	um	0.65	ul	0.14	ul	0.06	20	uh	6	ul
584	12	um	0.43	ul	0.21	ul	0.19	20	uh	6	um
585	7	um	0.61	ul	0.14	ul	0.03	20	uh	6	uh
586	37	um	0.63	ul	0.29	ul	0.09	40	ul	6	ul

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
587	12	um	0.43	ul	0.19	ul	0.11	40	ul	6	um
588	7	um	0.39	ul	0.01	ul	0.01	40	ul	6	uh
589	34	um	0.55	ul	0.16	ul	0.14	40	um	6	ul
590	11	um	0.35	ul	0.12	ul	0.09	40	um	6	um
591	7	um	0.4	ul	0.11	ul	0.03	40	um	6	uh
592	34	um	0.5	ul	0.02	ul	0.02	40	uh	6	ul
593	12	um	0.63	ul	0.22	ul	0.09	40	uh	6	um
594	7	um	0.52	ul	0.07	ul	0.07	40	uh	6	uh
595	40	um	0.64	um	0.63	ul	0.23	5	ul	6	ul
596	11	um	0.38	um	0.34	ul	0.1	5	ul	6	um
597	7	um	0.35	um	0.34	ul	0.13	5	ul	6	uh
598	32	um	0.35	um	0.35	ul	0.09	5	um	6	ul
599	12	um	0.61	um	0.39	ul	0.06	5	um	6	um
600	7	um	0.39	um	0.34	ul	0.34	5	um	6	uh
601	32	um	0.6	um	0.47	ul	0.34	5	uh	6	ul
602	11	um	0.55	um	0.52	ul	0.27	5	uh	6	um
603	7	um	0.49	um	0.4	ul	0.05	5	uh	6	uh
604	38	um	0.46	um	0.39	ul	0.04	20	ul	6	ul
605	12	um	0.47	um	0.34	ul	0.16	20	ul	6	um
606	6	um	0.59	um	0.53	ul	0.32	20	ul	6	uh
607	33	um	0.57	um	0.44	ul	0.22	20	um	6	ul
608	13	um	0.45	um	0.37	ul	0.29	20	um	6	um
609	7	um	0.5	um	0.36	ul	0.21	20	um	6	uh
610	47	um	0.41	um	0.37	ul	0.11	20	uh	6	ul
611	11	um	0.46	um	0.35	ul	0.04	20	uh	6	um
612	6	um	0.47	um	0.39	ul	0.29	20	uh	6	uh
613	35	um	0.61	um	0.5	ul	0.13	40	ul	6	ul
614	11	um	0.43	um	0.42	ul	0.33	40	ul	6	um
615	7	um	0.62	um	0.59	ul	0.14	40	ul	6	uh
616	31	um	0.53	um	0.51	ul	0.22	40	um	6	ul
617	11	um	0.38	um	0.38	ul	0.3	40	um	6	um
618	7	um	0.6	um	0.43	ul	0.29	40	um	6	uh
619	37	um	0.41	um	0.4	ul	0.31	40	uh	6	ul
620	11	um	0.63	um	0.48	ul	0.18	40	uh	6	um
621	7	um	0.46	um	0.34	ul	0.21	40	uh	6	uh
622	35	um	0.37	um	0.36	um	0.34	5	ul	6	ul
623	12	um	0.53	um	0.44	um	0.35	5	ul	6	um
624	7	um	0.49	um	0.43	um	0.43	5	ul	6	uh
625	33	um	0.64	um	0.36	um	0.35	5	um	6	ul
626	12	um	0.41	um	0.35	um	0.34	5	um	6	um

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
627	7	um	0.48	um	0.38	um	0.36	5	um	6	uh
628	34	um	0.4	um	0.36	um	0.35	5	uh	6	ul
629	13	um	0.5	um	0.37	um	0.37	5	uh	6	um
630	7	um	0.58	um	0.38	um	0.35	5	uh	6	uh
631	31	um	0.53	um	0.48	um	0.4	20	ul	6	ul
632	10	um	0.55	um	0.37	um	0.35	20	ul	6	um
633	7	um	0.52	um	0.51	um	0.46	20	ul	6	uh
634	38	um	0.62	um	0.52	um	0.35	20	um	6	ul
635	11	um	0.46	um	0.45	um	0.44	20	um	6	um
636	6	um	0.53	um	0.37	um	0.36	20	um	6	uh
637	39	um	0.46	um	0.34	um	0.34	20	uh	6	ul
638	12	um	0.55	um	0.46	um	0.41	20	uh	6	um
639	7	um	0.52	um	0.36	um	0.36	20	uh	6	uh
640	36	um	0.5	um	0.34	um	0.34	40	ul	6	ul
641	12	um	0.43	um	0.34	um	0.34	40	ul	6	um
642	7	um	0.49	um	0.34	um	0.34	40	ul	6	uh
643	33	um	0.4	um	0.36	um	0.35	40	um	6	ul
644	11	um	0.43	um	0.36	um	0.36	40	um	6	um
645	7	um	0.47	um	0.47	um	0.41	40	um	6	uh
646	39	um	0.44	um	0.37	um	0.35	40	uh	6	ul
647	11	um	0.34	um	0.34	um	0.34	40	uh	6	um
648	7	um	0.53	um	0.34	um	0.34	40	uh	6	uh
649	34	uh	0.95	ul	0.08	ul	0.03	5	ul	6	ul
650	11	uh	0.86	ul	0.17	ul	0.13	5	ul	6	um
651	6	uh	0.7	ul	0.05	ul	0.04	5	ul	6	uh
652	33	uh	0.95	ul	0.27	ul	0.15	5	um	6	ul
653	11	uh	0.89	ul	0.25	ul	0.18	5	um	6	um
654	7	uh	0.92	ul	0.04	ul	0.04	5	um	6	uh
655	31	uh	0.85	ul	0.2	ul	0.13	5	uh	6	ul
656	12	uh	0.96	ul	0.16	ul	0.14	5	uh	6	um
657	7	uh	0.97	ul	0.04	ul	0.02	5	uh	6	uh
658	31	uh	0.96	ul	0.05	ul	0.02	20	ul	6	ul
659	12	uh	0.98	ul	0.08	ul	0.06	20	ul	6	um
660	7	uh	0.71	ul	0.29	ul	0.04	20	ul	6	uh
661	34	uh	0.71	ul	0.11	ul	0.03	20	um	6	ul
662	12	uh	0.84	ul	0.26	ul	0.04	20	um	6	um
663	7	uh	0.78	ul	0.15	ul	0.13	20	um	6	uh
664	33	uh	0.9	ul	0.24	ul	0.22	20	uh	6	ul
665	13	uh	0.96	ul	0.3	ul	0.27	20	uh	6	um
666	7	uh	0.89	ul	0.19	ul	0.1	20	uh	6	uh

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
667	39	uh	0.97	ul	0.16	ul	0.07	40	ul	6	ul
668	13	uh	0.71	ul	0.2	ul	0.05	40	ul	6	um
669	7	uh	0.85	ul	0.08	ul	0.04	40	ul	6	uh
670	36	uh	0.84	ul	0.15	ul	0.02	40	um	6	ul
671	11	uh	0.73	ul	0.07	ul	0.03	40	um	6	um
672	7	uh	0.85	ul	0.22	ul	0.09	40	um	6	uh
673	39	uh	0.96	ul	0.04	ul	0.04	40	uh	6	ul
674	13	uh	0.85	ul	0.2	ul	0.09	40	uh	6	um
675	7	uh	0.93	ul	0.28	ul	0.11	40	uh	6	uh
676	36	uh	0.85	um	0.64	ul	0.08	5	ul	6	ul
677	12	uh	0.99	um	0.48	ul	0.23	5	ul	6	um
678	7	uh	0.95	um	0.51	ul	0.24	5	ul	6	uh
679	38	uh	0.91	um	0.64	ul	0.29	5	um	6	ul
680	12	uh	0.92	um	0.51	ul	0.11	5	um	6	um
681	7	uh	0.86	um	0.44	ul	0.06	5	um	6	uh
682	33	uh	0.69	um	0.51	ul	0.24	5	uh	6	ul
683	11	uh	0.67	um	0.52	ul	0.1	5	uh	6	um
684	7	uh	0.73	um	0.39	ul	0.32	5	uh	6	uh
685	35	uh	0.71	um	0.51	ul	0.26	20	ul	6	ul
686	12	uh	0.68	um	0.35	ul	0.04	20	ul	6	um
687	7	uh	0.8	um	0.65	ul	0.13	20	ul	6	uh
688	35	uh	0.8	um	0.5	ul	0.3	20	um	6	ul
689	11	uh	0.8	um	0.67	ul	0.31	20	um	6	um
690	7	uh	0.84	um	0.52	ul	0.31	20	um	6	uh
691	33	uh	0.96	um	0.54	ul	0.15	20	uh	6	ul
692	12	uh	0.76	um	0.44	ul	0.12	20	uh	6	um
693	7	uh	0.71	um	0.46	ul	0.26	20	uh	6	uh
694	35	uh	0.85	um	0.46	ul	0.16	40	ul	6	ul
695	11	uh	0.94	um	0.56	ul	0.28	40	ul	6	um
696	7	uh	0.79	um	0.64	ul	0.12	40	ul	6	uh
697	33	uh	0.78	um	0.55	ul	0.03	40	um	6	ul
698	12	uh	0.99	um	0.36	ul	0.13	40	um	6	um
699	7	uh	0.82	um	0.39	ul	0.05	40	um	6	uh
700	33	uh	0.76	um	0.65	ul	0.16	40	uh	6	ul
701	12	uh	0.93	um	0.55	ul	0.02	40	uh	6	um
702	6	uh	0.78	um	0.62	ul	0.12	40	uh	6	uh
703	32	uh	0.98	um	0.46	um	0.38	5	ul	6	ul
704	12	uh	0.68	um	0.44	um	0.38	5	ul	6	um
705	7	uh	0.99	um	0.64	um	0.37	5	ul	6	uh
706	37	uh	0.84	um	0.56	um	0.4	5	um	6	ul

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
707	11	uh	0.85	um	0.64	um	0.43	5	um	6	um
708	7	uh	0.86	um	0.44	um	0.37	5	um	6	uh
709	35	uh	0.69	um	0.37	um	0.36	5	uh	6	ul
710	11	uh	0.77	um	0.59	um	0.35	5	uh	6	um
711	6	uh	0.9	um	0.41	um	0.41	5	uh	6	uh
712	35	uh	0.7	um	0.51	um	0.39	20	ul	6	ul
713	11	uh	0.69	um	0.34	um	0.34	20	ul	6	um
714	7	uh	0.73	um	0.38	um	0.36	20	ul	6	uh
715	36	uh	0.85	um	0.46	um	0.43	20	um	6	ul
716	12	uh	0.95	um	0.42	um	0.35	20	um	6	um
717	6	uh	0.98	um	0.61	um	0.49	20	um	6	uh
718	33	uh	0.69	um	0.59	um	0.45	20	uh	6	ul
719	12	uh	0.84	um	0.41	um	0.34	20	uh	6	um
720	8	uh	0.76	um	0.47	um	0.41	20	uh	6	uh
721	37	uh	0.74	um	0.35	um	0.35	40	ul	6	ul
722	12	uh	0.85	um	0.45	um	0.38	40	ul	6	um
723	7	uh	0.73	um	0.57	um	0.48	40	ul	6	uh
724	33	uh	0.76	um	0.63	um	0.5	40	um	6	ul
725	12	uh	0.67	um	0.46	um	0.35	40	um	6	um
726	7	uh	0.67	um	0.38	um	0.37	40	um	6	uh
727	33	uh	0.67	um	0.64	um	0.46	40	uh	6	ul
728	12	uh	0.71	um	0.57	um	0.46	40	uh	6	um
729	7	uh	0.99	um	0.37	um	0.35	40	uh	6	uh
730	35	uh	0.94	uh	0.94	ul	0.25	5	ul	6	ul
731	12	uh	0.77	uh	0.77	ul	0.2	5	ul	6	um
732	7	uh	0.81	uh	0.71	ul	0.21	5	ul	6	uh
733	34	uh	0.7	uh	0.68	ul	0.19	5	um	6	ul
734	11	uh	0.78	uh	0.78	ul	0.32	5	um	6	um
735	6	uh	0.76	uh	0.73	ul	0.05	5	um	6	uh
736	31	uh	0.67	uh	0.67	ul	0.07	5	uh	6	ul
737	12	uh	0.99	uh	0.69	ul	0.06	5	uh	6	um
738	7	uh	0.88	uh	0.85	ul	0.31	5	uh	6	uh
739	42	uh	0.91	uh	0.85	ul	0.25	20	ul	6	ul
740	11	uh	0.73	uh	0.68	ul	0.25	20	ul	6	um
741	6	uh	0.7	uh	0.67	ul	0.32	20	ul	6	uh
742	40	uh	0.97	uh	0.74	ul	0.19	20	um	6	ul
743	11	uh	0.91	uh	0.76	ul	0.19	20	um	6	um
744	7	uh	0.93	uh	0.86	ul	0.22	20	um	6	uh
745	32	uh	0.82	uh	0.73	ul	0.03	20	uh	6	ul
746	11	uh	0.76	uh	0.67	ul	0.12	20	uh	6	um

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
747	6	uh	0.89	uh	0.82	ul	0.16	20	uh	6	uh
748	29	uh	0.68	uh	0.67	ul	0.05	40	ul	6	ul
749	12	uh	0.79	uh	0.78	ul	0.22	40	ul	6	um
750	6	uh	0.84	uh	0.67	ul	0.09	40	ul	6	uh
751	31	uh	0.86	uh	0.76	ul	0.2	40	um	6	ul
752	11	uh	0.69	uh	0.67	ul	0.04	40	um	6	um
753	7	uh	0.97	uh	0.89	ul	0.18	40	um	6	uh
754	35	uh	0.71	uh	0.68	ul	0.05	40	uh	6	ul
755	11	uh	0.99	uh	0.93	ul	0.12	40	uh	6	um
756	7	uh	0.83	uh	0.83	ul	0.12	40	uh	6	uh
757	34	uh	0.77	uh	0.76	um	0.48	5	ul	6	ul
758	12	uh	0.92	uh	0.8	um	0.56	5	ul	6	um
759	7	uh	0.75	uh	0.7	um	0.53	5	ul	6	uh
760	35	uh	0.74	uh	0.73	um	0.63	5	um	6	ul
761	11	uh	0.99	uh	0.82	um	0.59	5	um	6	um
762	7	uh	0.85	uh	0.82	um	0.48	5	um	6	uh
763	25	uh	0.86	uh	0.68	um	0.57	5	uh	6	ul
764	11	uh	0.94	uh	0.85	um	0.65	5	uh	6	um
765	7	uh	0.77	uh	0.72	um	0.56	5	uh	6	uh
766	31	uh	0.83	uh	0.82	um	0.59	20	ul	6	ul
767	10	uh	0.67	uh	0.67	um	0.48	20	ul	6	um
768	7	uh	0.94	uh	0.7	um	0.54	20	ul	6	uh
769	32	uh	0.9	uh	0.71	um	0.51	20	um	6	ul
770	12	uh	0.85	uh	0.74	um	0.62	20	um	6	um
771	7	uh	0.9	uh	0.67	um	0.46	20	um	6	uh
772	33	uh	0.68	uh	0.68	um	0.65	20	uh	6	ul
773	12	uh	0.78	uh	0.73	um	0.57	20	uh	6	um
774	7	uh	0.88	uh	0.73	um	0.67	20	uh	6	uh
775	33	uh	0.75	uh	0.74	um	0.37	40	ul	6	ul
776	12	uh	0.98	uh	0.97	um	0.36	40	ul	6	um
777	7	uh	0.75	uh	0.71	um	0.36	40	ul	6	uh
778	32	uh	0.7	uh	0.7	um	0.5	40	um	6	ul
779	12	uh	0.69	uh	0.69	um	0.55	40	um	6	um
780	7	uh	0.69	uh	0.67	um	0.62	40	um	6	uh
781	34	uh	0.83	uh	0.67	um	0.58	40	uh	6	ul
782	11	uh	0.91	uh	0.71	um	0.63	40	uh	6	um
783	7	uh	0.88	uh	0.76	um	0.37	40	uh	6	uh
784	36	uh	0.92	uh	0.86	uh	0.67	5	ul	6	ul
785	13	uh	0.86	uh	0.76	uh	0.69	5	ul	6	um
786	7	uh	0.93	uh	0.79	uh	0.79	5	ul	6	uh

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
787	36	uh	0.74	uh	0.71	uh	0.69	5	um	6	ul
788	12	uh	0.92	uh	0.77	uh	0.71	5	um	6	um
789	7	uh	0.87	uh	0.74	uh	0.72	5	um	6	uh
790	35	uh	0.9	uh	0.76	uh	0.7	5	uh	6	ul
791	12	uh	0.95	uh	0.94	uh	0.81	5	uh	6	um
792	7	uh	0.8	uh	0.79	uh	0.7	5	uh	6	uh
793	32	uh	0.85	uh	0.85	uh	0.79	20	ul	6	ul
794	11	uh	0.67	uh	0.67	uh	0.67	20	ul	6	um
795	7	uh	0.97	uh	0.91	uh	0.9	20	ul	6	uh
796	41	uh	0.86	uh	0.77	uh	0.74	20	um	6	ul
797	11	uh	0.74	uh	0.74	uh	0.72	20	um	6	um
798	7	uh	0.99	uh	0.98	uh	0.96	20	um	6	uh
799	38	uh	0.76	uh	0.67	uh	0.67	20	uh	6	ul
800	13	uh	0.82	uh	0.8	uh	0.77	20	uh	6	um
801	7	uh	0.96	uh	0.8	uh	0.8	20	uh	6	uh
802	33	uh	0.81	uh	0.68	uh	0.68	40	ul	6	ul
803	10	uh	0.7	uh	0.67	uh	0.67	40	ul	6	um
804	7	uh	0.71	uh	0.67	uh	0.67	40	ul	6	uh
805	41	uh	0.79	uh	0.74	uh	0.68	40	um	6	ul
806	11	uh	0.92	uh	0.68	uh	0.68	40	um	6	um
807	7	uh	0.67	uh	0.67	uh	0.67	40	um	6	uh
808	35	uh	0.95	uh	0.92	uh	0.76	40	uh	6	ul
809	11	uh	0.74	uh	0.67	uh	0.67	40	uh	6	um
810	6	uh	0.81	uh	0.75	uh	0.68	40	uh	6	uh
811	47	ul	0.08	ul	0.01	ul	0.01	5	ul	8	ul
812	15	ul	0.09	ul	0.03	ul	0.03	5	ul	8	um
813	10	ul	0.24	ul	0.05	ul	0.04	5	ul	8	uh
814	47	ul	0.13	ul	0.05	ul	0.01	5	um	8	ul
815	15	ul	0.25	ul	0.08	ul	0.01	5	um	8	um
816	9	ul	0.32	ul	0.3	ul	0.17	5	um	8	uh
817	47	ul	0.09	ul	0.02	ul	0.01	5	uh	8	ul
818	15	ul	0.2	ul	0.17	ul	0.04	5	uh	8	um
819	9	ul	0.23	ul	0.02	ul	0.02	5	uh	8	uh
820	47	ul	0.07	ul	0.04	ul	0.03	20	ul	8	ul
821	15	ul	0.1	ul	0.03	ul	0.01	20	ul	8	um
822	10	ul	0.08	ul	0.03	ul	0.03	20	ul	8	uh
823	49	ul	0.16	ul	0.09	ul	0.08	20	um	8	ul
824	16	ul	0.33	ul	0.31	ul	0.29	20	um	8	um
825	9	ul	0.2	ul	0.19	ul	0.04	20	um	8	uh
826	41	ul	0.28	ul	0.11	ul	0.07	20	uh	8	ul

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
827	16	ul	0.25	ul	0.19	ul	0.1	20	uh	8	um
828	10	ul	0.2	ul	0.13	ul	0.05	20	uh	8	uh
829	42	ul	0.3	ul	0.09	ul	0.01	40	ul	8	ul
830	16	ul	0.07	ul	0.07	ul	0.06	40	ul	8	um
831	9	ul	0.31	ul	0.18	ul	0.08	40	ul	8	uh
832	47	ul	0.05	ul	0.03	ul	0.01	40	um	8	ul
833	16	ul	0.32	ul	0.17	ul	0.17	40	um	8	um
834	9	ul	0.3	ul	0.17	ul	0.12	40	um	8	uh
835	50	ul	0.19	ul	0.14	ul	0.09	40	uh	8	ul
836	15	ul	0.26	ul	0.05	ul	0.01	40	uh	8	um
837	10	ul	0.1	ul	0.05	ul	0.01	40	uh	8	uh
838	48	um	0.48	ul	0.33	ul	0.14	5	ul	8	ul
839	16	um	0.42	ul	0.32	ul	0.03	5	ul	8	um
840	9	um	0.63	ul	0.19	ul	0.16	5	ul	8	uh
841	49	um	0.5	ul	0.05	ul	0.01	5	um	8	ul
842	16	um	0.62	ul	0.2	ul	0.18	5	um	8	um
843	9	um	0.5	ul	0.12	ul	0.12	5	um	8	uh
844	43	um	0.54	ul	0.19	ul	0.1	5	uh	8	ul
845	16	um	0.66	ul	0.27	ul	0.01	5	uh	8	um
846	9	um	0.57	ul	0.33	ul	0.1	5	uh	8	uh
847	52	um	0.61	ul	0.26	ul	0.18	20	ul	8	ul
848	14	um	0.41	ul	0.26	ul	0.23	20	ul	8	um
849	9	um	0.6	ul	0.13	ul	0.04	20	ul	8	uh
850	44	um	0.54	ul	0.16	ul	0.14	20	um	8	ul
851	15	um	0.48	ul	0.07	ul	0.06	20	um	8	um
852	10	um	0.6	ul	0.04	ul	0.01	20	um	8	uh
853	48	um	0.59	ul	0.2	ul	0.07	20	uh	8	ul
854	17	um	0.34	ul	0.26	ul	0.18	20	uh	8	um
855	10	um	0.44	ul	0.22	ul	0.21	20	uh	8	uh
856	48	um	0.49	ul	0.15	ul	0.12	40	ul	8	ul
857	16	um	0.66	ul	0.17	ul	0.11	40	ul	8	um
858	9	um	0.55	ul	0.03	ul	0.02	40	ul	8	uh
859	46	um	0.37	ul	0.1	ul	0.08	40	um	8	ul
860	16	um	0.46	ul	0.12	ul	0.01	40	um	8	um
861	9	um	0.54	ul	0.17	ul	0.09	40	um	8	uh
862	45	um	0.39	ul	0.18	ul	0.07	40	uh	8	ul
863	15	um	0.47	ul	0.08	ul	0.03	40	uh	8	um
864	9	um	0.36	ul	0.02	ul	0.02	40	uh	8	uh
865	46	um	0.42	um	0.42	ul	0.24	5	ul	8	ul
866	15	um	0.43	um	0.4	ul	0.31	5	ul	8	um

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
867	9	um	0.48	um	0.44	ul	0.32	5	ul	8	uh
868	51	um	0.61	um	0.37	ul	0.01	5	um	8	ul
869	16	um	0.37	um	0.37	ul	0.11	5	um	8	um
870	8	um	0.43	um	0.42	ul	0.16	5	um	8	uh
871	49	um	0.37	um	0.34	ul	0.01	5	uh	8	ul
872	14	um	0.46	um	0.36	ul	0.26	5	uh	8	um
873	10	um	0.4	um	0.36	ul	0.08	5	uh	8	uh
874	44	um	0.59	um	0.48	ul	0.15	20	ul	8	ul
875	16	um	0.51	um	0.39	ul	0.3	20	ul	8	um
876	10	um	0.48	um	0.48	ul	0.1	20	ul	8	uh
877	42	um	0.62	um	0.55	ul	0.08	20	um	8	ul
878	16	um	0.55	um	0.51	ul	0.31	20	um	8	um
879	9	um	0.45	um	0.38	ul	0.05	20	um	8	uh
880	54	um	0.43	um	0.37	ul	0.13	20	uh	8	ul
881	15	um	0.54	um	0.37	ul	0.06	20	uh	8	um
882	10	um	0.6	um	0.57	ul	0.13	20	uh	8	uh
883	49	um	0.65	um	0.54	ul	0.24	40	ul	8	ul
884	17	um	0.5	um	0.5	ul	0.14	40	ul	8	um
885	9	um	0.47	um	0.42	ul	0.24	40	ul	8	uh
886	48	um	0.37	um	0.36	ul	0.08	40	um	8	ul
887	15	um	0.66	um	0.41	ul	0.29	40	um	8	um
888	9	um	0.4	um	0.4	ul	0.01	40	um	8	uh
889	48	um	0.5	um	0.37	ul	0.07	40	uh	8	ul
890	16	um	0.44	um	0.42	ul	0.1	40	uh	8	um
891	9	um	0.64	um	0.37	ul	0.26	40	uh	8	uh
892	51	um	0.52	um	0.36	um	0.34	5	ul	8	ul
893	16	um	0.6	um	0.58	um	0.53	5	ul	8	um
894	9	um	0.41	um	0.34	um	0.34	5	ul	8	uh
895	52	um	0.66	um	0.49	um	0.47	5	um	8	ul
896	16	um	0.46	um	0.38	um	0.35	5	um	8	um
897	9	um	0.34	um	0.34	um	0.34	5	um	8	uh
898	43	um	0.57	um	0.38	um	0.35	5	uh	8	ul
899	15	um	0.65	um	0.65	um	0.58	5	uh	8	um
900	9	um	0.5	um	0.48	um	0.4	5	uh	8	uh
901	46	um	0.51	um	0.36	um	0.34	20	ul	8	ul
902	17	um	0.47	um	0.35	um	0.35	20	ul	8	um
903	9	um	0.58	um	0.43	um	0.38	20	ul	8	uh
904	42	um	0.59	um	0.34	um	0.34	20	um	8	ul
905	15	um	0.63	um	0.6	um	0.54	20	um	8	um
906	9	um	0.37	um	0.36	um	0.34	20	um	8	uh

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
907	48	um	0.44	um	0.39	um	0.36	20	uh	8	ul
908	15	um	0.61	um	0.59	um	0.4	20	uh	8	um
909	9	um	0.35	um	0.35	um	0.35	20	uh	8	uh
910	53	um	0.57	um	0.42	um	0.38	40	ul	8	ul
911	14	um	0.41	um	0.39	um	0.38	40	ul	8	um
912	9	um	0.54	um	0.53	um	0.36	40	ul	8	uh
913	48	um	0.47	um	0.4	um	0.39	40	um	8	ul
914	15	um	0.54	um	0.43	um	0.38	40	um	8	um
915	9	um	0.56	um	0.42	um	0.35	40	um	8	uh
916	54	um	0.4	um	0.4	um	0.38	40	uh	8	ul
917	16	um	0.34	um	0.34	um	0.34	40	uh	8	um
918	9	um	0.37	um	0.37	um	0.36	40	uh	8	uh
919	49	uh	0.82	ul	0.26	ul	0.22	5	ul	8	ul
920	16	uh	0.81	ul	0.26	ul	0.18	5	ul	8	um
921	9	uh	0.96	ul	0.01	ul	0.01	5	ul	8	uh
922	45	uh	0.98	ul	0.05	ul	0.05	5	um	8	ul
923	14	uh	0.88	ul	0.07	ul	0.07	5	um	8	um
924	10	uh	0.88	ul	0.17	ul	0.13	5	um	8	uh
925	49	uh	0.91	ul	0.13	ul	0.03	5	uh	8	ul
926	15	uh	0.95	ul	0.22	ul	0.12	5	uh	8	um
927	9	uh	0.97	ul	0.21	ul	0.01	5	uh	8	uh
928	49	uh	0.75	ul	0.29	ul	0.24	20	ul	8	ul
929	15	uh	0.79	ul	0.08	ul	0.04	20	ul	8	um
930	9	uh	0.93	ul	0.08	ul	0.05	20	ul	8	uh
931	41	uh	0.91	ul	0.22	ul	0.08	20	um	8	ul
932	14	uh	0.69	ul	0.3	ul	0.04	20	um	8	um
933	9	uh	0.82	ul	0.27	ul	0.03	20	um	8	uh
934	52	uh	0.91	ul	0.34	ul	0.34	20	uh	8	ul
935	15	uh	0.88	ul	0.32	ul	0.03	20	uh	8	um
936	9	uh	0.78	ul	0.17	ul	0.17	20	uh	8	uh
937	46	uh	0.7	ul	0.19	ul	0.12	40	ul	8	ul
938	18	uh	0.78	ul	0.27	ul	0.14	40	ul	8	um
939	9	uh	0.69	ul	0.27	ul	0.16	40	ul	8	uh
940	44	uh	0.95	ul	0.21	ul	0.08	40	um	8	ul
941	16	uh	0.96	ul	0.19	ul	0.1	40	um	8	um
942	9	uh	0.81	ul	0.07	ul	0.01	40	um	8	uh
943	43	uh	0.84	ul	0.05	ul	0.05	40	uh	8	ul
944	15	uh	0.68	ul	0.29	ul	0.2	40	uh	8	um
945	9	uh	0.74	ul	0.11	ul	0.07	40	uh	8	uh
946	43	uh	0.91	um	0.51	ul	0.26	5	ul	8	ul

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
947	15	uh	0.99	um	0.46	ul	0.15	5	ul	8	um
948	9	uh	0.84	um	0.66	ul	0.01	5	ul	8	uh
949	40	uh	0.67	um	0.34	ul	0.27	5	um	8	ul
950	16	uh	0.78	um	0.41	ul	0.33	5	um	8	um
951	9	uh	0.82	um	0.37	ul	0.32	5	um	8	uh
952	42	uh	0.79	um	0.39	ul	0.33	5	uh	8	ul
953	16	uh	0.95	um	0.66	ul	0.31	5	uh	8	um
954	9	uh	0.97	um	0.64	ul	0.02	5	uh	8	uh
955	40	uh	0.75	um	0.47	ul	0.04	20	ul	8	ul
956	15	uh	0.8	um	0.57	ul	0.1	20	ul	8	um
957	10	uh	0.69	um	0.52	ul	0.13	20	ul	8	uh
958	45	uh	0.72	um	0.4	ul	0.11	20	um	8	ul
959	15	uh	0.79	um	0.35	ul	0.07	20	um	8	um
960	10	uh	0.96	um	0.58	ul	0.09	20	um	8	uh
961	44	uh	0.89	um	0.5	ul	0.31	20	uh	8	ul
962	16	uh	0.85	um	0.67	ul	0.22	20	uh	8	um
963	9	uh	0.7	um	0.49	ul	0.09	20	uh	8	uh
964	48	uh	0.67	um	0.36	ul	0.33	40	ul	8	ul
965	16	uh	0.82	um	0.6	ul	0.08	40	ul	8	um
966	9	uh	0.93	um	0.59	ul	0.19	40	ul	8	uh
967	45	uh	0.79	um	0.44	ul	0.19	40	um	8	ul
968	14	uh	0.96	um	0.48	ul	0.02	40	um	8	um
969	10	uh	0.67	um	0.5	ul	0.34	40	um	8	uh
970	50	uh	0.98	um	0.5	ul	0.29	40	uh	8	ul
971	16	uh	0.68	um	0.34	ul	0.23	40	uh	8	um
972	9	uh	0.95	um	0.47	ul	0.18	40	uh	8	uh
973	50	uh	0.75	um	0.6	um	0.59	5	ul	8	ul
974	14	uh	0.83	um	0.35	um	0.35	5	ul	8	um
975	9	uh	0.93	um	0.57	um	0.48	5	ul	8	uh
976	47	uh	0.91	um	0.6	um	0.53	5	um	8	ul
977	16	uh	0.67	um	0.52	um	0.36	5	um	8	um
978	9	uh	0.69	um	0.61	um	0.56	5	um	8	uh
979	43	uh	0.82	um	0.53	um	0.52	5	uh	8	ul
980	16	uh	0.9	um	0.44	um	0.41	5	uh	8	um
981	9	uh	0.67	um	0.55	um	0.36	5	uh	8	uh
982	38	uh	0.97	um	0.49	um	0.36	20	ul	8	ul
983	15	uh	0.79	um	0.49	um	0.38	20	ul	8	um
984	9	uh	0.8	um	0.58	um	0.36	20	ul	8	uh
985	46	uh	0.68	um	0.44	um	0.44	20	um	8	ul
986	15	uh	0.9	um	0.51	um	0.34	20	um	8	um

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
987	9	uh	0.71	um	0.62	um	0.35	20	um	8	uh
988	50	uh	0.74	um	0.45	um	0.4	20	uh	8	ul
989	16	uh	0.7	um	0.51	um	0.48	20	uh	8	um
990	8	uh	0.76	um	0.39	um	0.34	20	uh	8	uh
991	43	uh	0.96	um	0.39	um	0.35	40	ul	8	ul
992	15	uh	0.77	um	0.52	um	0.38	40	ul	8	um
993	10	uh	0.93	um	0.56	um	0.47	40	ul	8	uh
994	48	uh	0.81	um	0.34	um	0.34	40	um	8	ul
995	16	uh	0.86	um	0.5	um	0.36	40	um	8	um
996	9	uh	0.75	um	0.59	um	0.34	40	um	8	uh
997	41	uh	0.69	um	0.37	um	0.36	40	uh	8	ul
998	14	uh	0.69	um	0.65	um	0.51	40	uh	8	um
999	9	uh	0.98	um	0.42	um	0.34	40	uh	8	uh
1000	45	uh	0.71	uh	0.68	ul	0.26	5	ul	8	ul
1001	16	uh	0.72	uh	0.72	ul	0.31	5	ul	8	um
1002	9	uh	0.75	uh	0.69	ul	0.28	5	ul	8	uh
1003	48	uh	0.98	uh	0.89	ul	0.24	5	um	8	ul
1004	17	uh	0.84	uh	0.71	ul	0.06	5	um	8	um
1005	9	uh	0.86	uh	0.81	ul	0.22	5	um	8	uh
1006	42	uh	0.67	uh	0.67	ul	0.14	5	uh	8	ul
1007	16	uh	0.82	uh	0.67	ul	0.33	5	uh	8	um
1008	9	uh	0.79	uh	0.73	ul	0.08	5	uh	8	uh
1009	49	uh	0.77	uh	0.72	ul	0.08	20	ul	8	ul
1010	16	uh	0.88	uh	0.7	ul	0.09	20	ul	8	um
1011	9	uh	0.7	uh	0.69	ul	0.17	20	ul	8	uh
1012	46	uh	0.78	uh	0.74	ul	0.03	20	um	8	ul
1013	15	uh	0.94	uh	0.92	ul	0.21	20	um	8	um
1014	9	uh	0.8	uh	0.73	ul	0.25	20	um	8	uh
1015	47	uh	0.92	uh	0.9	ul	0.27	20	uh	8	ul
1016	15	uh	0.77	uh	0.73	ul	0.22	20	uh	8	um
1017	9	uh	0.92	uh	0.71	ul	0.1	20	uh	8	uh
1018	45	uh	0.83	uh	0.7	ul	0.09	40	ul	8	ul
1019	15	uh	0.82	uh	0.82	ul	0.31	40	ul	8	um
1020	9	uh	0.89	uh	0.84	ul	0.3	40	ul	8	uh
1021	43	uh	0.81	uh	0.69	ul	0.31	40	um	8	ul
1022	16	uh	0.96	uh	0.81	ul	0.31	40	um	8	um
1023	9	uh	0.82	uh	0.71	ul	0.17	40	um	8	uh
1024	53	uh	0.97	uh	0.94	ul	0.07	40	uh	8	ul
1025	17	uh	0.99	uh	0.69	ul	0.2	40	uh	8	um
1026	9	uh	0.76	uh	0.75	ul	0.11	40	uh	8	uh

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
1027	47	uh	0.79	uh	0.72	um	0.6	5	ul	8	ul
1028	15	uh	0.95	uh	0.84	um	0.59	5	ul	8	um
1029	9	uh	0.86	uh	0.86	um	0.45	5	ul	8	uh
1030	44	uh	0.72	uh	0.69	um	0.59	5	um	8	ul
1031	15	uh	0.71	uh	0.71	um	0.49	5	um	8	um
1032	9	uh	0.88	uh	0.84	um	0.36	5	um	8	uh
1033	39	uh	0.68	uh	0.67	um	0.48	5	uh	8	ul
1034	16	uh	0.91	uh	0.9	um	0.49	5	uh	8	um
1035	9	uh	0.87	uh	0.86	um	0.38	5	uh	8	uh
1036	49	uh	0.71	uh	0.67	um	0.47	20	ul	8	ul
1037	16	uh	0.84	uh	0.75	um	0.61	20	ul	8	um
1038	9	uh	0.85	uh	0.7	um	0.66	20	ul	8	uh
1039	51	uh	0.77	uh	0.76	um	0.43	20	um	8	ul
1040	15	uh	0.92	uh	0.82	um	0.46	20	um	8	um
1041	9	uh	0.86	uh	0.67	um	0.58	20	um	8	uh
1042	43	uh	0.94	uh	0.83	um	0.63	20	uh	8	ul
1043	16	uh	0.68	uh	0.67	um	0.39	20	uh	8	um
1044	10	uh	0.9	uh	0.84	um	0.35	20	uh	8	uh
1045	48	uh	0.77	uh	0.69	um	0.6	40	ul	8	ul
1046	16	uh	0.9	uh	0.84	um	0.64	40	ul	8	um
1047	10	uh	0.97	uh	0.74	um	0.4	40	ul	8	uh
1048	41	uh	0.76	uh	0.74	um	0.37	40	um	8	ul
1049	15	uh	0.92	uh	0.71	um	0.6	40	um	8	um
1050	9	uh	0.82	uh	0.76	um	0.37	40	um	8	uh
1051	50	uh	0.98	uh	0.68	um	0.37	40	uh	8	ul
1052	16	uh	0.7	uh	0.7	um	0.46	40	uh	8	um
1053	9	uh	0.9	uh	0.84	um	0.47	40	uh	8	uh
1054	42	uh	0.98	uh	0.81	uh	0.75	5	ul	8	ul
1055	16	uh	0.78	uh	0.78	uh	0.72	5	ul	8	um
1056	9	uh	0.98	uh	0.78	uh	0.67	5	ul	8	uh
1057	44	uh	0.96	uh	0.72	uh	0.7	5	um	8	ul
1058	14	uh	0.73	uh	0.7	uh	0.7	5	um	8	um
1059	9	uh	0.99	uh	0.84	uh	0.79	5	um	8	uh
1060	47	uh	0.92	uh	0.78	uh	0.74	5	uh	8	ul
1061	15	uh	0.82	uh	0.79	uh	0.68	5	uh	8	um
1062	9	uh	0.82	uh	0.7	uh	0.7	5	uh	8	uh
1063	50	uh	0.67	uh	0.67	uh	0.67	20	ul	8	ul
1064	15	uh	0.94	uh	0.91	uh	0.83	20	ul	8	um
1065	9	uh	0.77	uh	0.69	uh	0.68	20	ul	8	uh
1066	57	uh	0.88	uh	0.75	uh	0.69	20	um	8	ul

ID	no_tasks	total_tx_dis	$total_{Tx}$	max_tx_dis	max_{Tx}	min_tx_dis	min_{Tx}	total_no_obj	no_obj_tx_dis	$\hat{U}$	u_i_dist
1067	15	uh	0.7	uh	0.7	uh	0.68	20	um	8	um
1068	10	uh	0.7	uh	0.67	uh	0.67	20	um	8	uh
1069	43	uh	0.86	uh	0.75	uh	0.73	20	uh	8	ul
1070	15	uh	0.81	uh	0.68	uh	0.68	20	uh	8	um
1071	9	uh	0.85	uh	0.72	uh	0.71	20	uh	8	uh
1072	46	uh	0.74	uh	0.72	uh	0.7	40	ul	8	ul
1073	15	uh	0.7	uh	0.7	uh	0.67	40	ul	8	um
1074	9	uh	0.88	uh	0.83	uh	0.74	40	ul	8	uh
1075	49	uh	0.84	uh	0.73	uh	0.71	40	um	8	ul
1076	15	uh	0.82	uh	0.71	uh	0.69	40	um	8	um
1077	9	uh	0.74	uh	0.7	uh	0.67	40	um	8	uh
1078	46	uh	0.77	uh	0.75	uh	0.7	40	uh	8	ul
1079	16	uh	0.87	uh	0.8	uh	0.78	40	uh	8	um
1080	9	uh	0.75	uh	0.71	uh	0.71	40	uh	8	uh

Appendix B

Complete Deadline Satisfaction Results

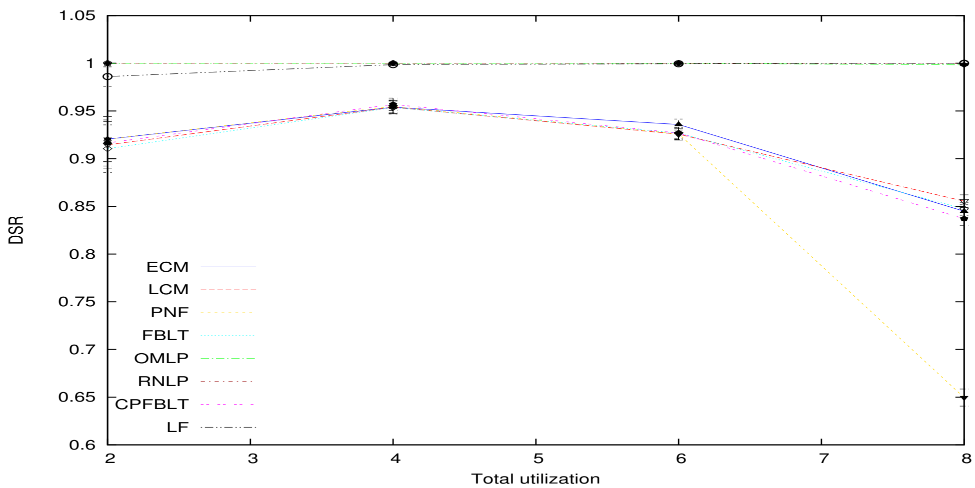


Figure B.1: DSR for Tasksets 1, 271, 541 and 811

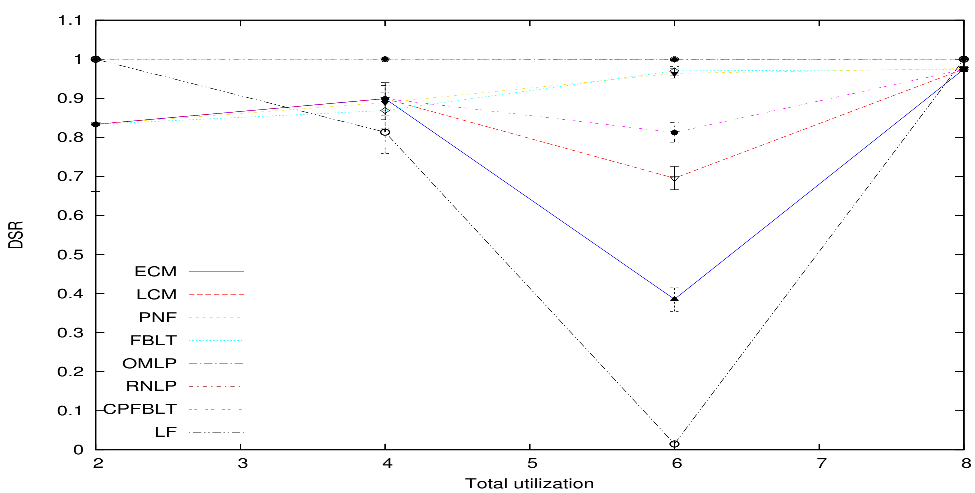


Figure B.2: DSR for Tasksets 2, 272, 542 and 812

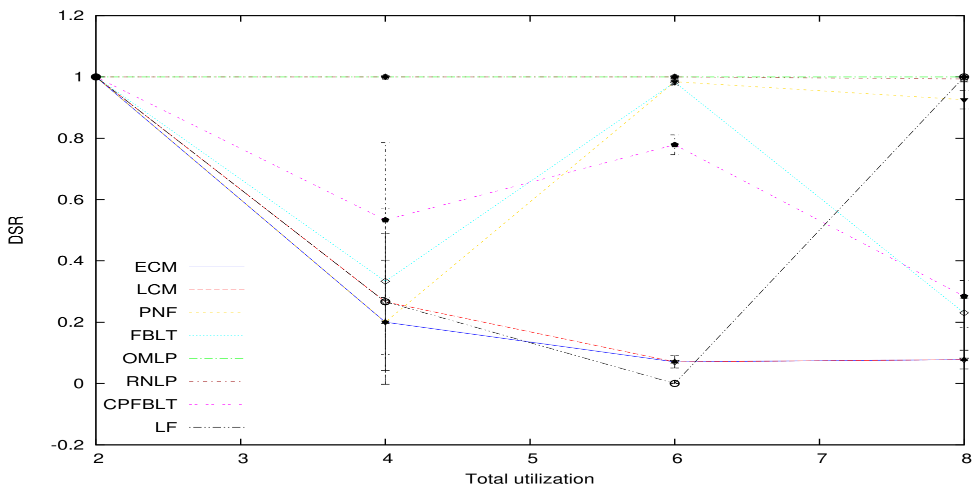


Figure B.3: DSR for Tasksets 3, 273, 543 and 813

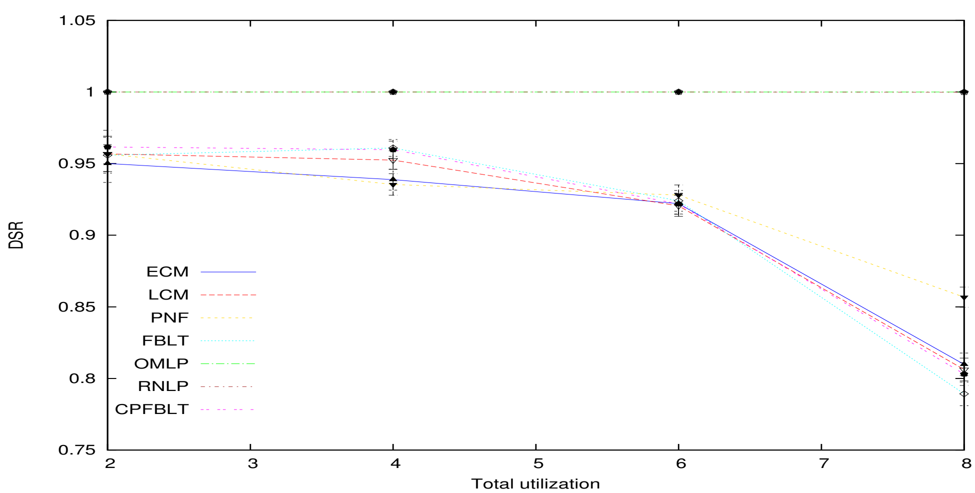


Figure B.4: DSR for Tasksets 4, 274, 544 and 814

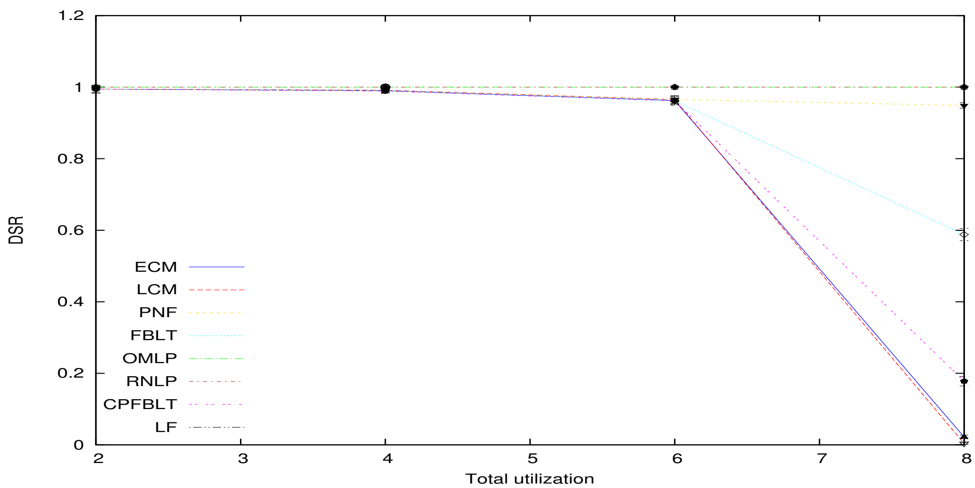


Figure B.5: DSR for Tasksets 5, 275, 545 and 815

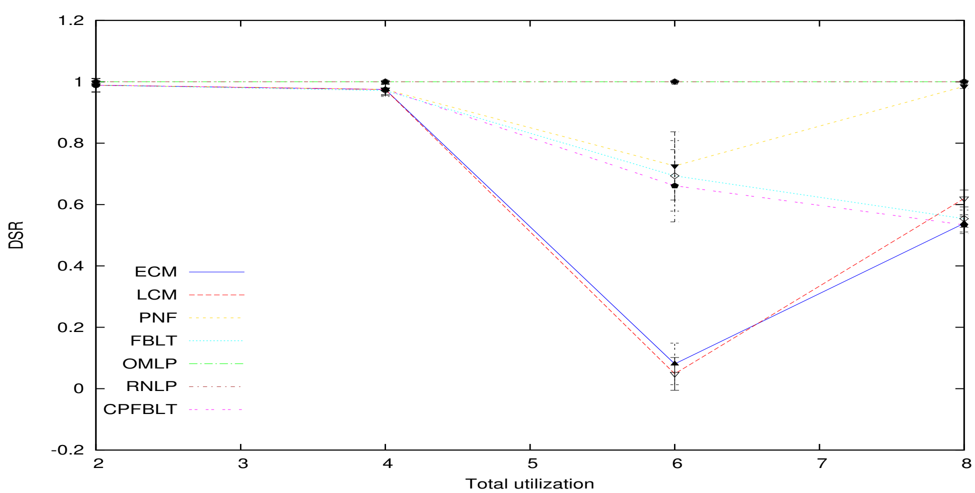


Figure B.6: DSR for Tasksets 6, 276, 546 and 816

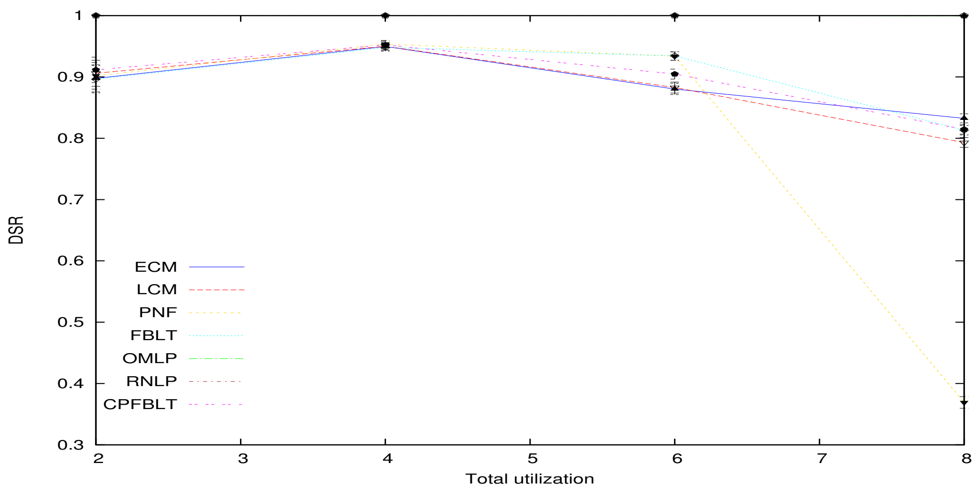


Figure B.7: DSR for Tasksets 7, 277, 547 and 817

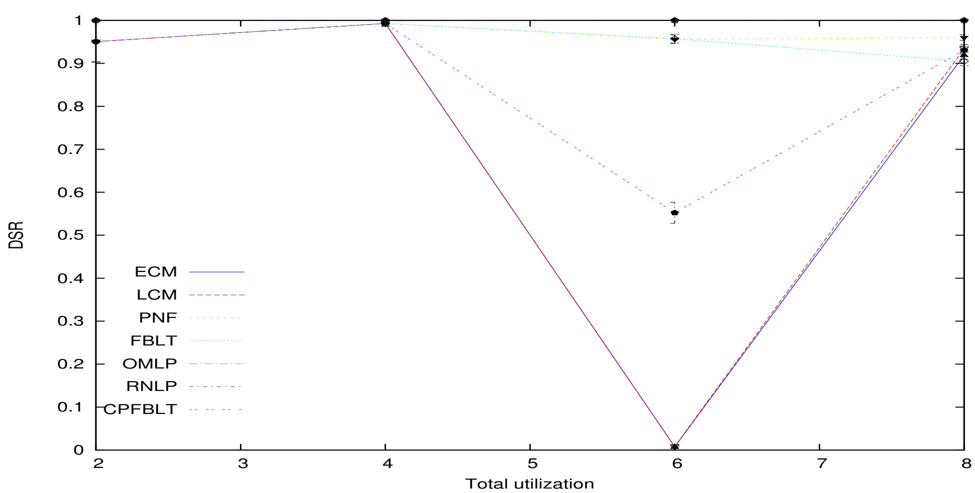


Figure B.8: DSR for Tasksets 8, 278, 548 and 818

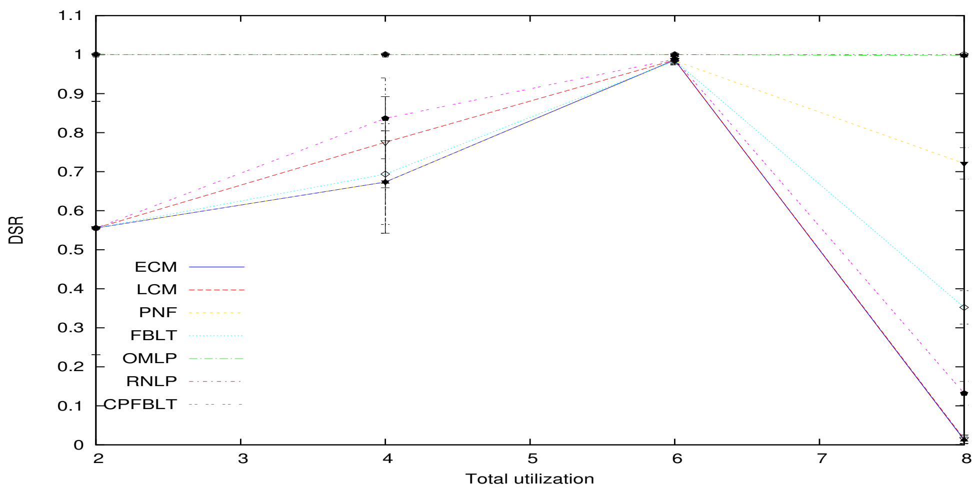


Figure B.9: DSR for Tasksets 9, 279, 549 and 819

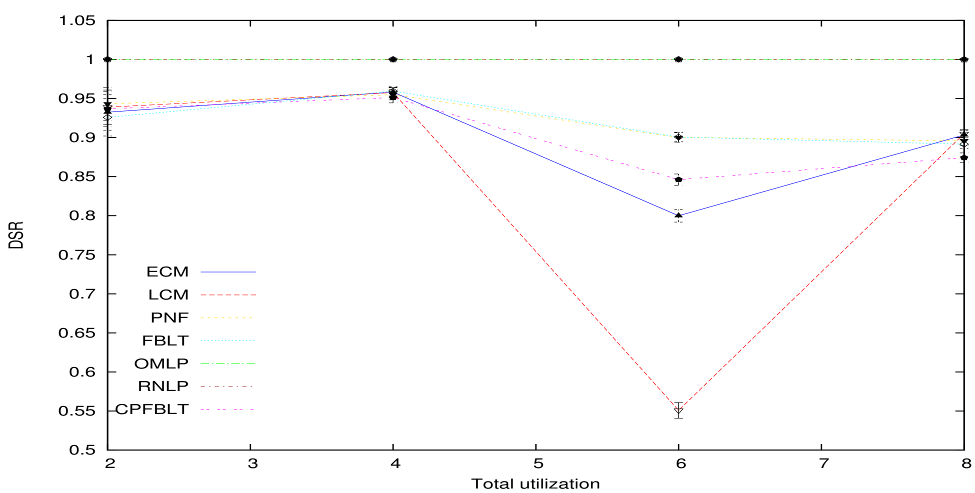


Figure B.10: DSR for Tasksets 10, 280, 550 and 820

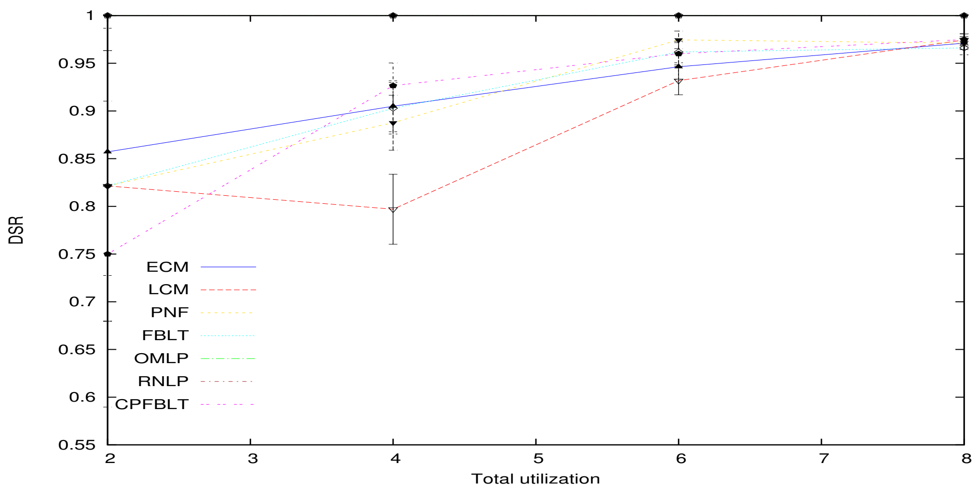


Figure B.11: DSR for Tasksets 11, 281, 551 and 821

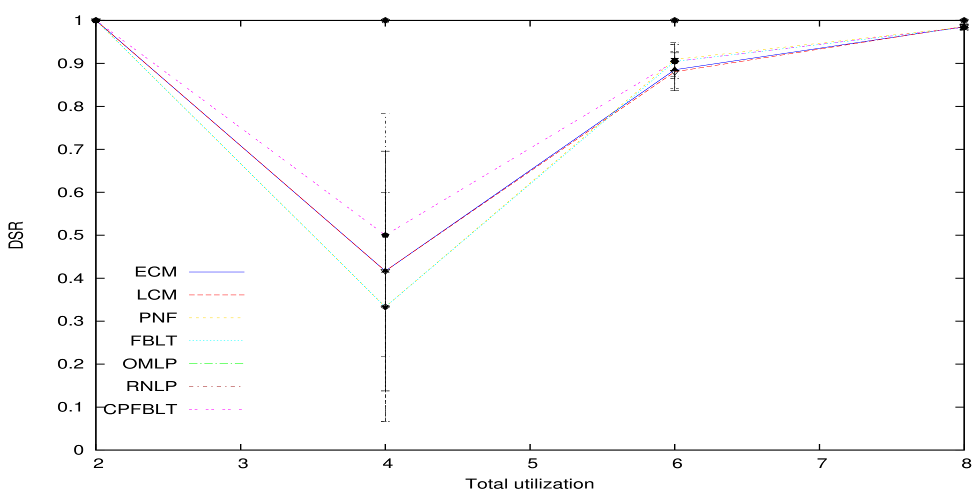


Figure B.12: DSR for Tasksets 12, 282, 552 and 822

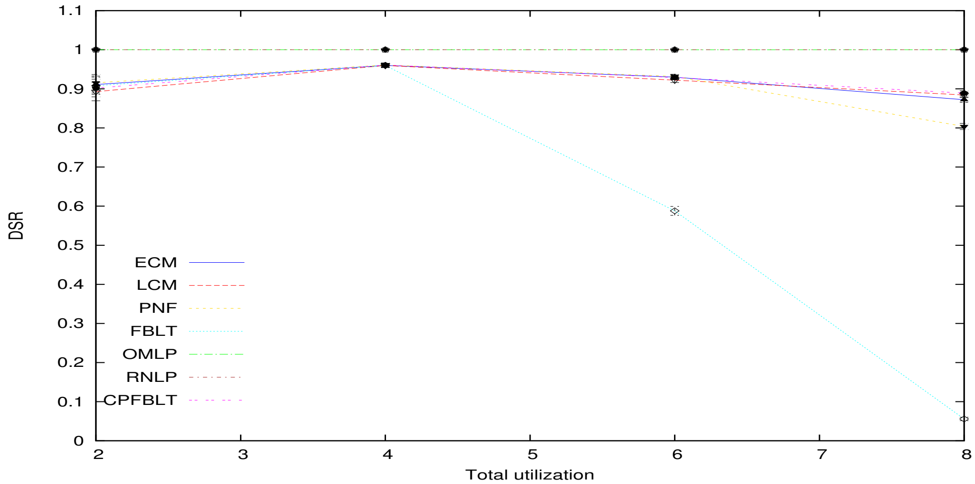


Figure B.13: DSR for Tasksets 13, 283, 553 and 823

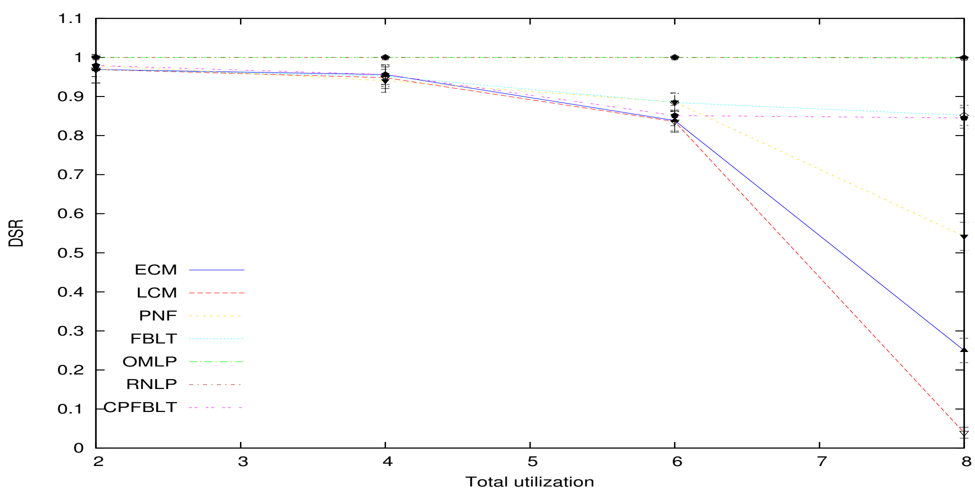


Figure B.14: DSR for Tasksets 14, 284, 554 and 824

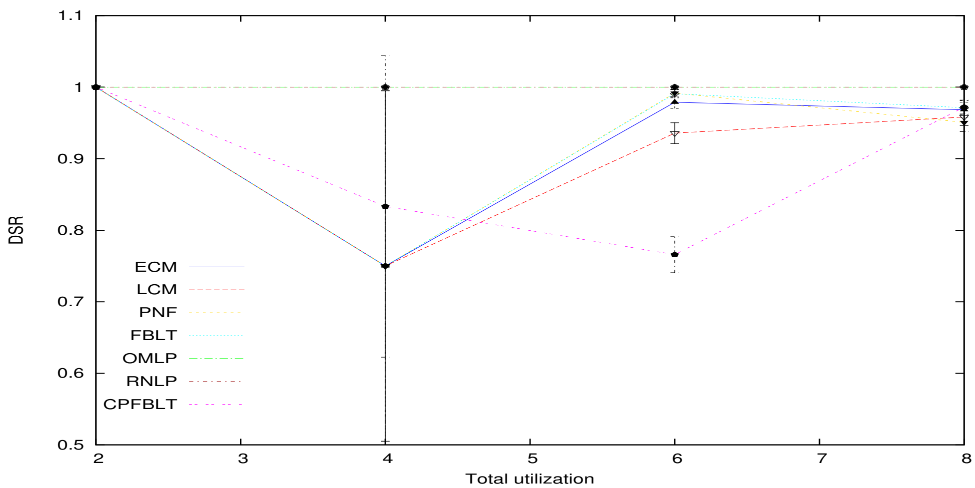


Figure B.15: DSR for Tasksets 15, 285, 555 and 825

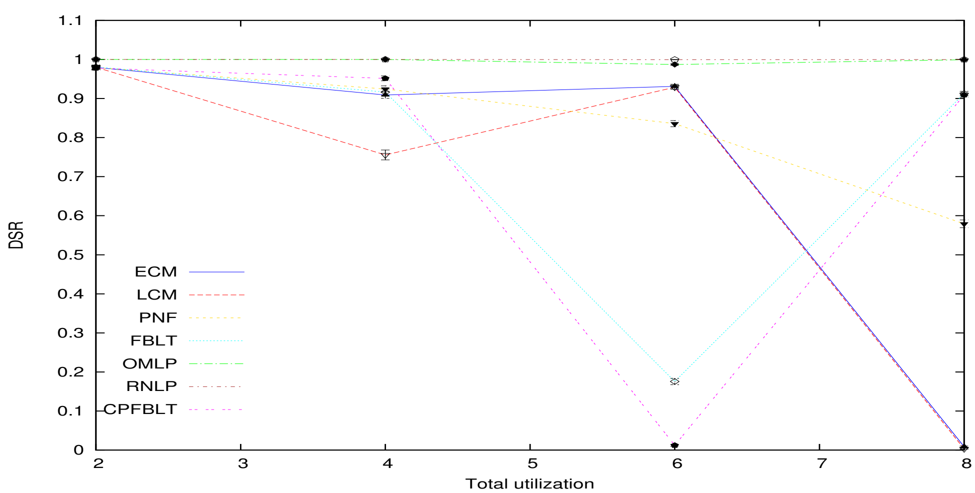


Figure B.16: DSR for Tasksets 16, 286, 556 and 826

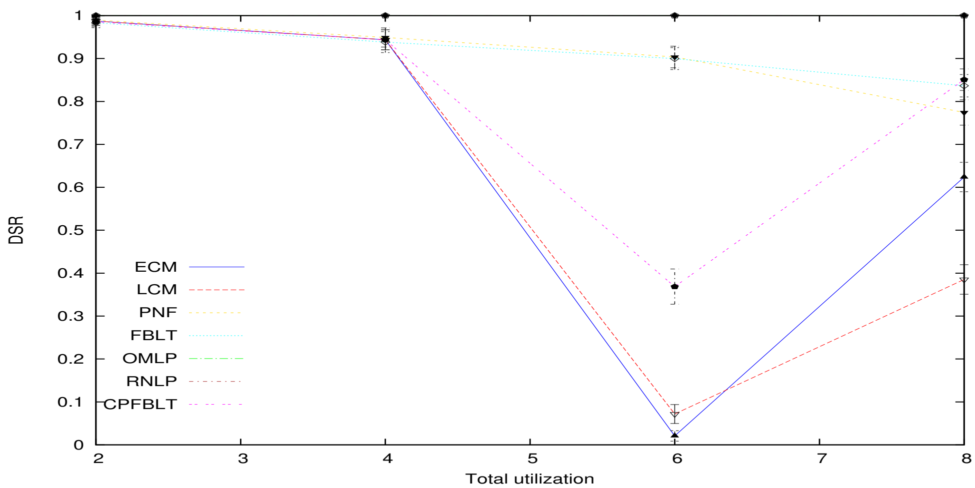


Figure B.17: DSR for Tasksets 17, 287, 557 and 827

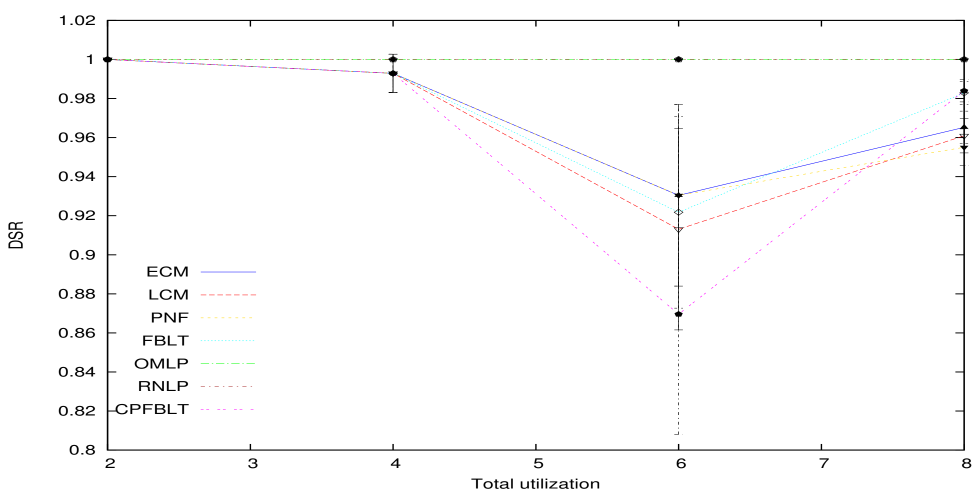


Figure B.18: DSR for Tasksets 18, 288, 558 and 828

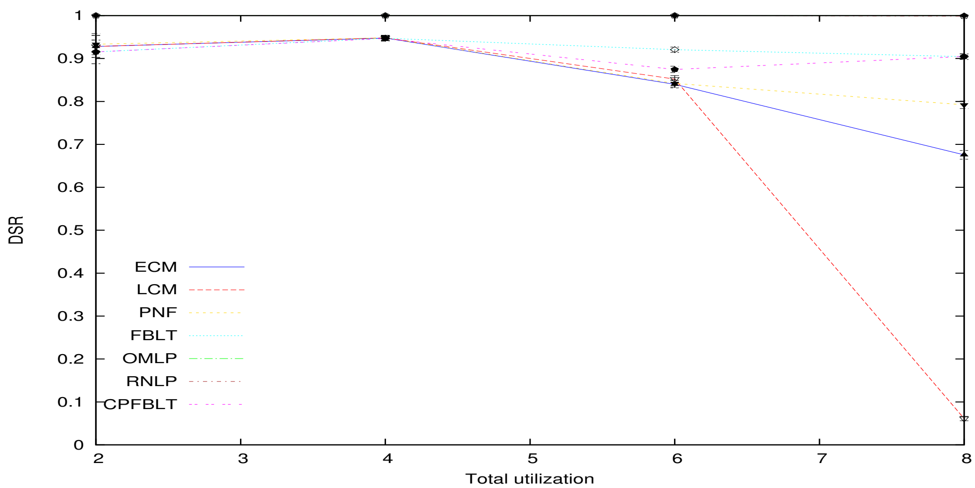


Figure B.19: DSR for Tasksets 19, 289, 559 and 829

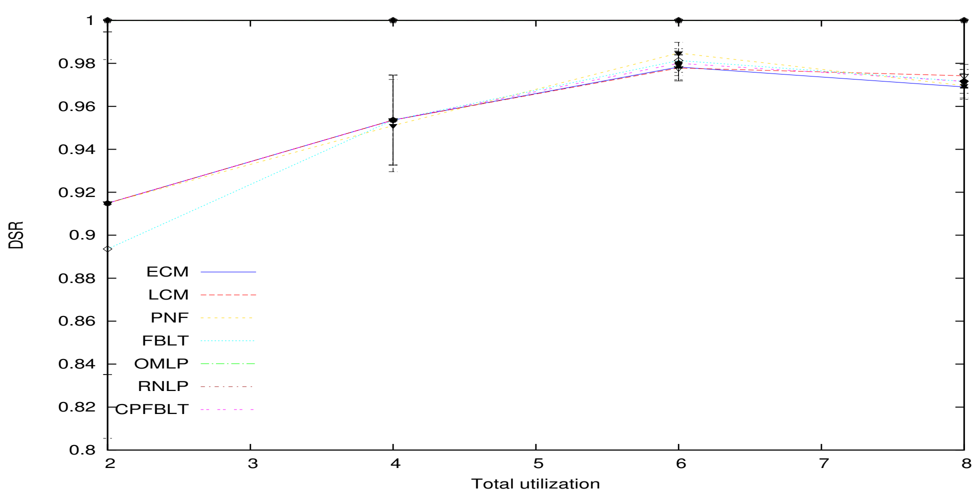


Figure B.20: DSR for Tasksets 20, 290, 560 and 830

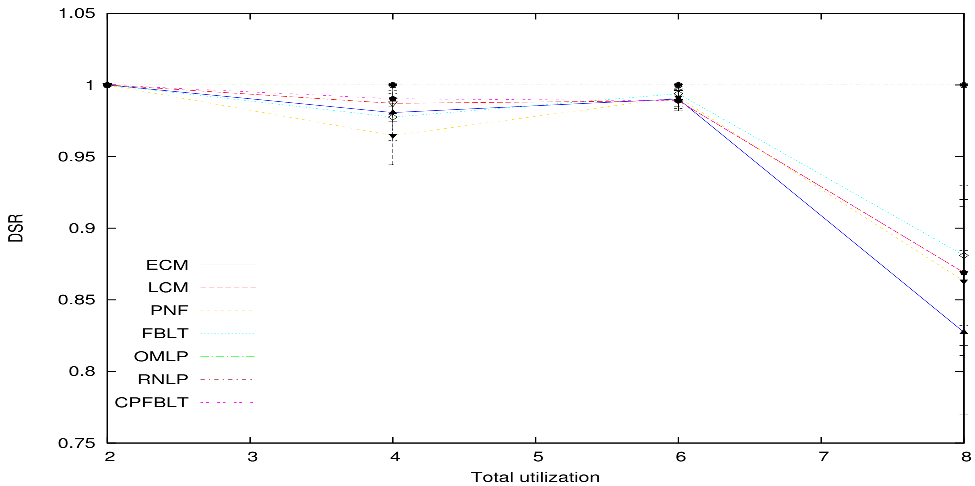


Figure B.21: DSR for Tasksets 21, 291, 561 and 831

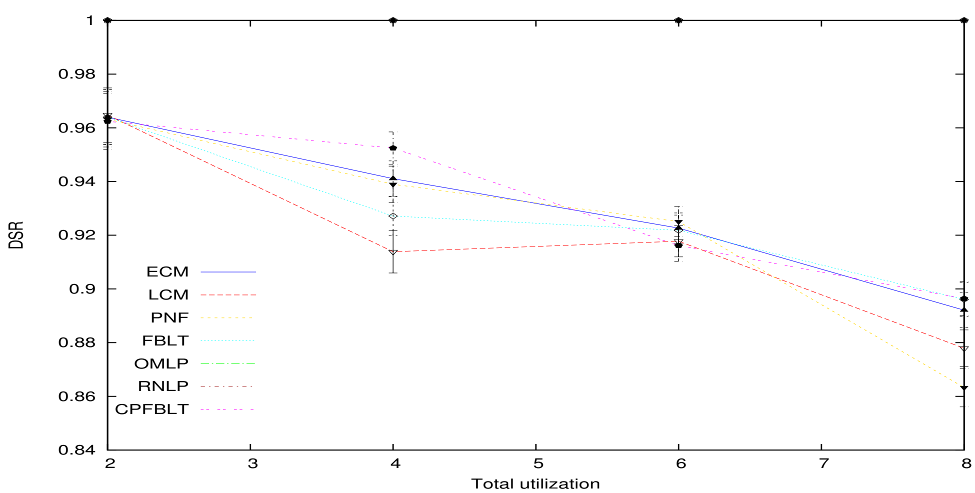


Figure B.22: DSR for Tasksets 22, 292, 562 and 832

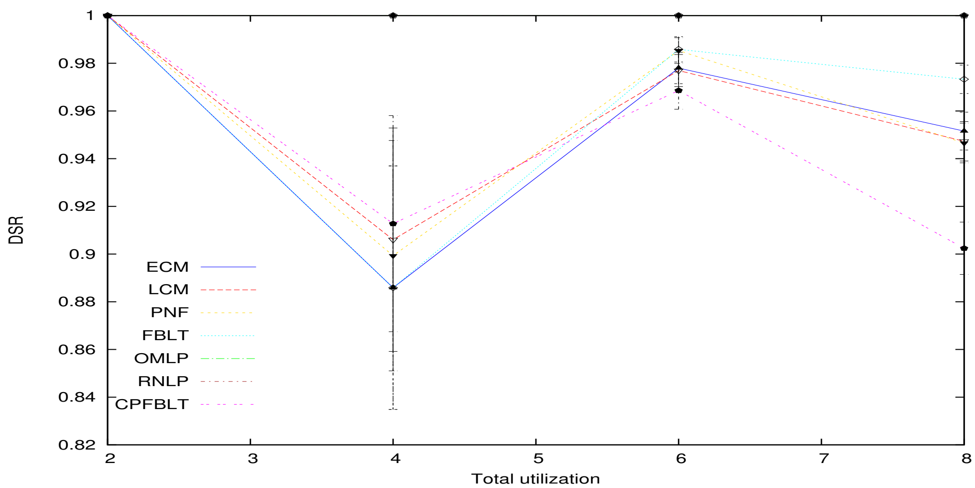


Figure B.23: DSR for Tasksets 23, 293, 563 and 833

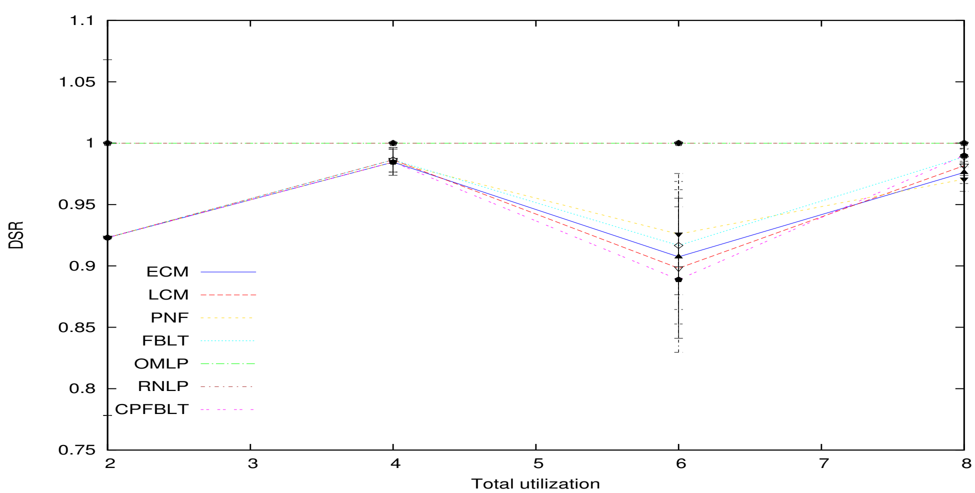


Figure B.24: DSR for Tasksets 24, 294, 564 and 834

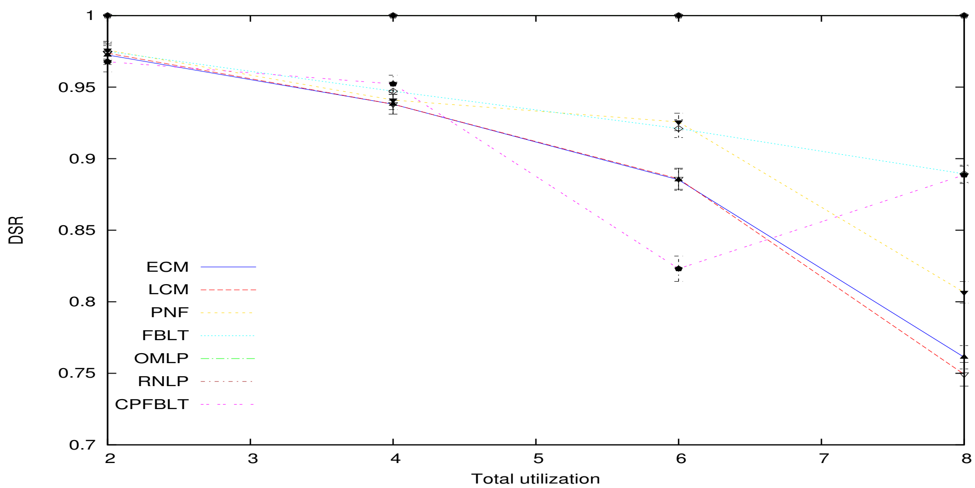


Figure B.25: DSR for Tasksets 25, 295, 565 and 835

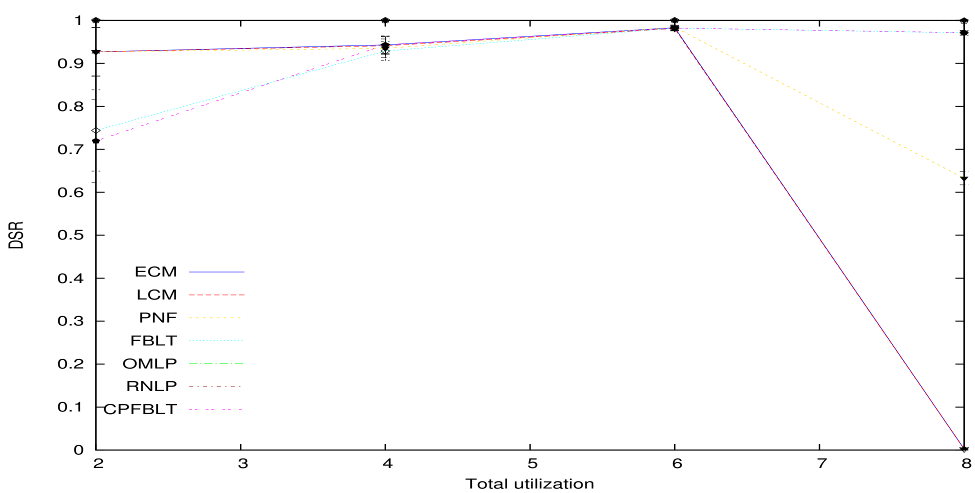


Figure B.26: DSR for Tasksets 26, 296, 566 and 836

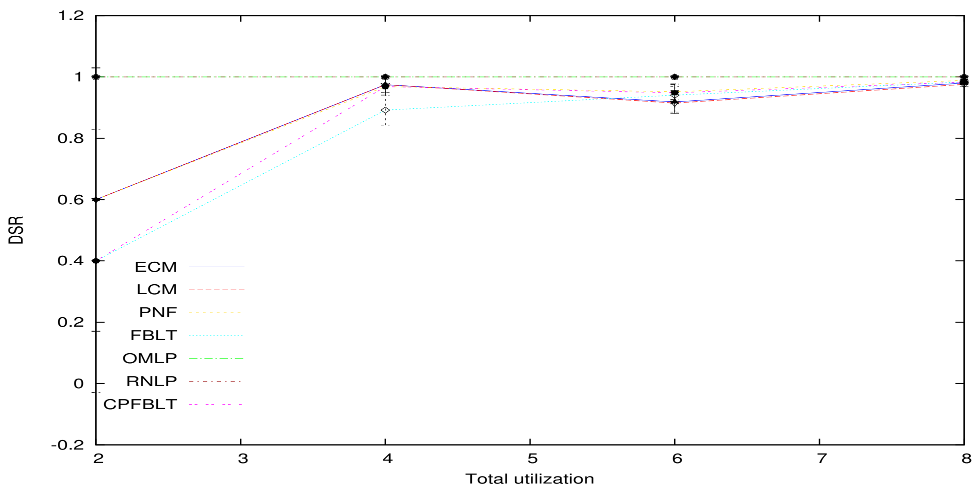


Figure B.27: DSR for Tasksets 27, 297, 567 and 837

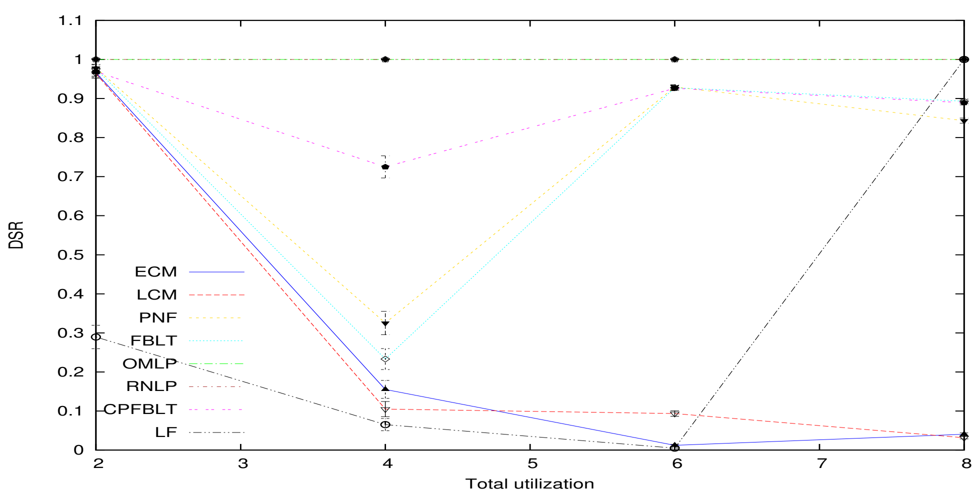


Figure B.28: DSR for Tasksets 28, 298, 568 and 838

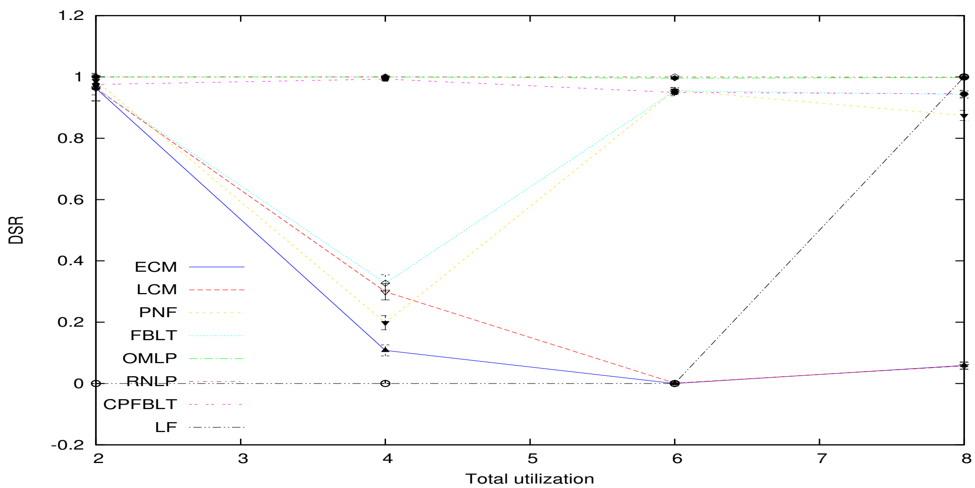


Figure B.29: DSR for Tasksets 29, 299, 569 and 839

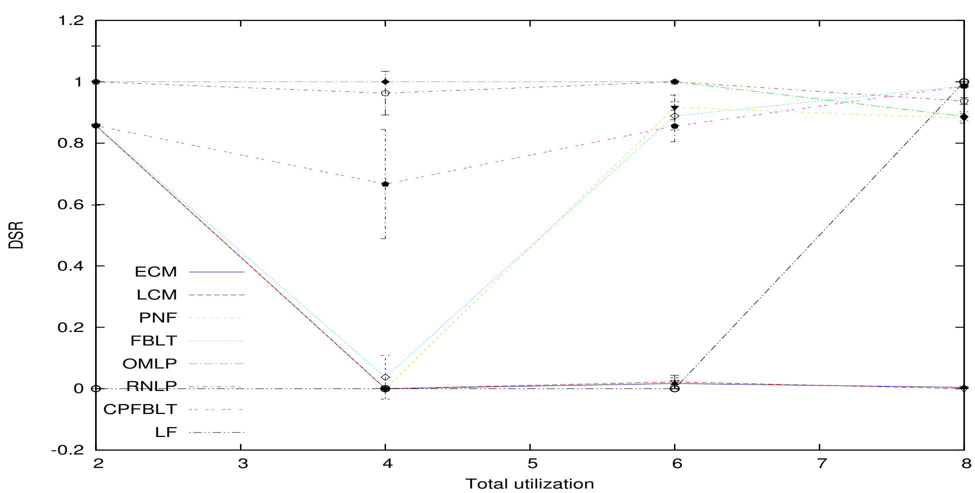


Figure B.30: DSR for Tasksets 30, 300, 570 and 840

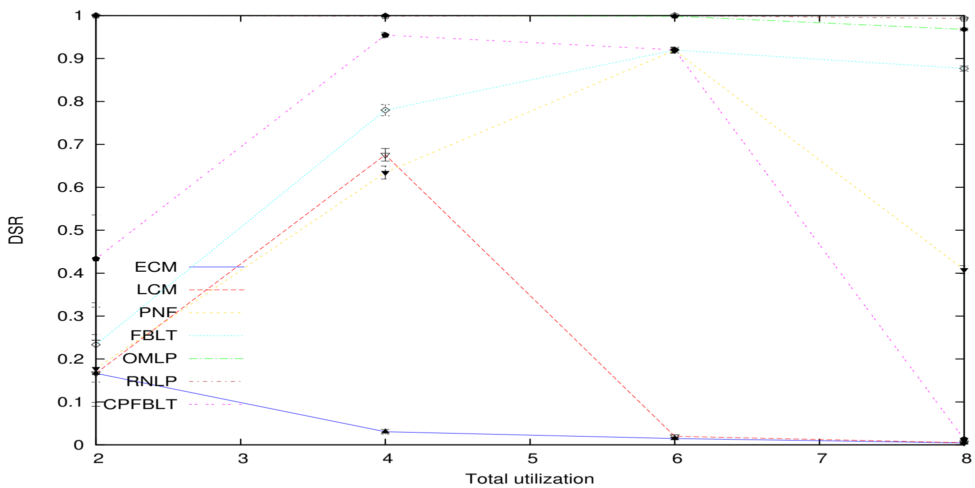


Figure B.31: DSR for Tasksets 31, 301, 571 and 841

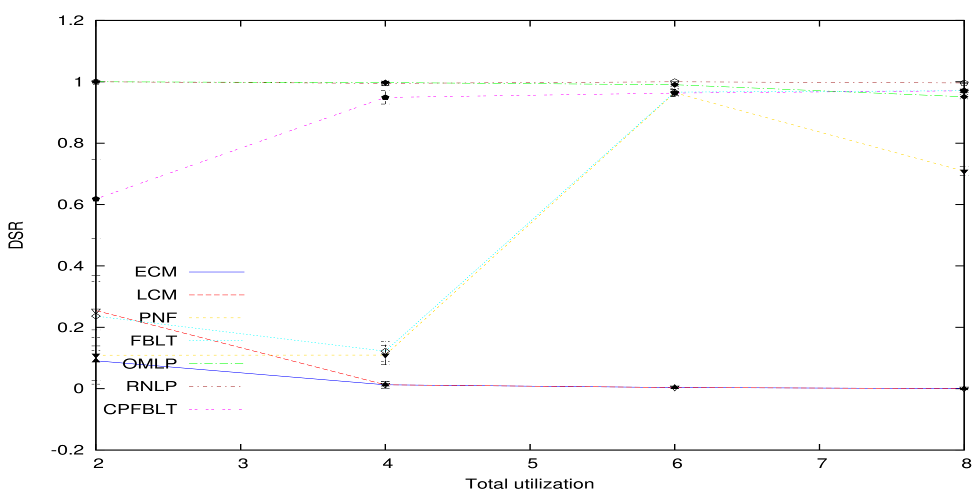


Figure B.32: DSR for Tasksets 32, 302, 572 and 842

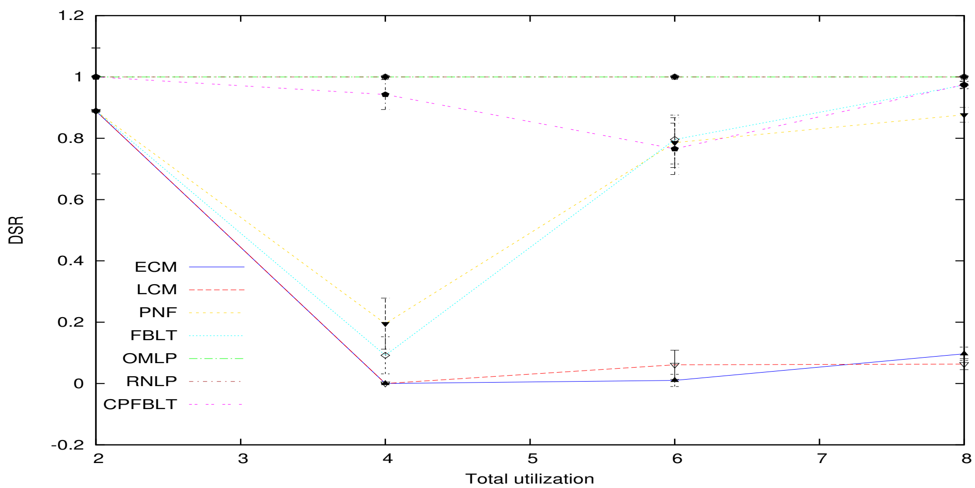


Figure B.33: DSR for Tasksets 33, 303, 573 and 843

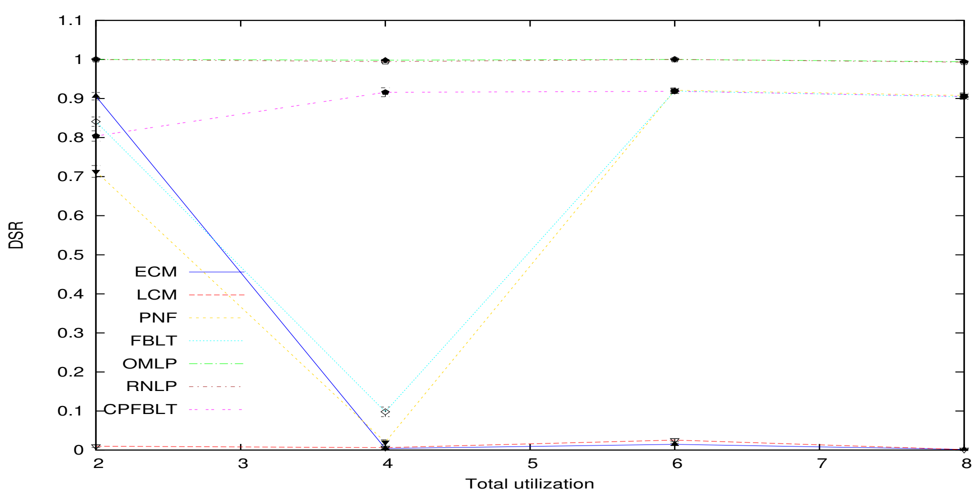


Figure B.34: DSR for Tasksets 34, 304, 574 and 844

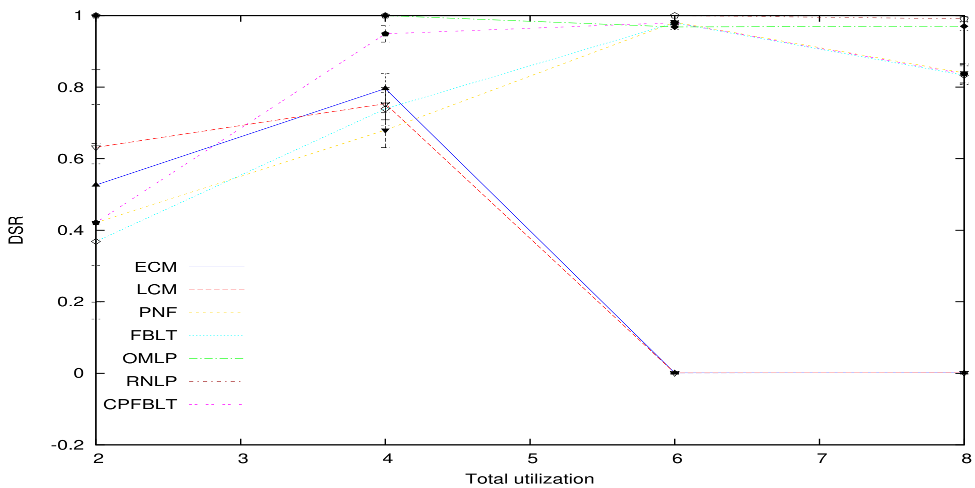


Figure B.35: DSR for Tasksets 35, 305, 575 and 845

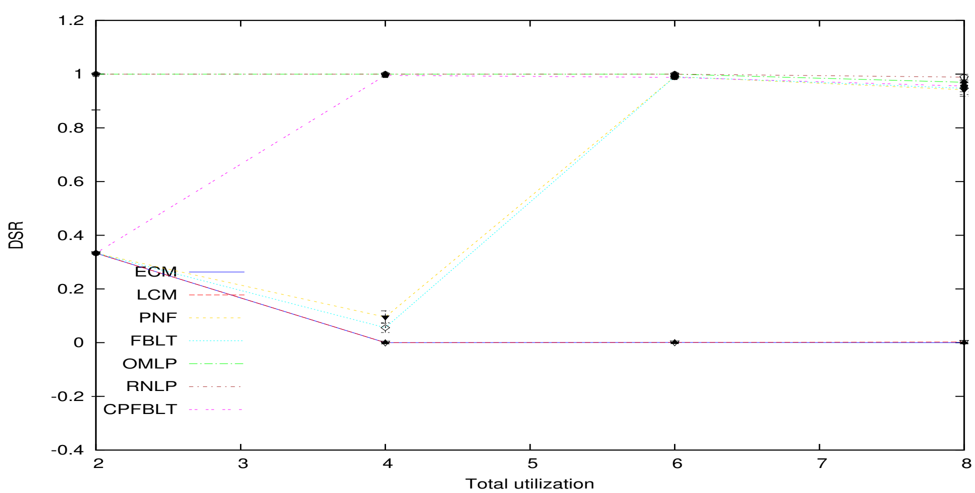


Figure B.36: DSR for Tasksets 36, 306, 576 and 846

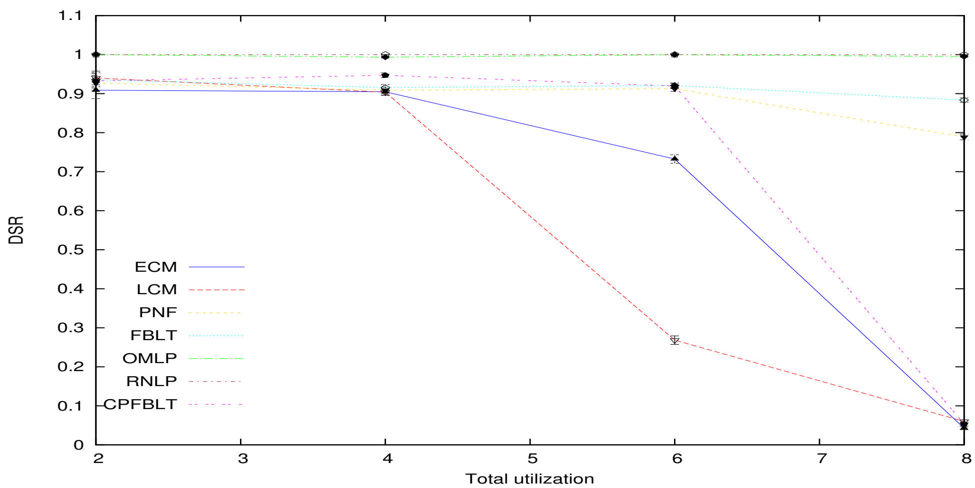


Figure B.37: DSR for Tasksets 37, 307, 577 and 847

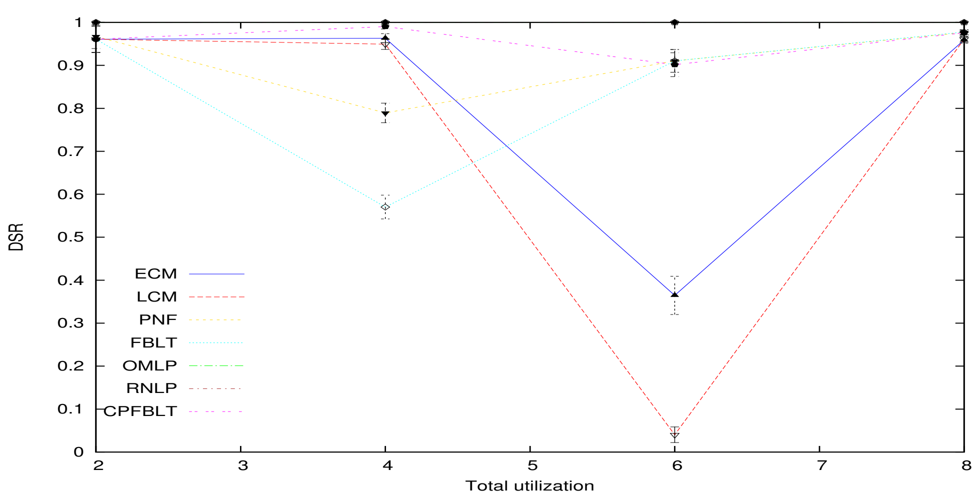


Figure B.38: DSR for Tasksets 38, 308, 578 and 848

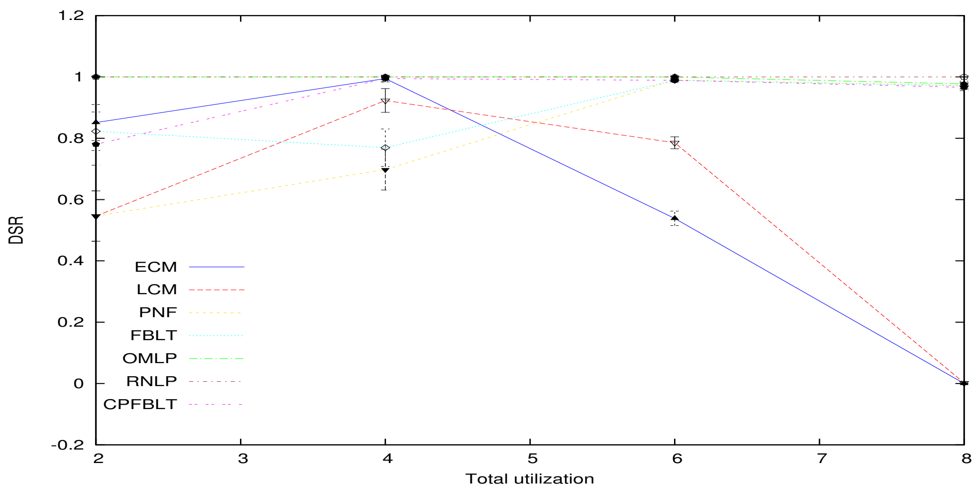


Figure B.39: DSR for Tasksets 39, 309, 579 and 849

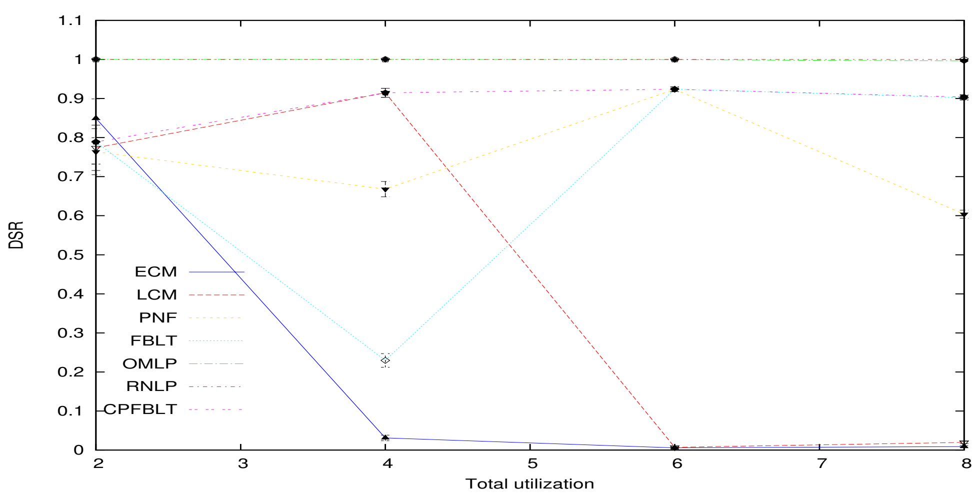


Figure B.40: DSR for Tasksets 40, 310, 580 and 850

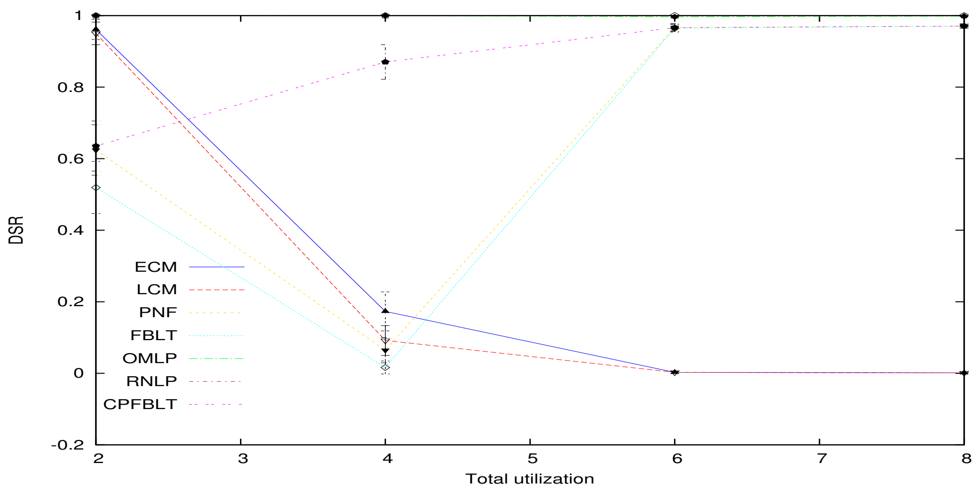


Figure B.41: DSR for Tasksets 41, 311, 581 and 851

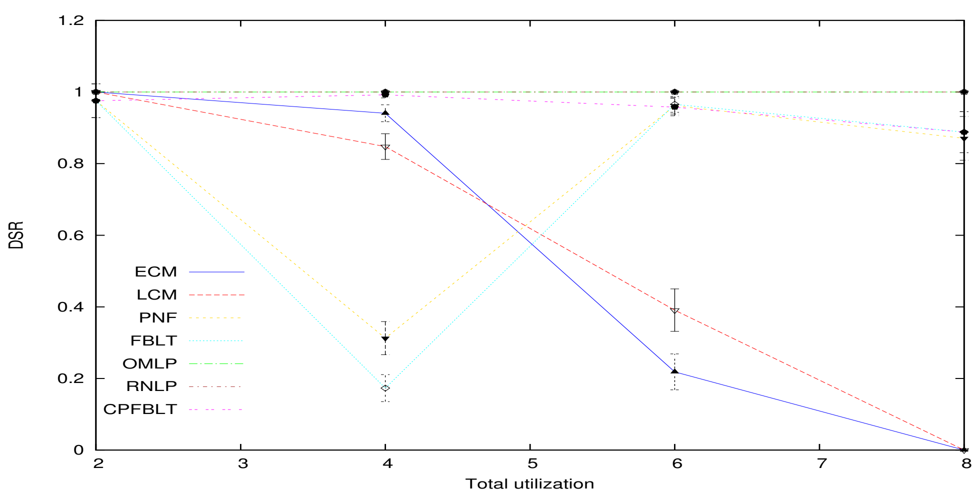


Figure B.42: DSR for Tasksets 42, 312, 582 and 852

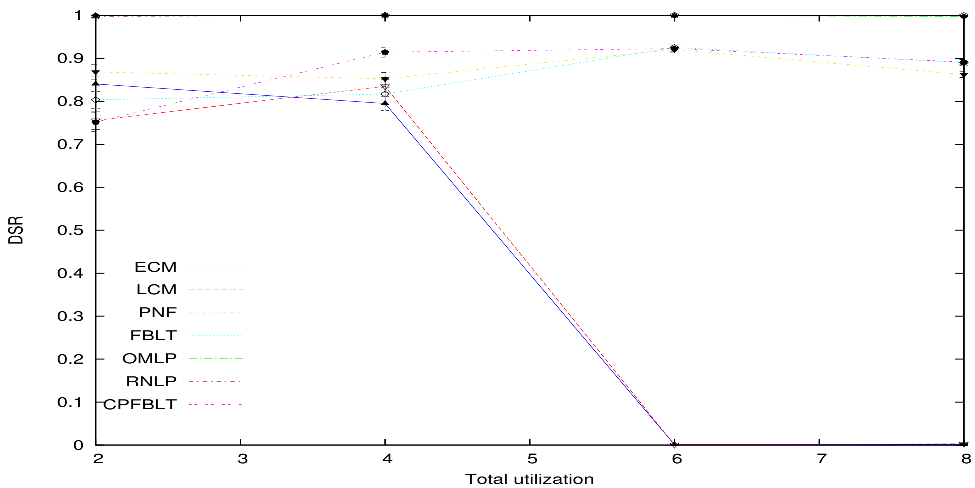


Figure B.43: DSR for Tasksets 43, 313, 583 and 853

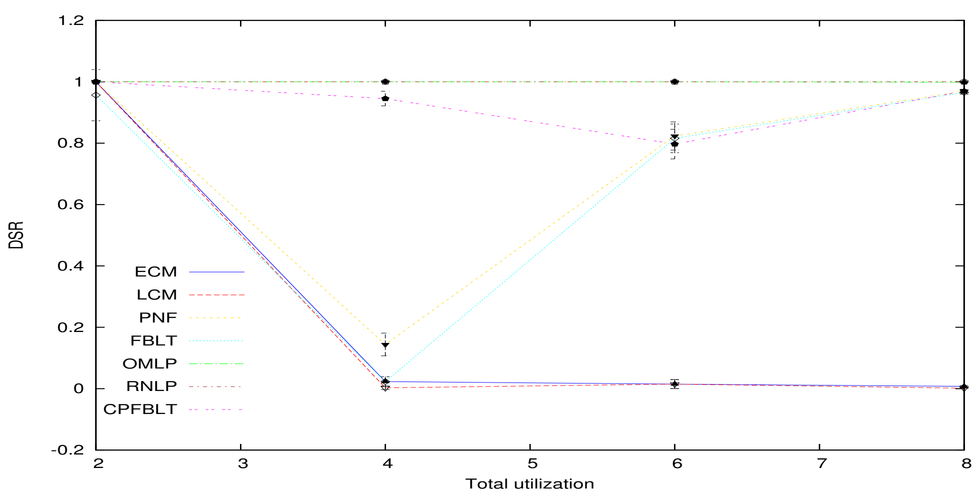


Figure B.44: DSR for Tasksets 44, 314, 584 and 854

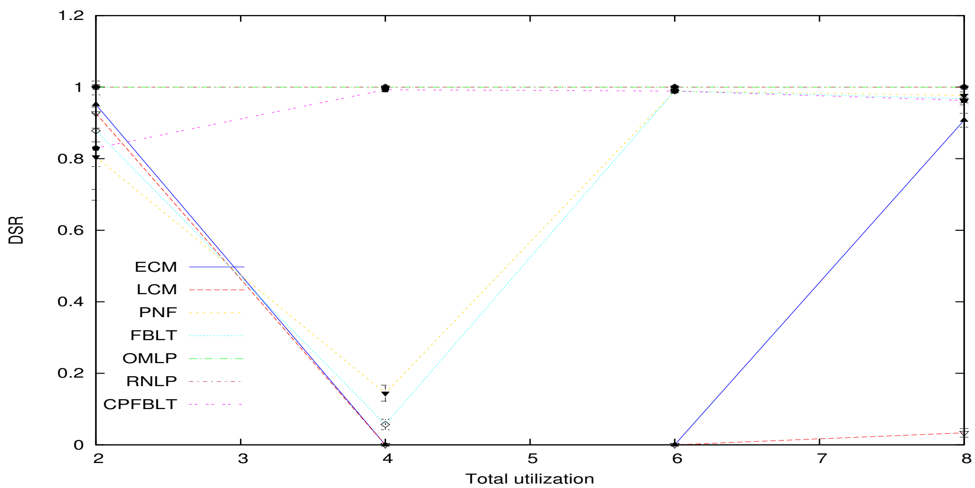


Figure B.45: DSR for Tasksets 45, 315, 585 and 855

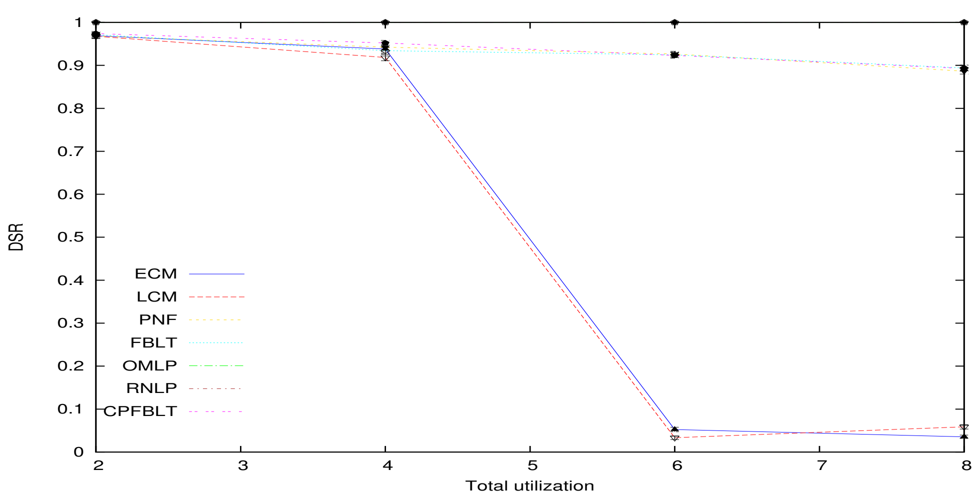


Figure B.46: DSR for Tasksets 46, 316, 586 and 856

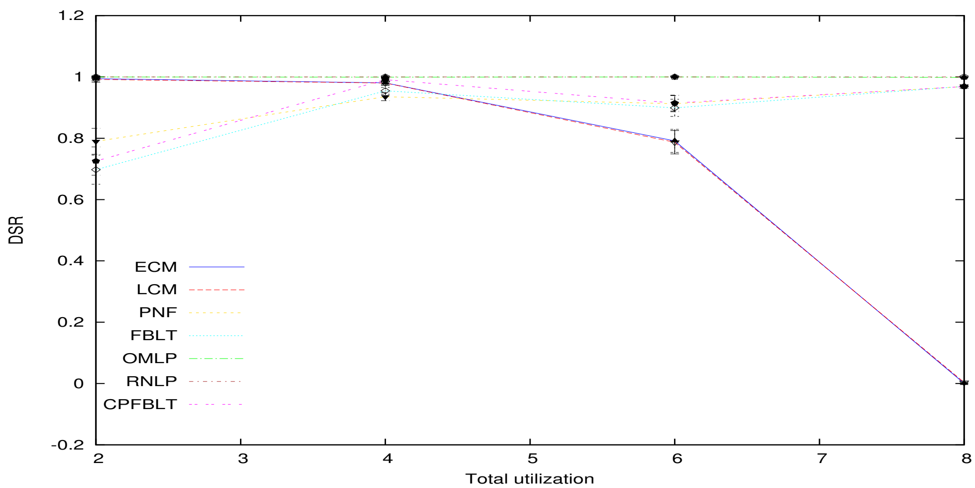


Figure B.47: DSR for Tasksets 47, 317, 587 and 857

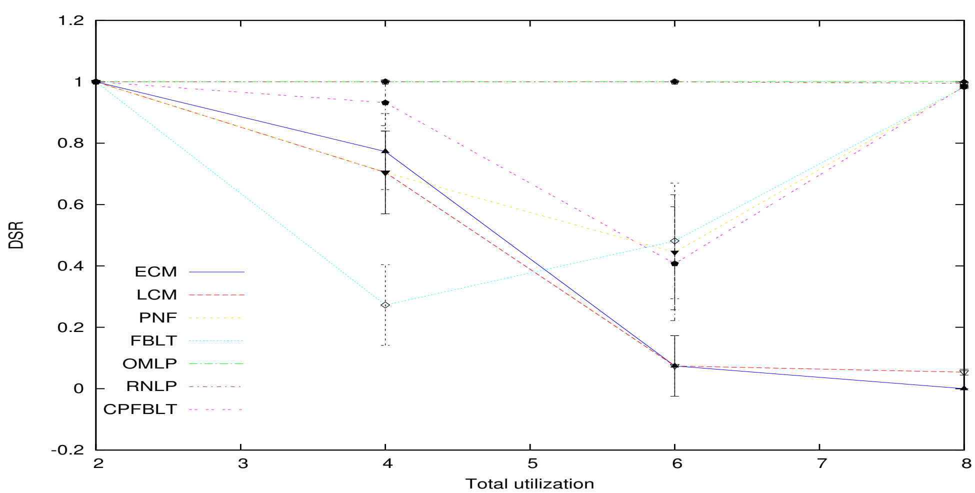


Figure B.48: DSR for Tasksets 48, 318, 588 and 858

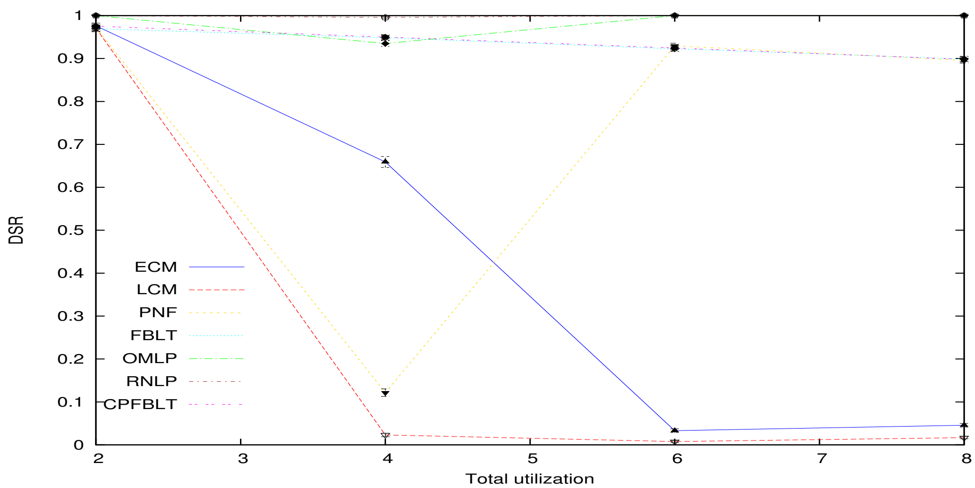


Figure B.49: DSR for Tasksets 49, 319, 589 and 859

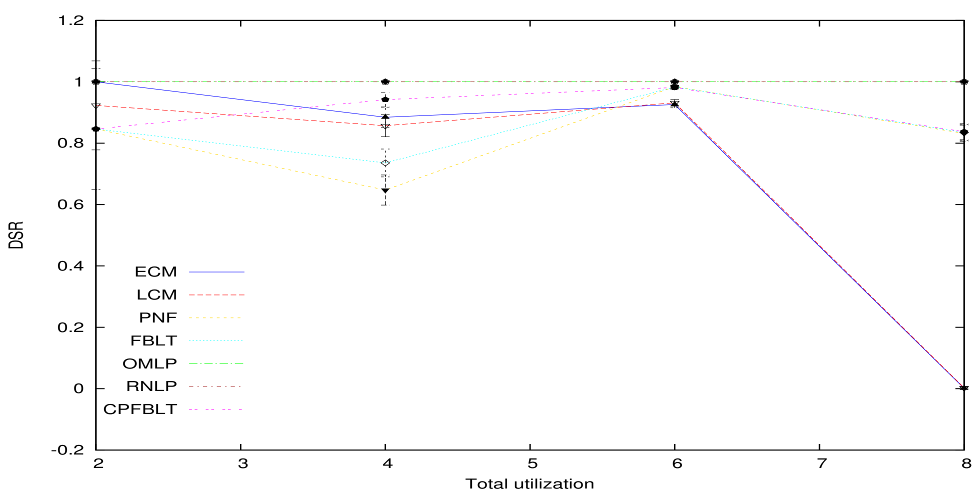


Figure B.50: DSR for Tasksets 50, 320, 590 and 860

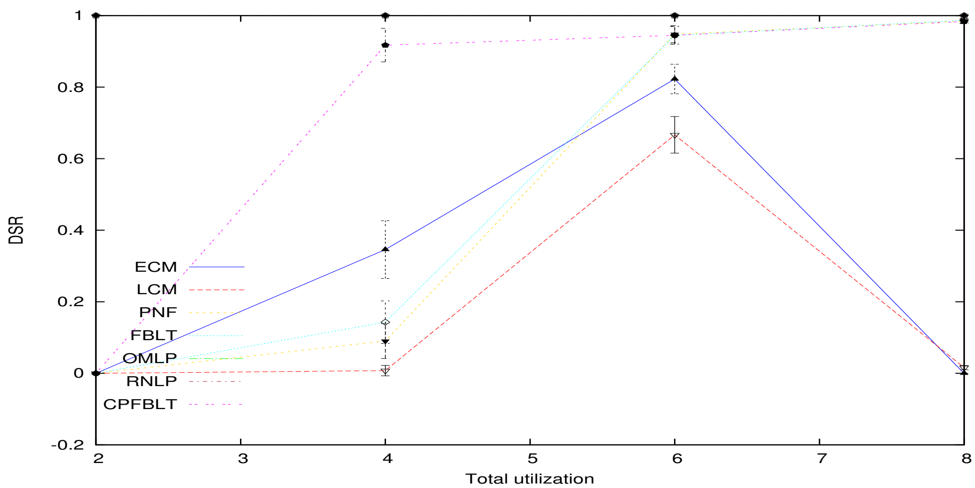


Figure B.51: DSR for Tasksets 51, 321, 591 and 861

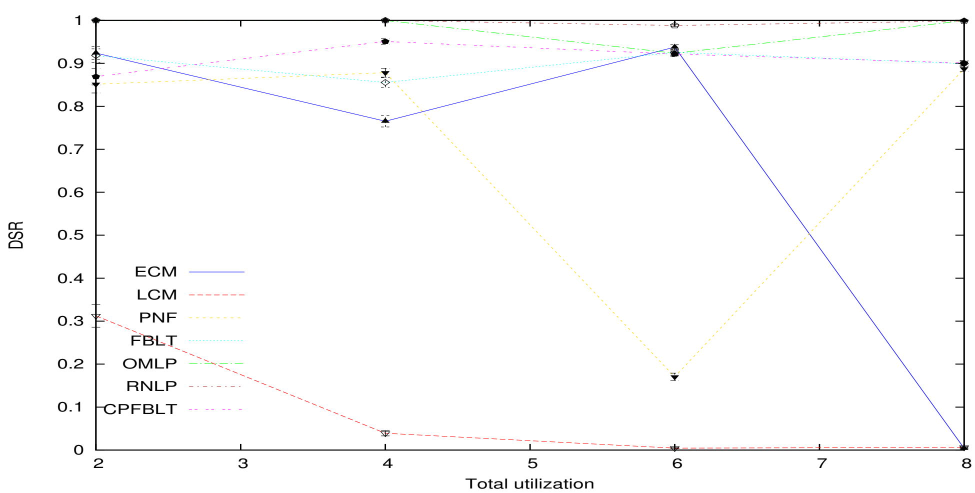


Figure B.52: DSR for Tasksets 52, 322, 592 and 862

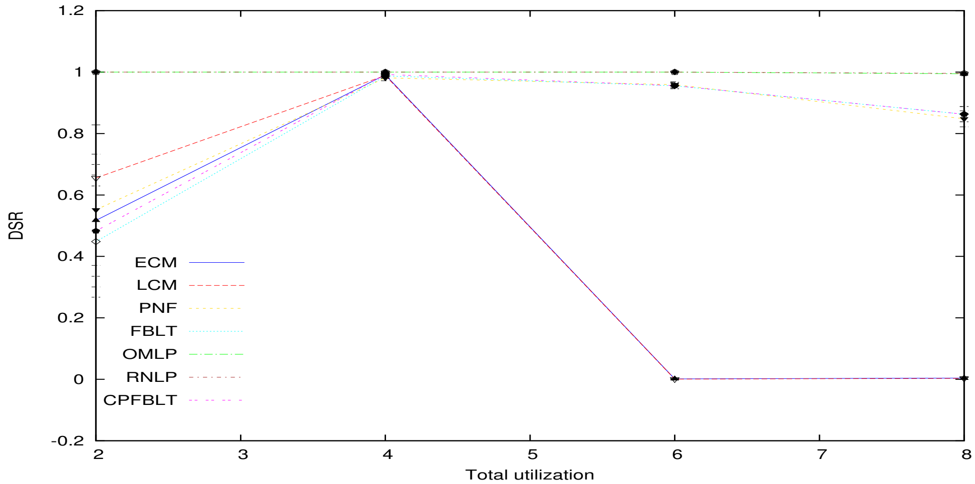


Figure B.53: DSR for Tasksets 53, 323, 593 and 863

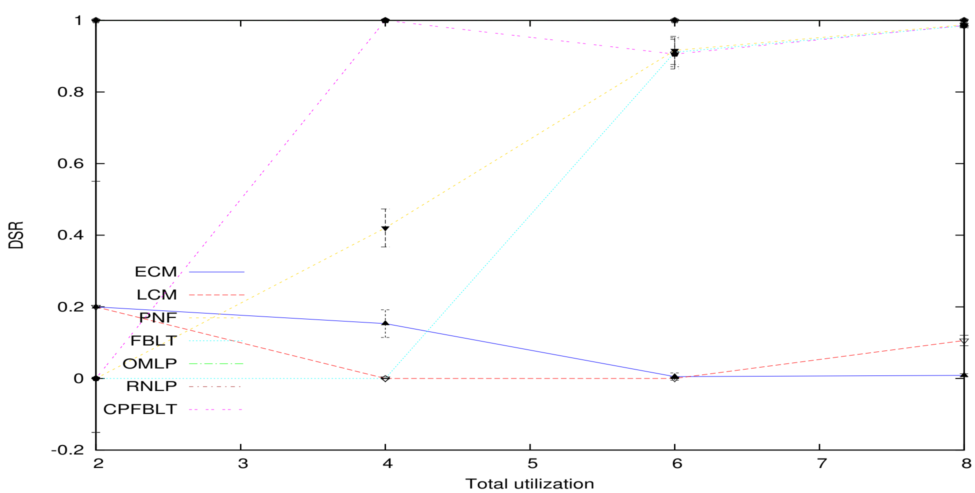


Figure B.54: DSR for Tasksets 54, 324, 594 and 864

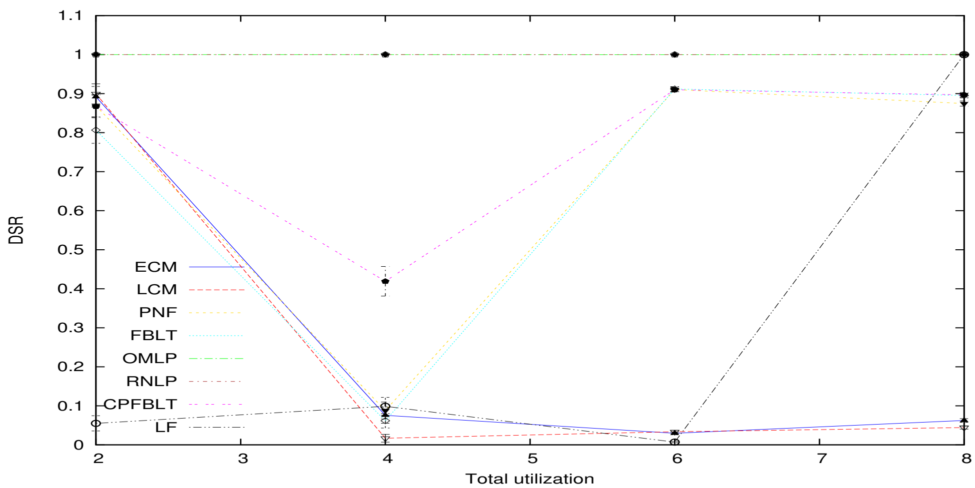


Figure B.55: DSR for Tasksets 55, 325, 595 and 865

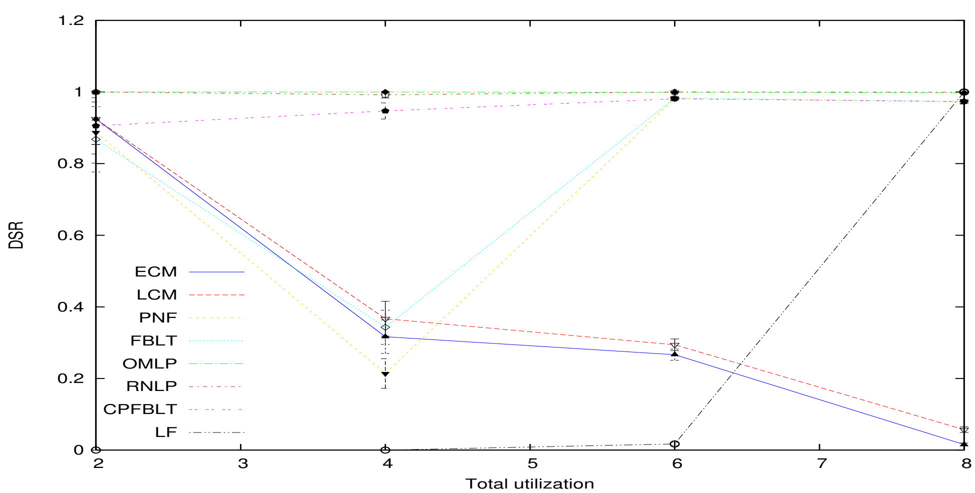


Figure B.56: DSR for Tasksets 56, 326, 596 and 866

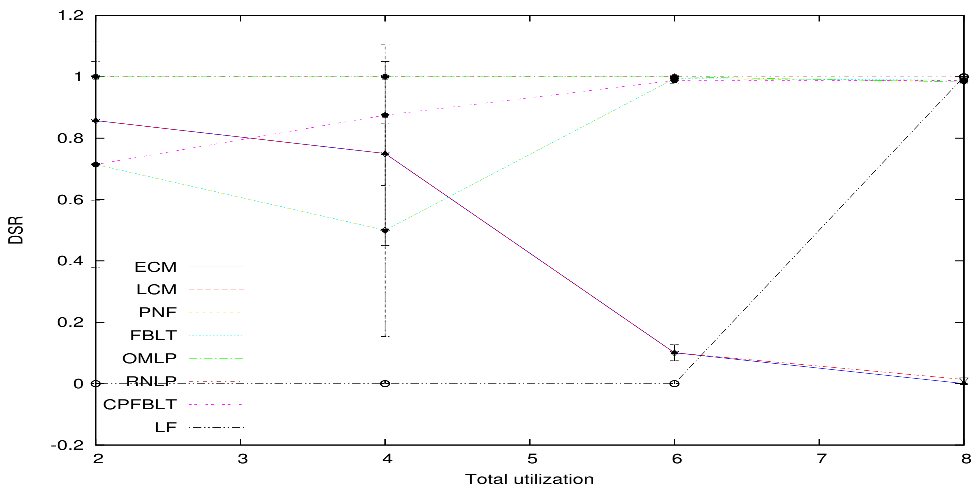


Figure B.57: DSR for Tasksets 57, 327, 597 and 867

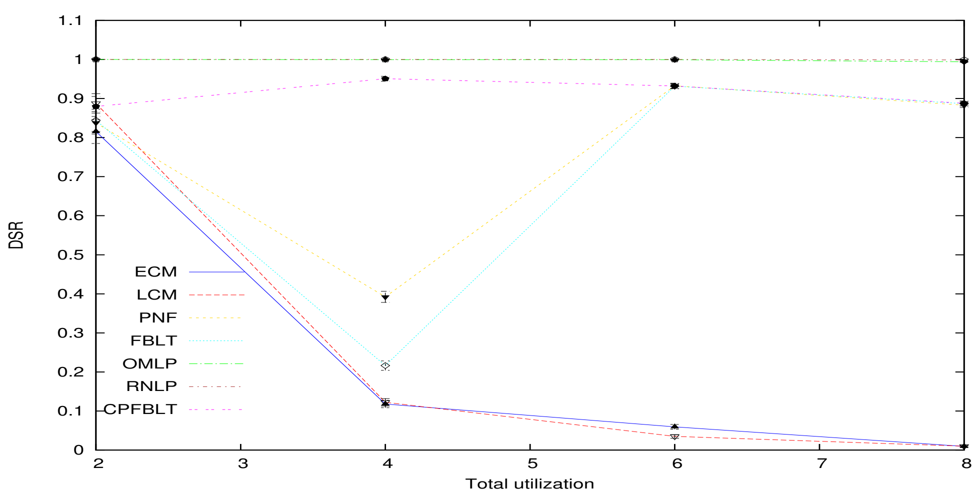


Figure B.58: DSR for Tasksets 58, 328, 598 and 868

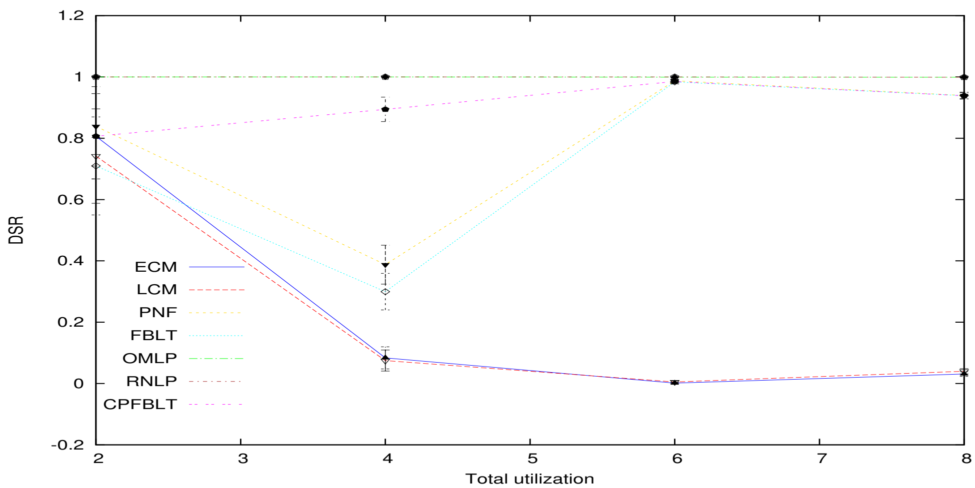


Figure B.59: DSR for Tasksets 59, 329, 599 and 869

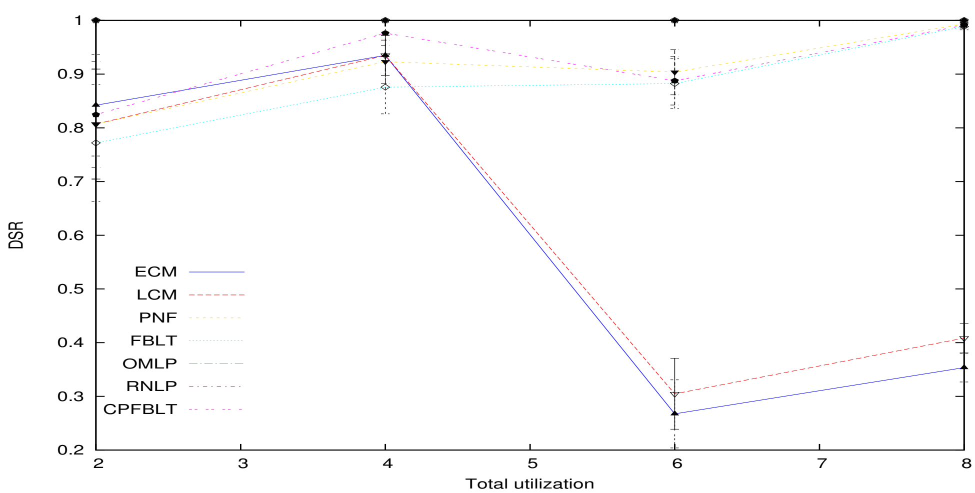


Figure B.60: DSR for Tasksets 60, 330, 600 and 870

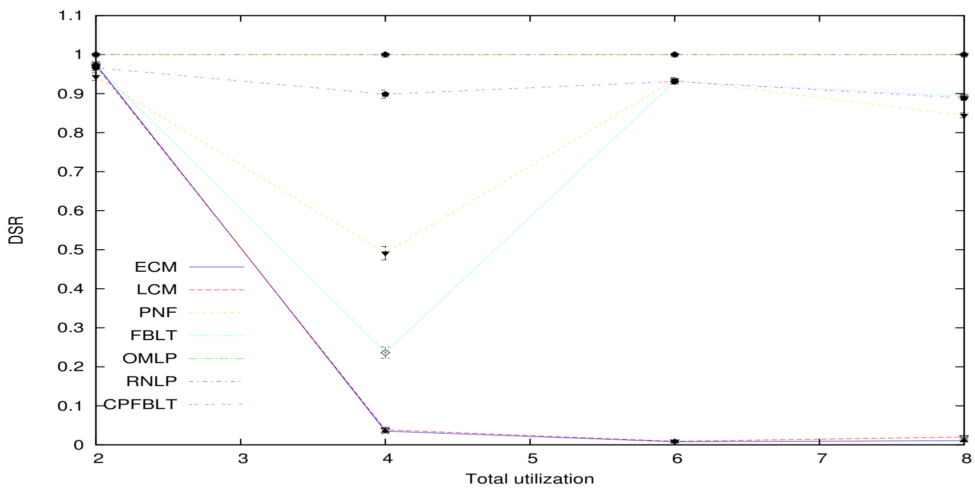


Figure B.61: DSR for Tasksets 61, 331, 601 and 871

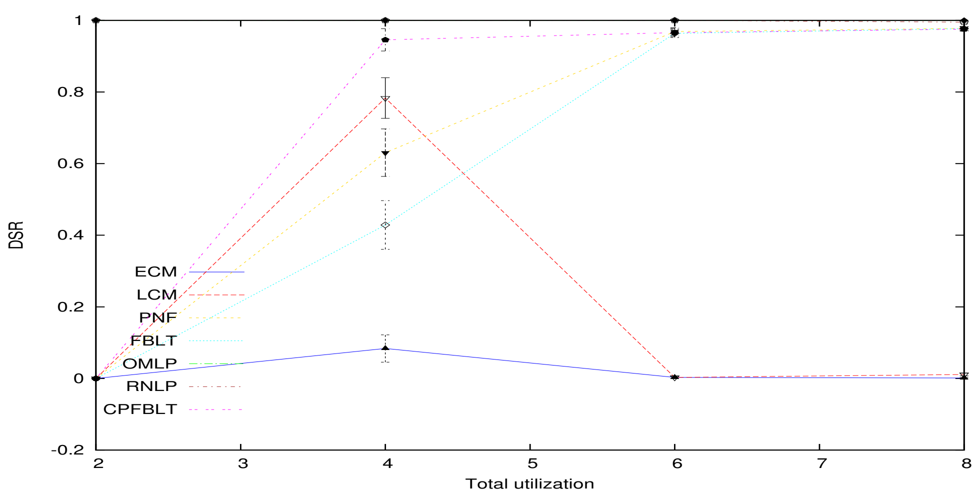


Figure B.62: DSR for Tasksets 62, 332, 602 and 872

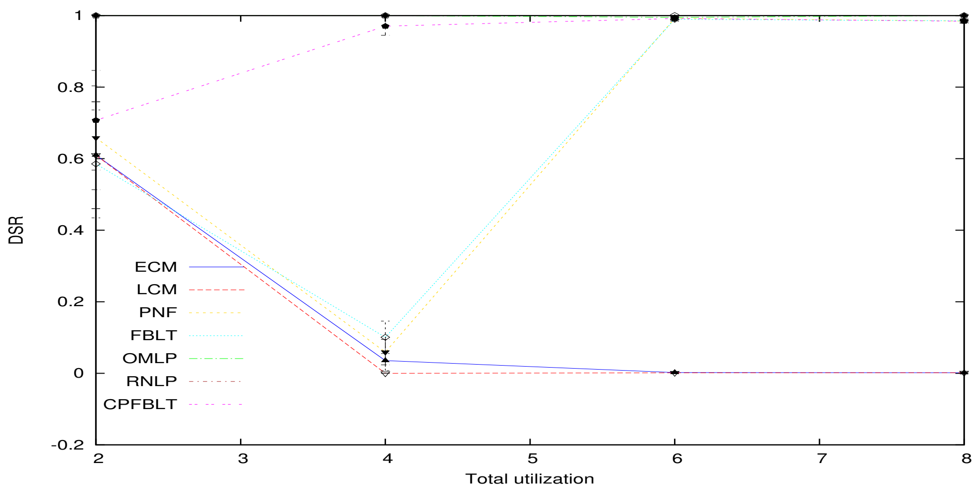


Figure B.63: DSR for Tasksets 63, 333, 603 and 873

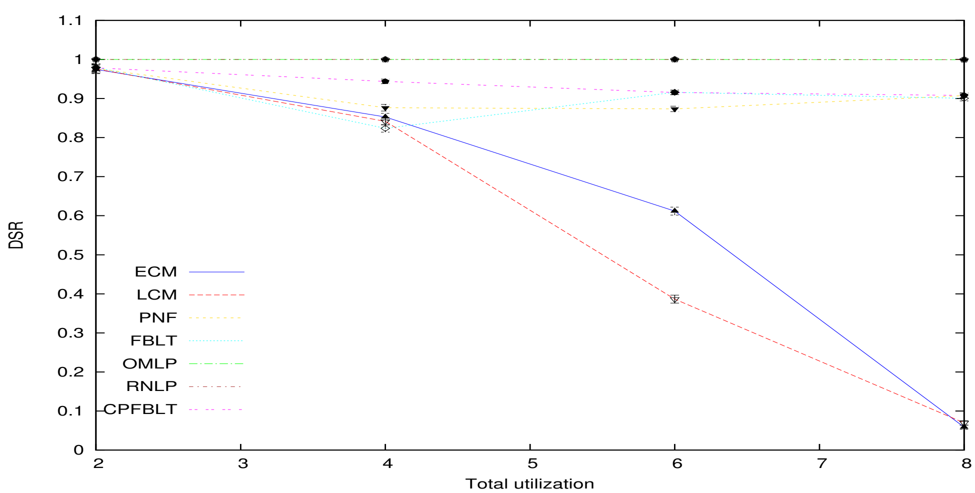


Figure B.64: DSR for Tasksets 64, 334, 604 and 874

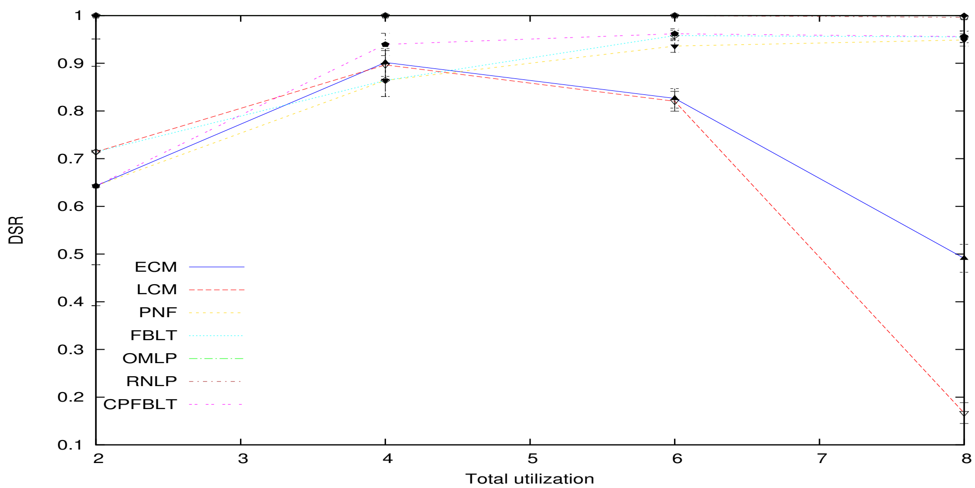


Figure B.65: DSR for Tasksets 65, 335, 605 and 875

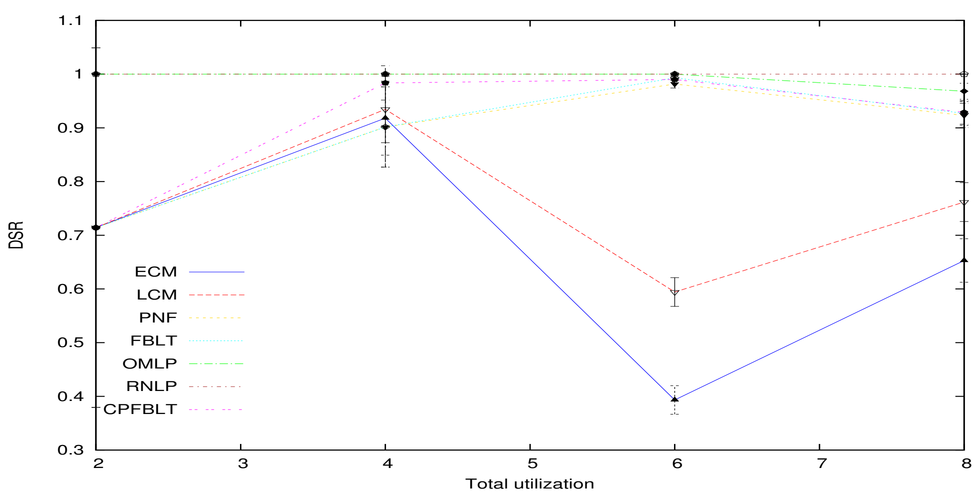


Figure B.66: DSR for Tasksets 66, 336, 606 and 876

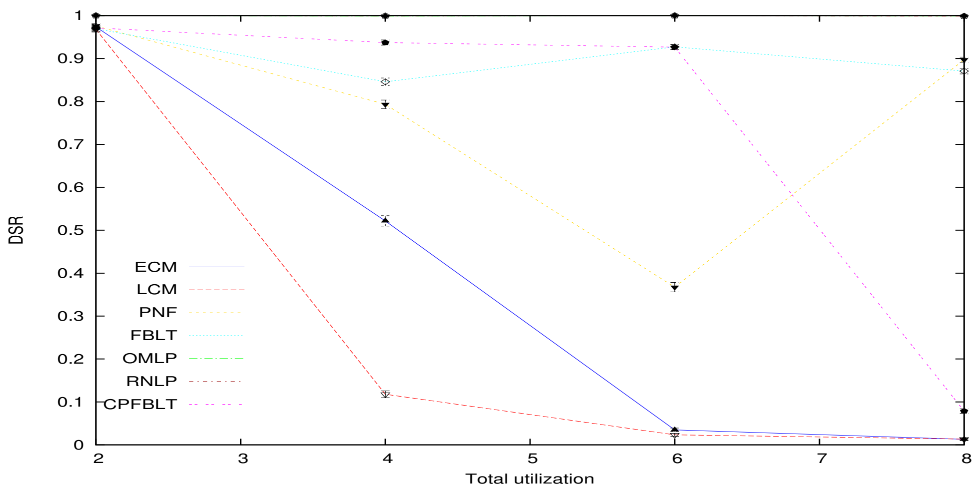


Figure B.67: DSR for Tasksets 67, 337, 607 and 877

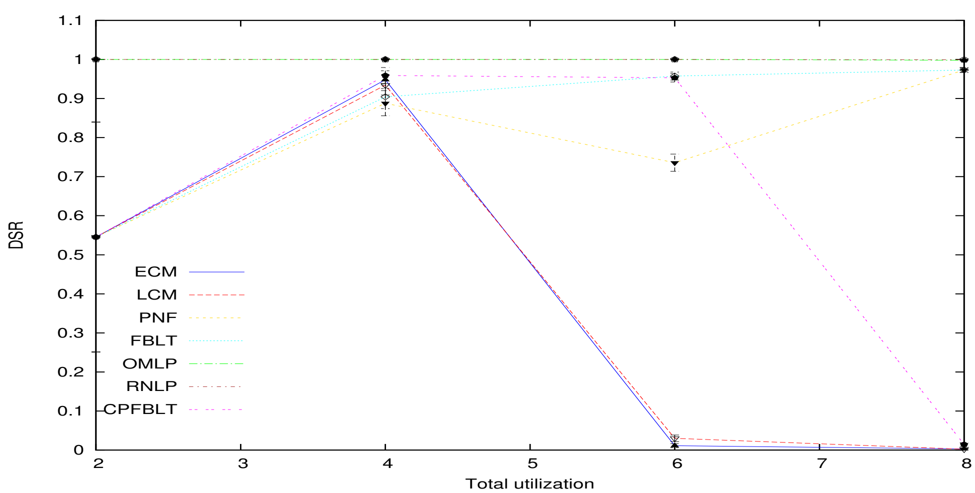


Figure B.68: DSR for Tasksets 68, 338, 608 and 878

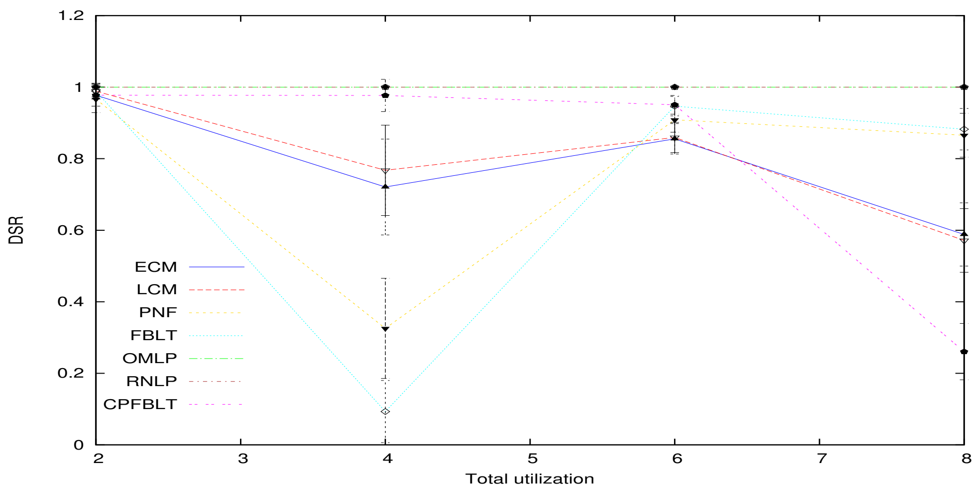


Figure B.69: DSR for Tasksets 69, 339, 609 and 879

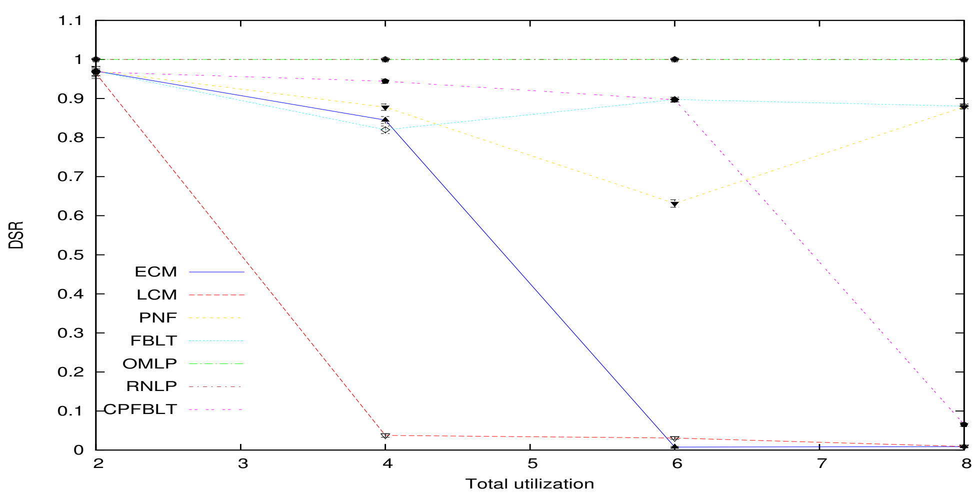


Figure B.70: DSR for Tasksets 70, 340, 610 and 880

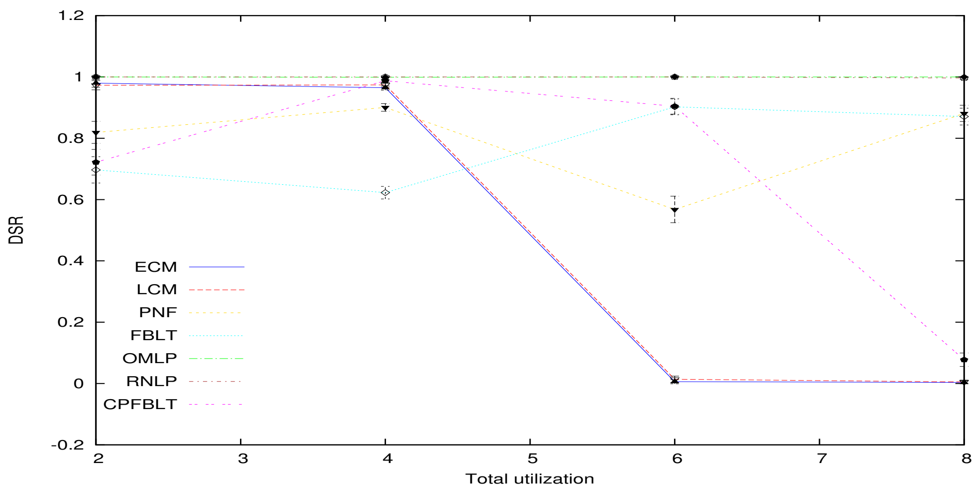


Figure B.71: DSR for Tasksets 71, 341, 611 and 881

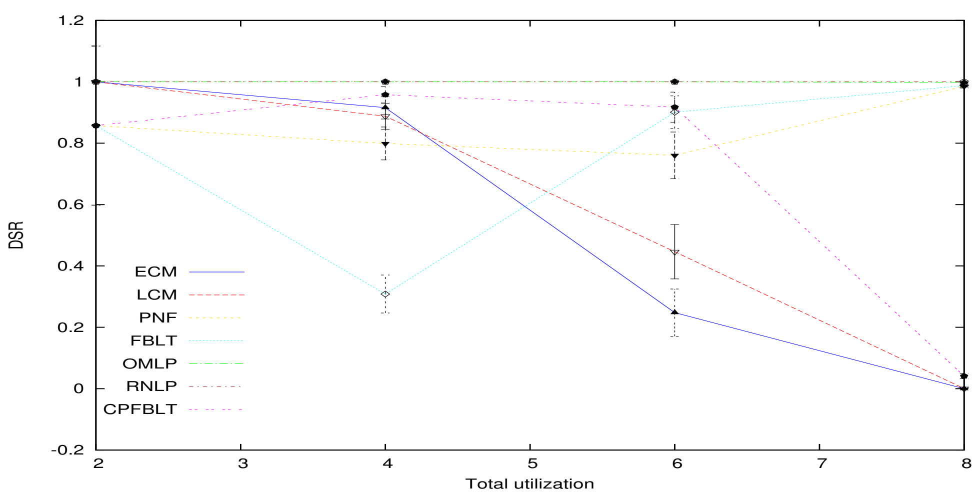


Figure B.72: DSR for Tasksets 72, 342, 612 and 882

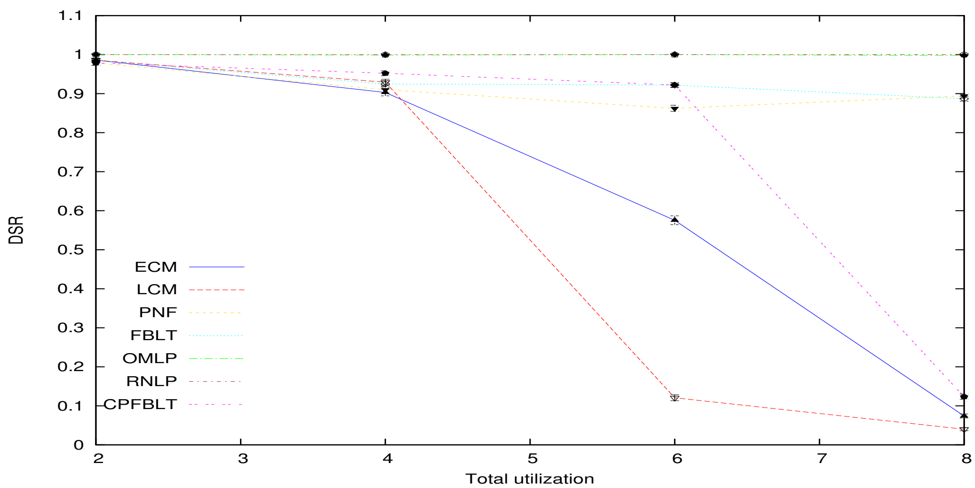


Figure B.73: DSR for Tasksets 73, 343, 613 and 883

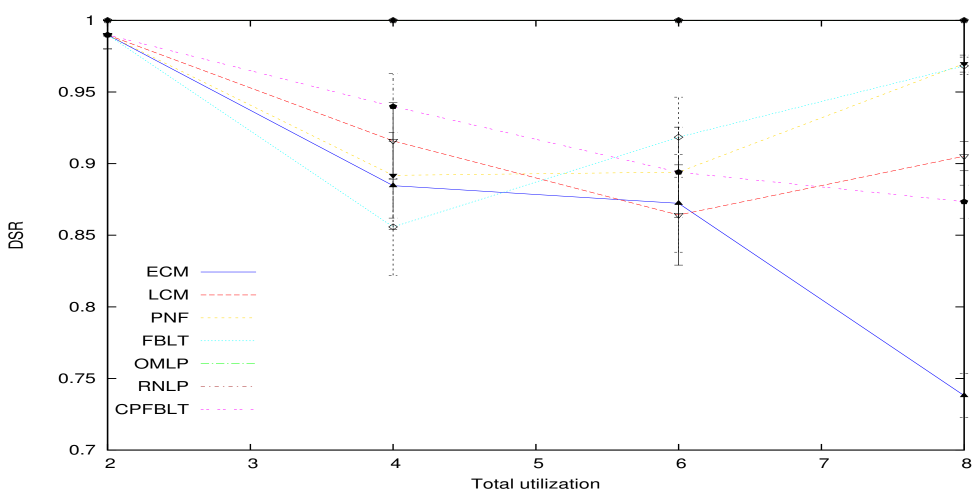


Figure B.74: DSR for Tasksets 74, 344, 614 and 884

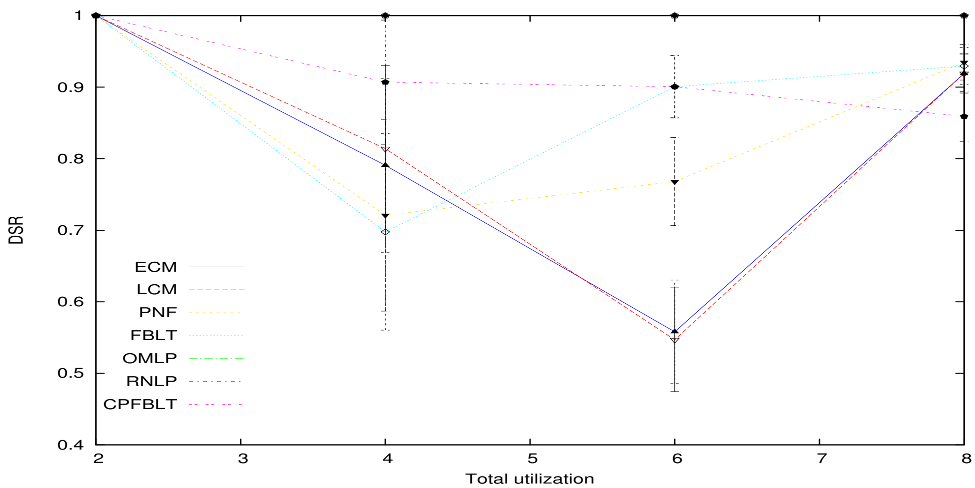


Figure B.75: DSR for Tasksets 75, 345, 615 and 885

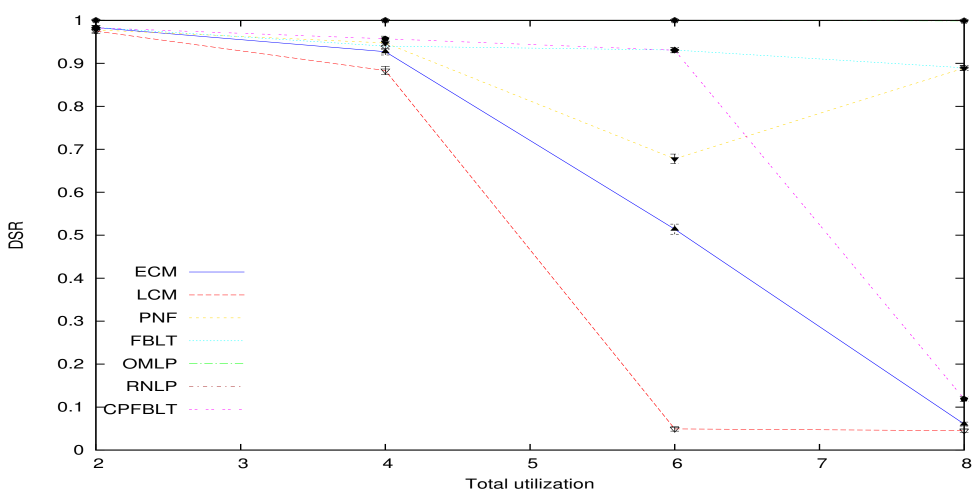


Figure B.76: DSR for Tasksets 76, 346, 616 and 886

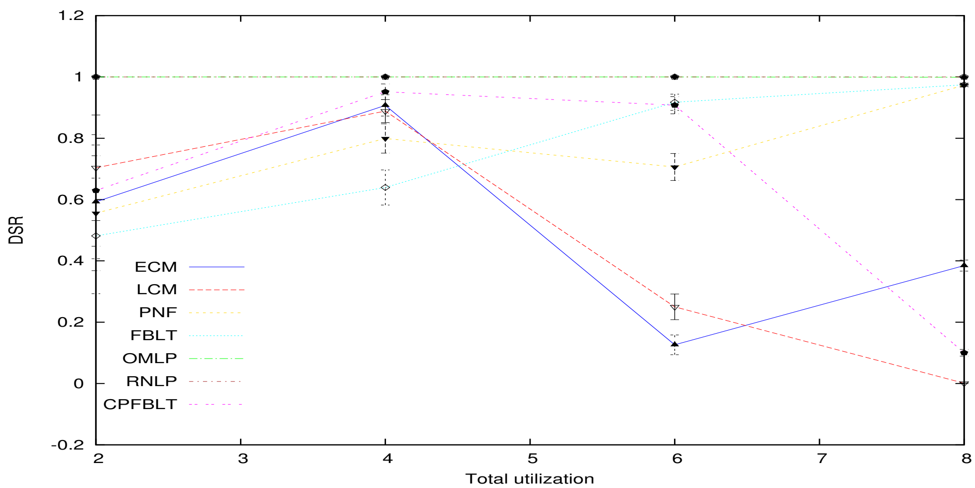


Figure B.77: DSR for Tasksets 77, 347, 617 and 887

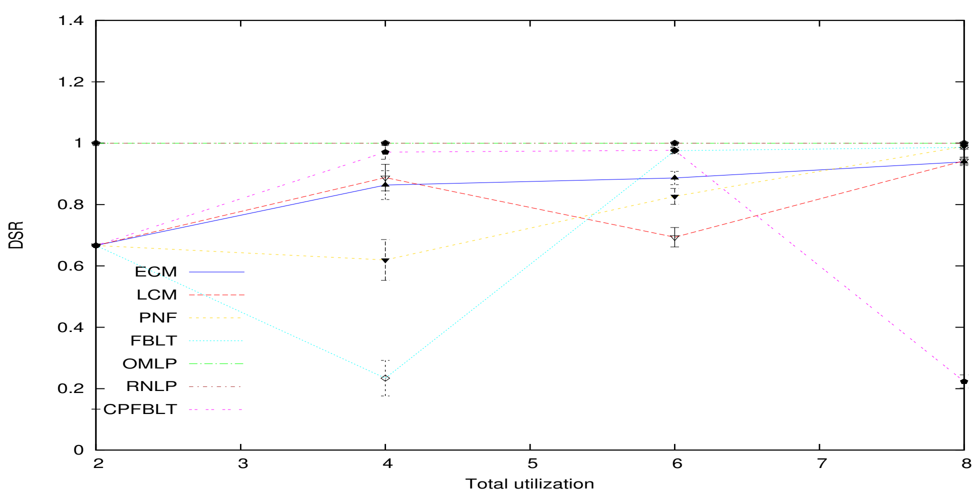


Figure B.78: DSR for Tasksets 78, 348, 618 and 888

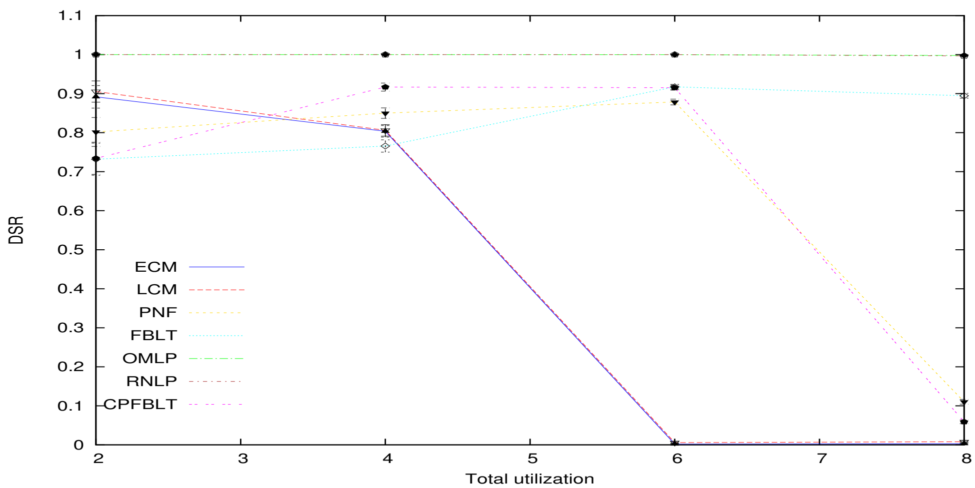


Figure B.79: DSR for Tasksets 79, 349, 619 and 889

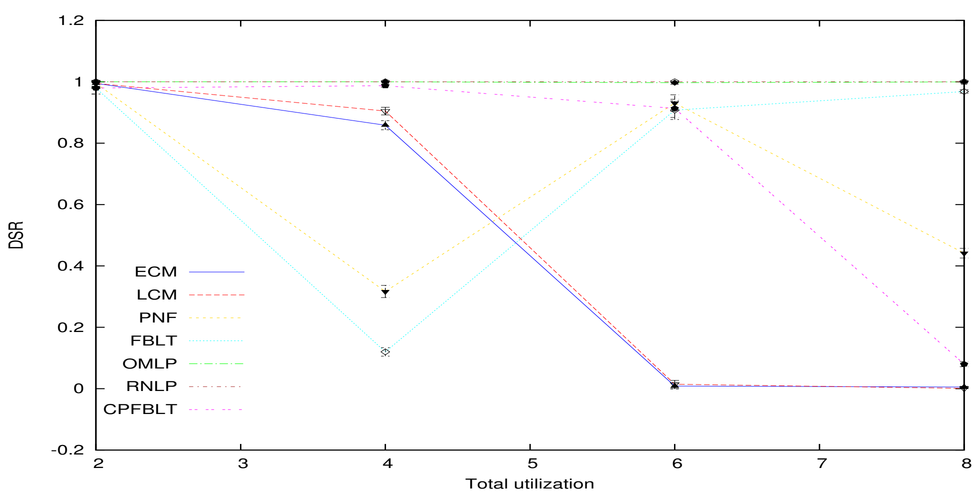


Figure B.80: DSR for Tasksets 80, 350, 620 and 890

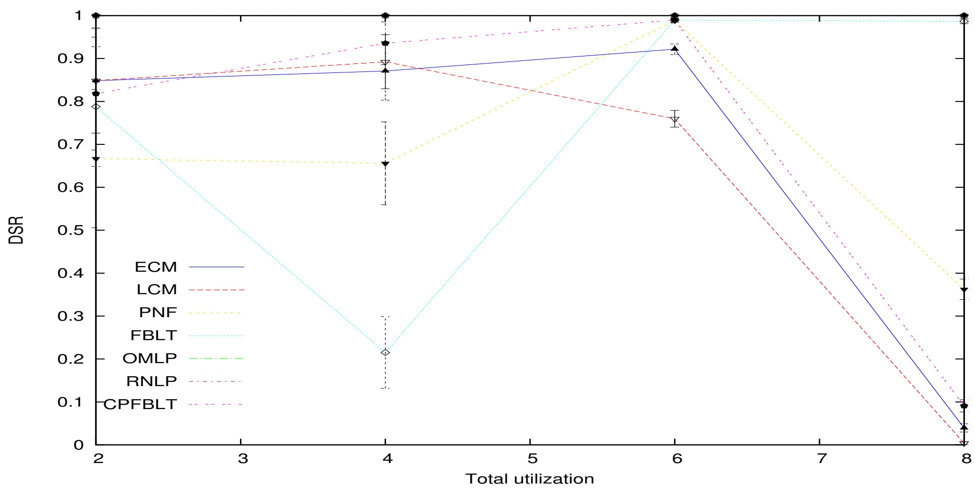


Figure B.81: DSR for Tasksets 81, 351, 621 and 891

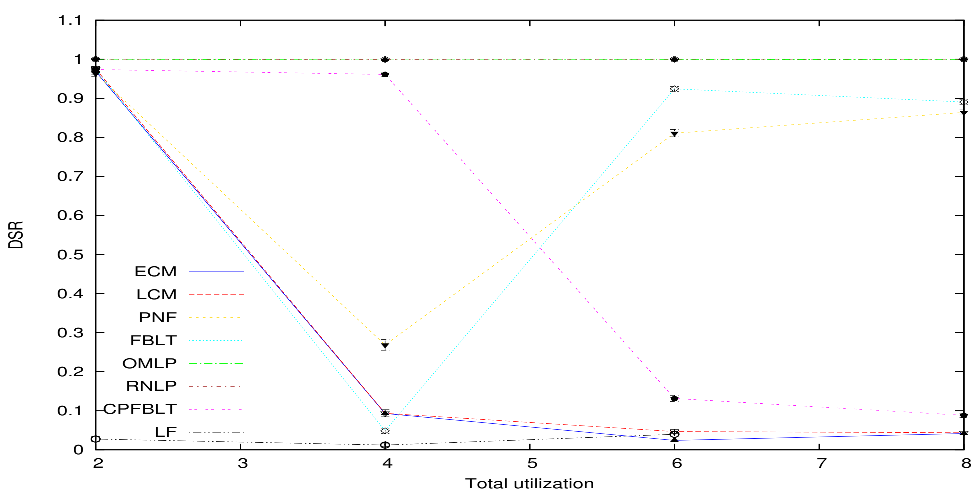


Figure B.82: DSR for Tasksets 82, 352, 622 and 892

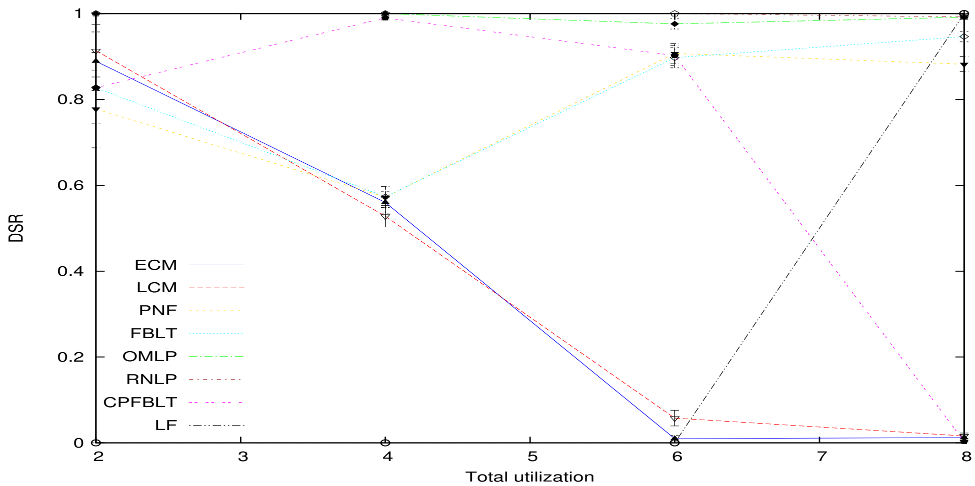


Figure B.83: DSR for Tasksets 83, 353, 623 and 893

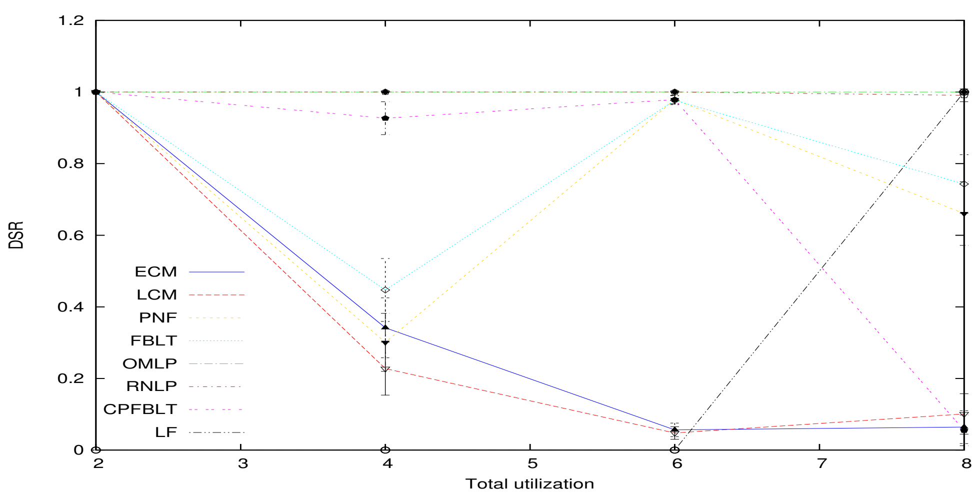


Figure B.84: DSR for Tasksets 84, 354, 624 and 894

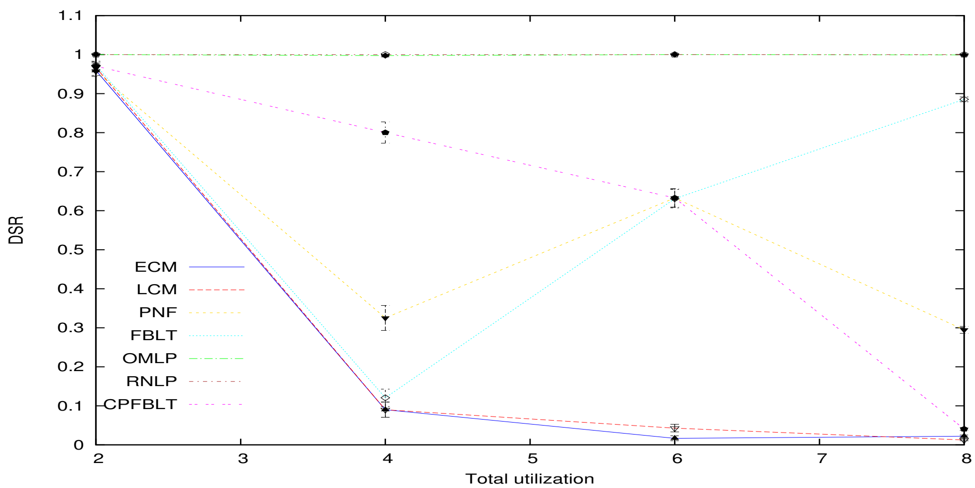


Figure B.85: DSR for Tasksets 85, 355, 625 and 895

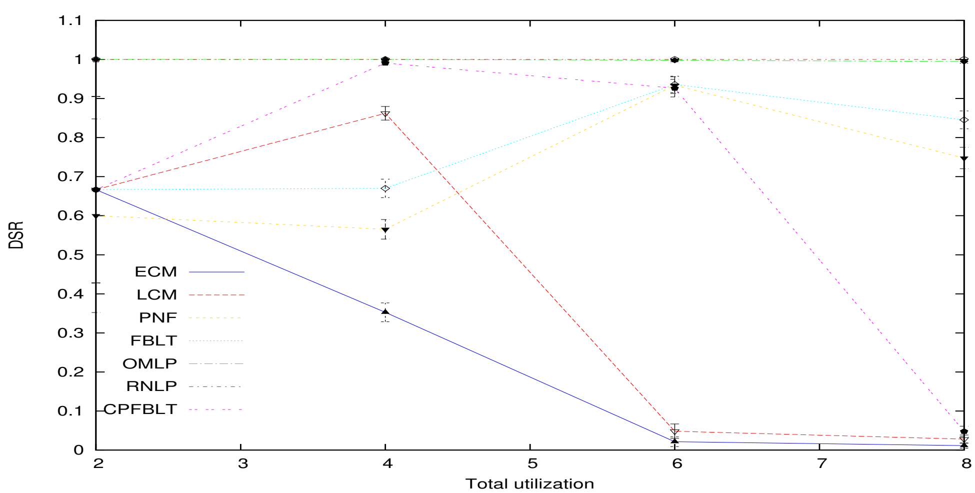


Figure B.86: DSR for Tasksets 86, 356, 626 and 896

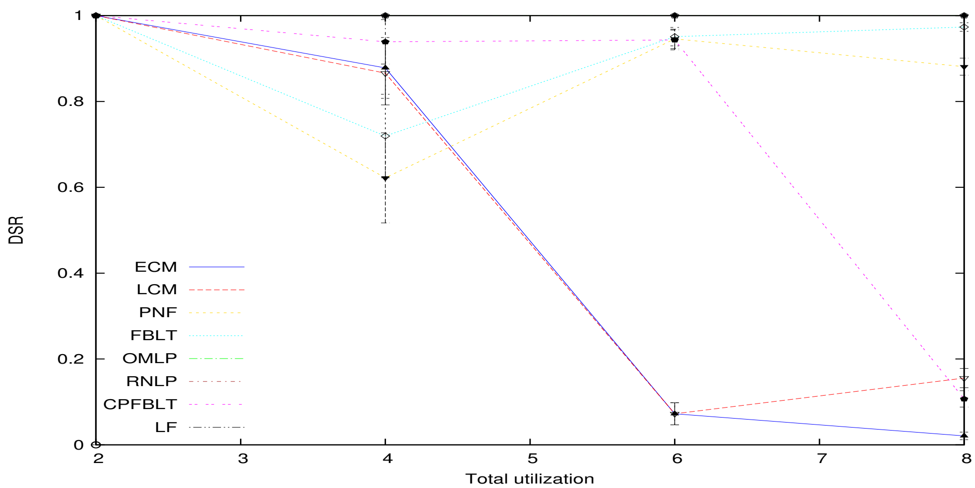


Figure B.87: DSR for Tasksets 87, 357, 627 and 897

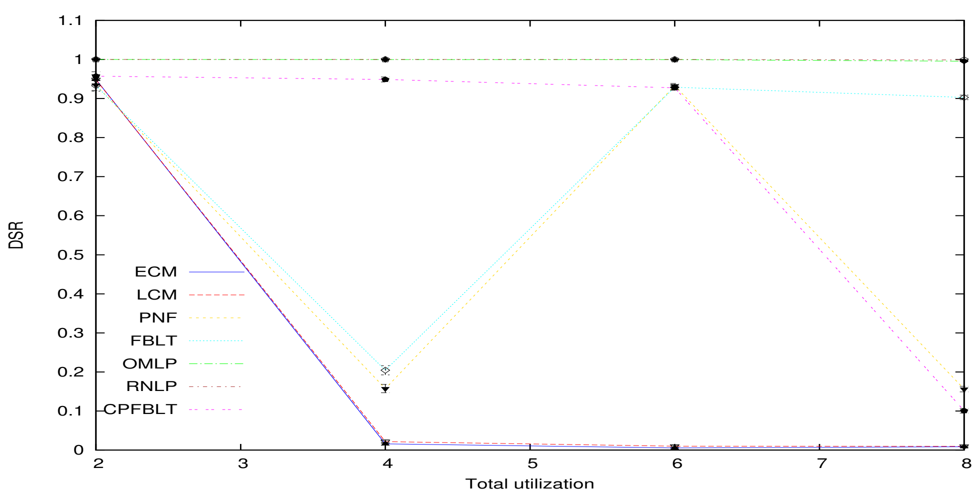


Figure B.88: DSR for Tasksets 88, 358, 628 and 898

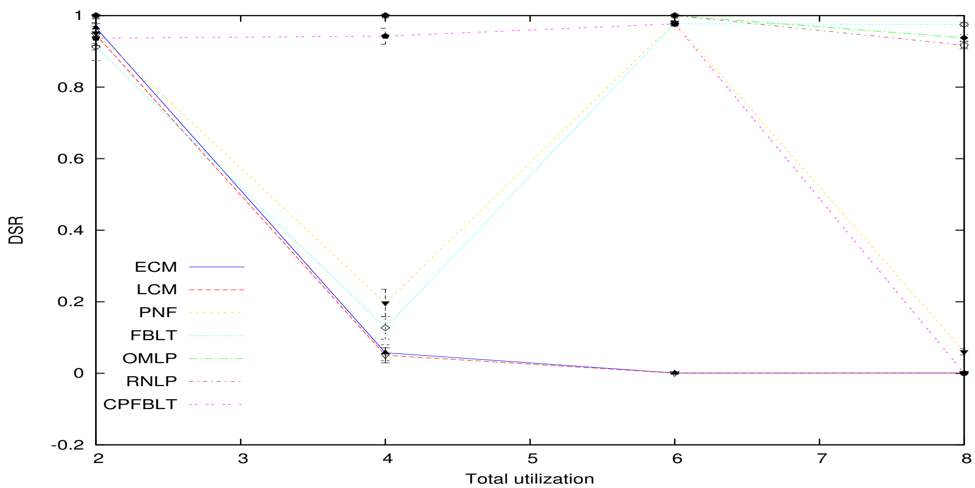


Figure B.89: DSR for Tasksets 89, 359, 629 and 899

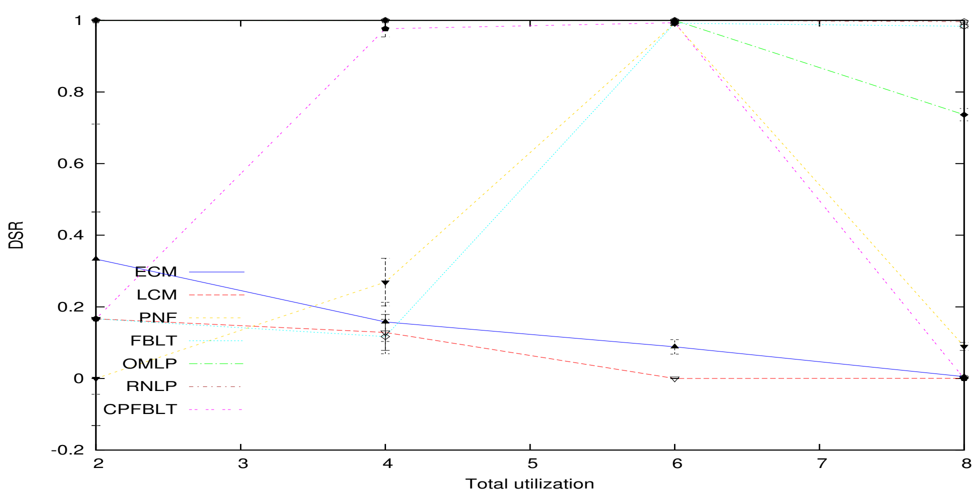


Figure B.90: DSR for Tasksets 90, 360, 630 and 900

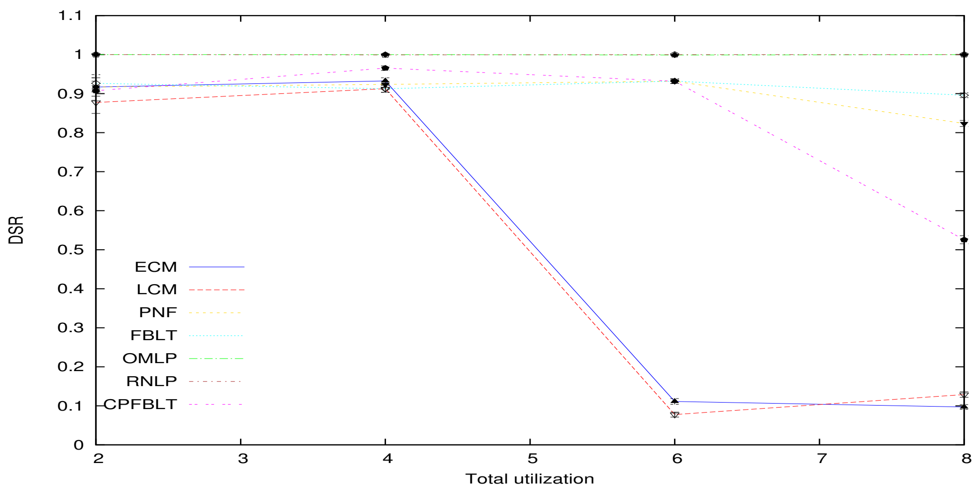


Figure B.91: DSR for Tasksets 91, 361, 631 and 901

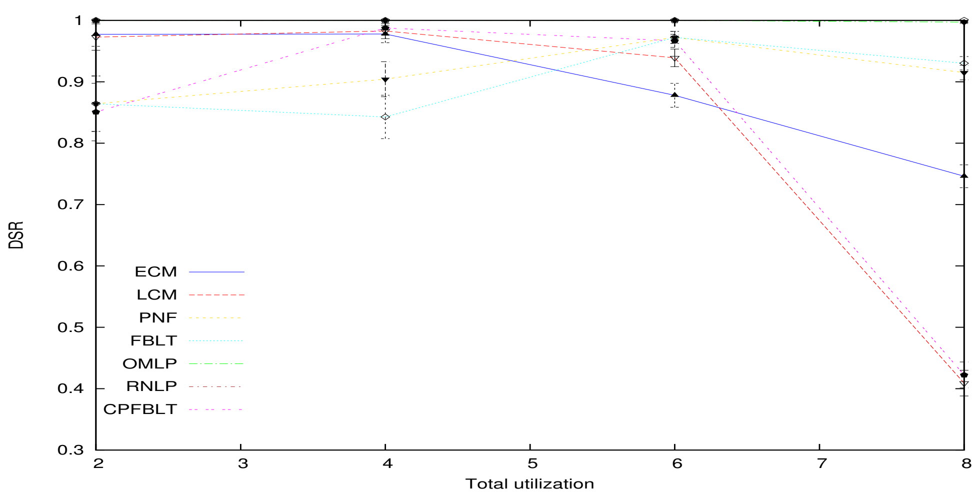


Figure B.92: DSR for Tasksets 92, 362, 632 and 902

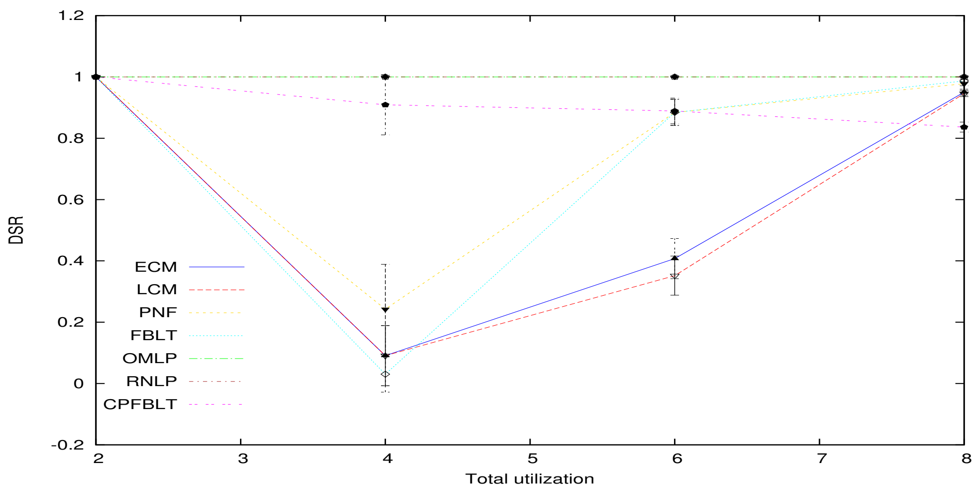


Figure B.93: DSR for Tasksets 93, 363, 633 and 903

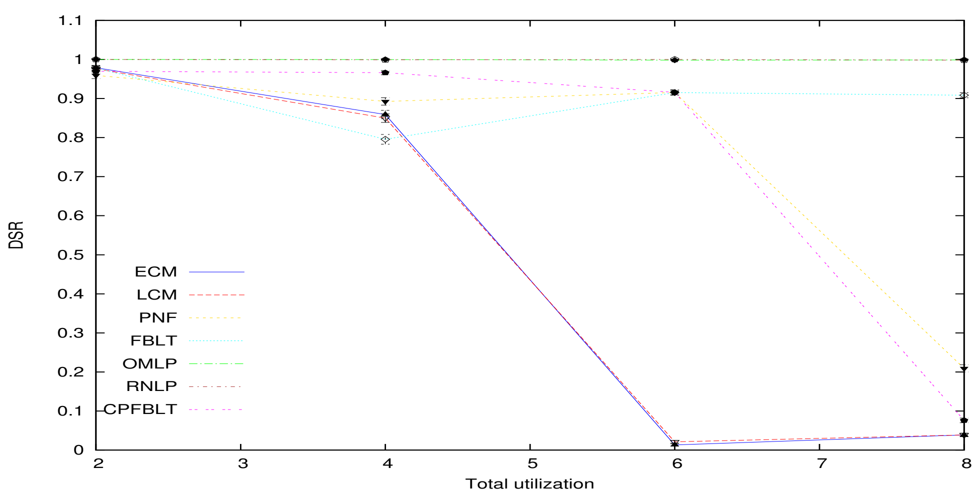


Figure B.94: DSR for Tasksets 94, 364, 634 and 904

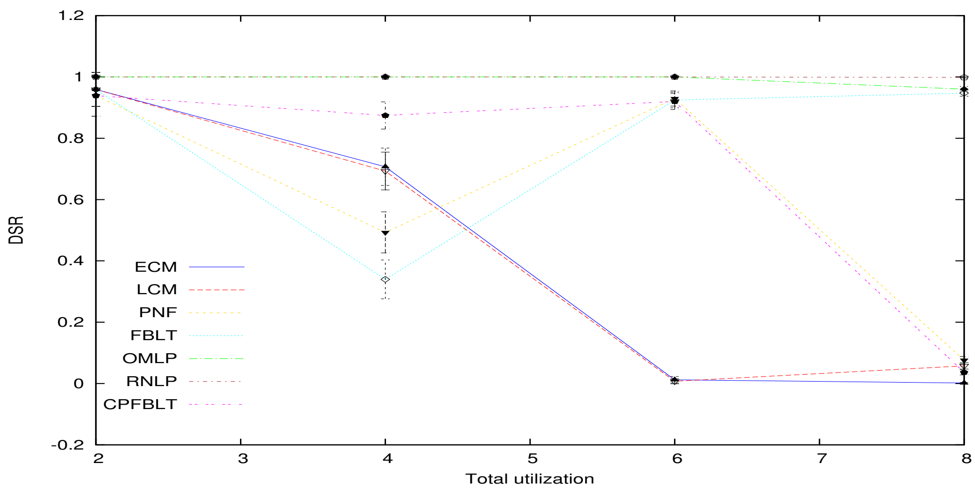


Figure B.95: DSR for Tasksets 95, 365, 635 and 905

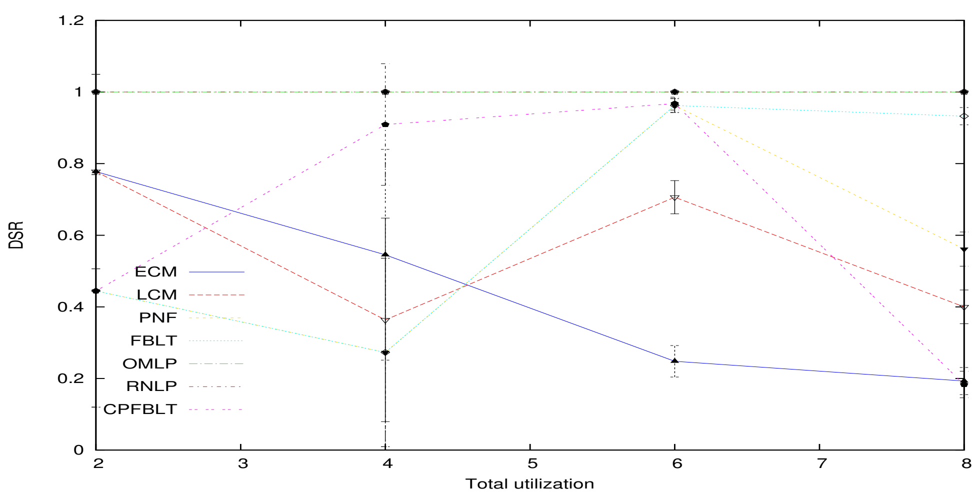


Figure B.96: DSR for Tasksets 96, 366, 636 and 906

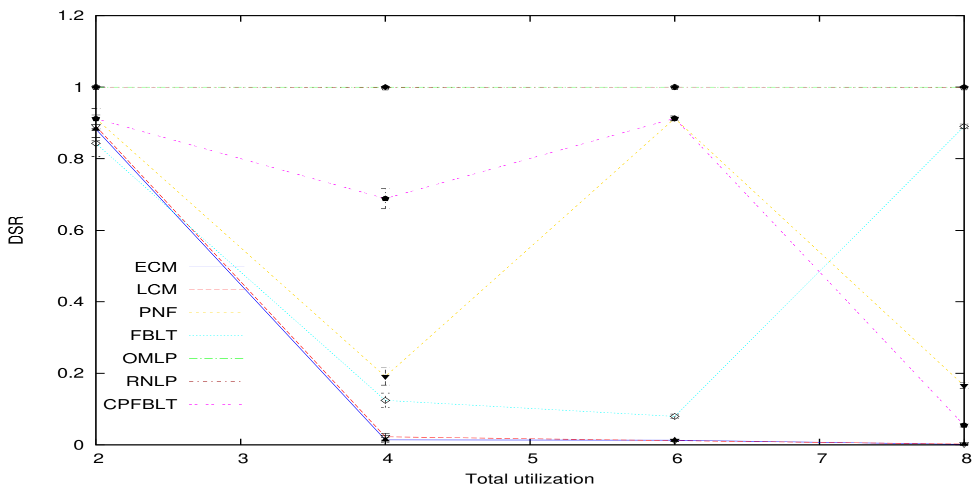


Figure B.97: DSR for Tasksets 97, 367, 637 and 907

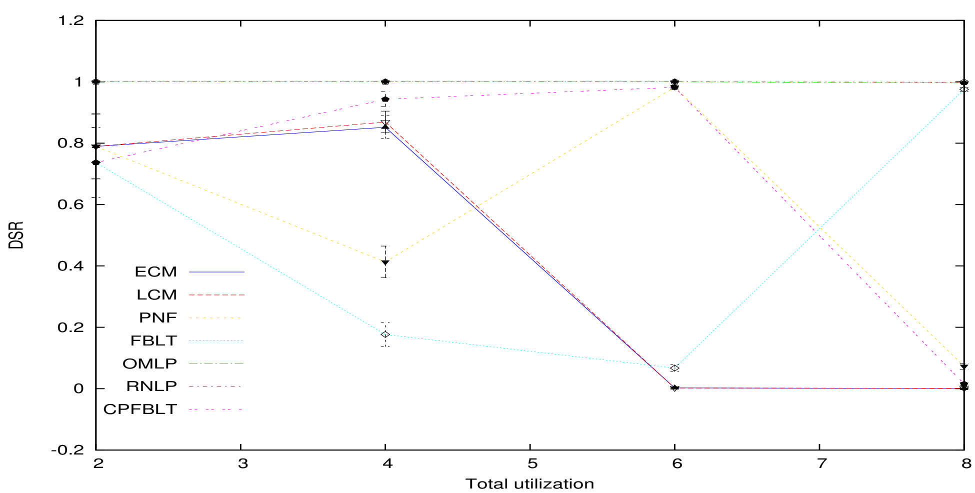


Figure B.98: DSR for Tasksets 98, 368, 638 and 908

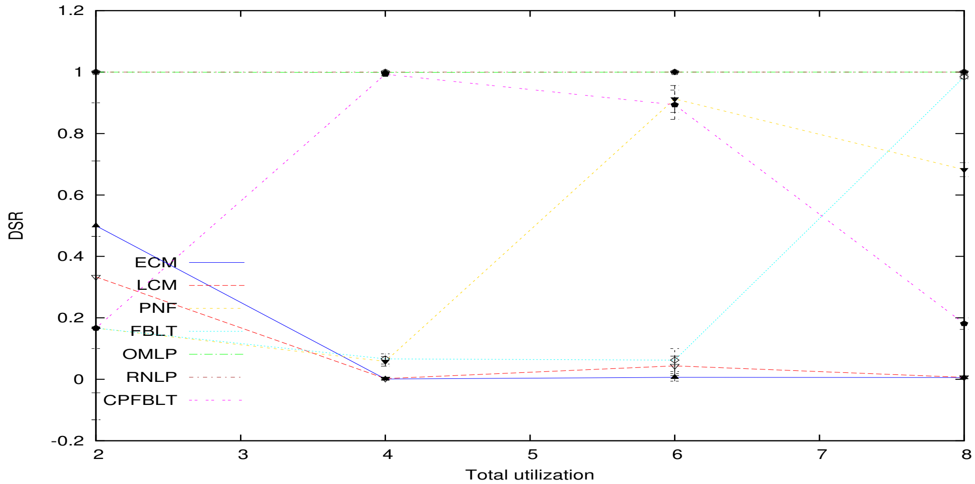


Figure B.99: DSR for Tasksets 99, 369, 639 and 909

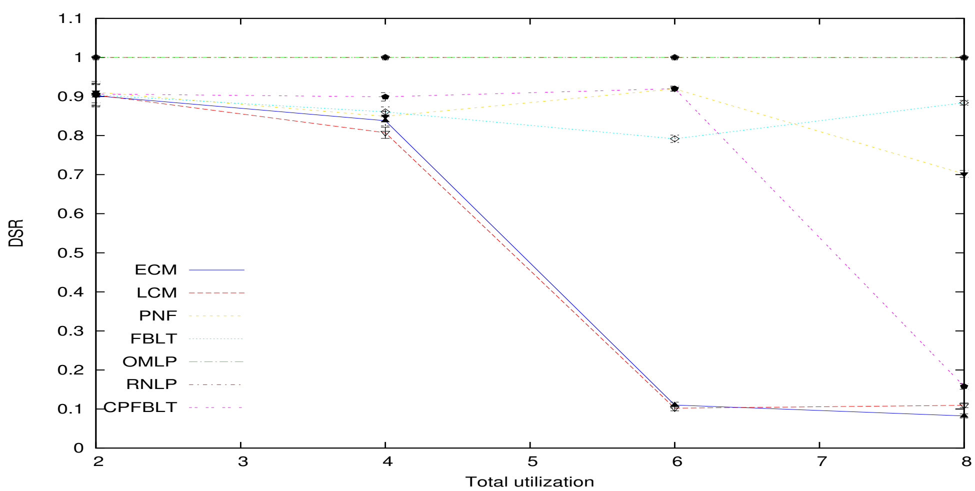


Figure B.100: DSR for Tasksets 100, 370, 640 and 910

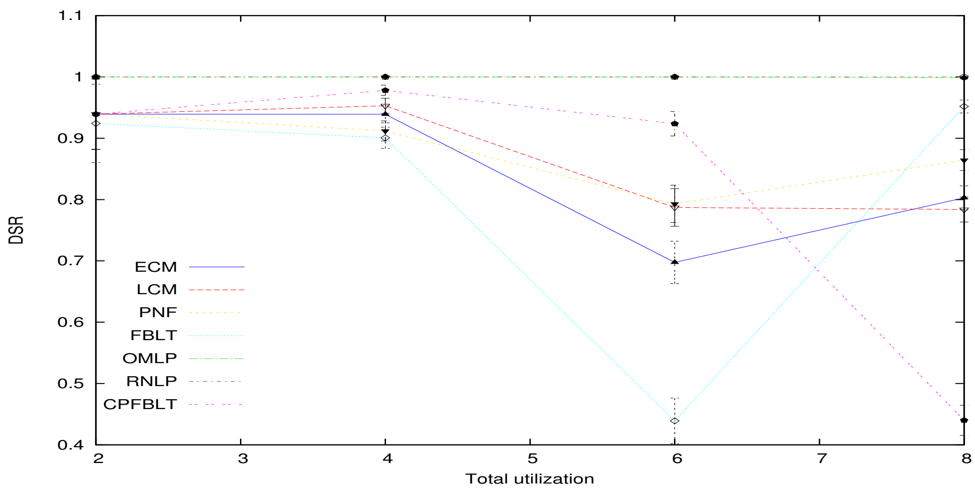


Figure B.101: DSR for Tasksets 101, 371, 641 and 911

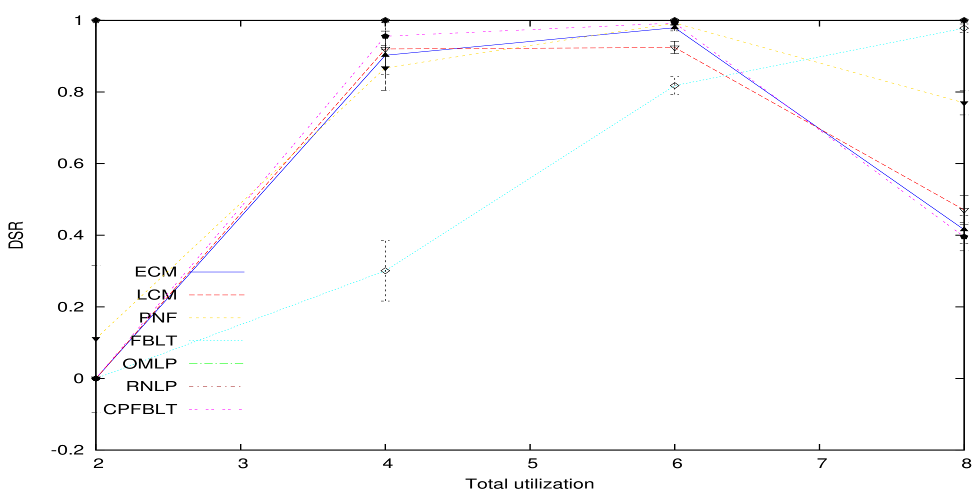


Figure B.102: DSR for Tasksets 102, 372, 642 and 912

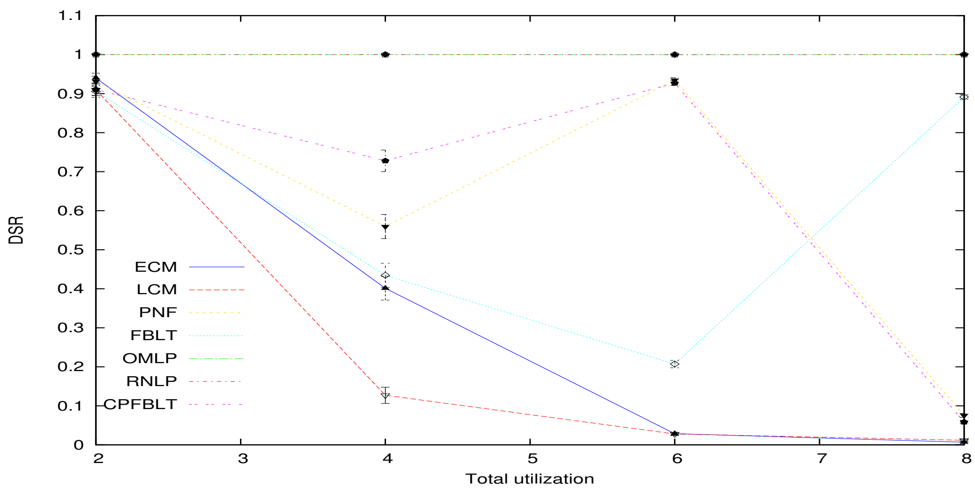


Figure B.103: DSR for Tasksets 103, 373, 643 and 913

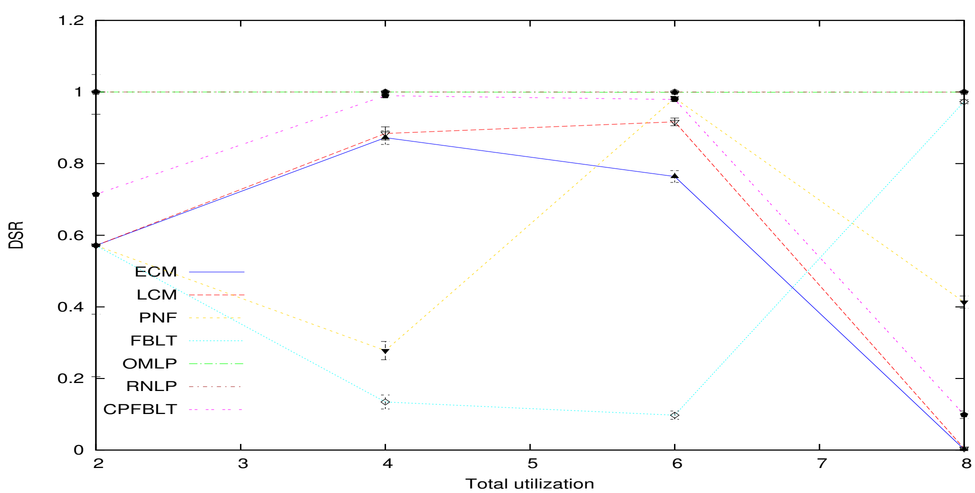


Figure B.104: DSR for Tasksets 104, 374, 644 and 914

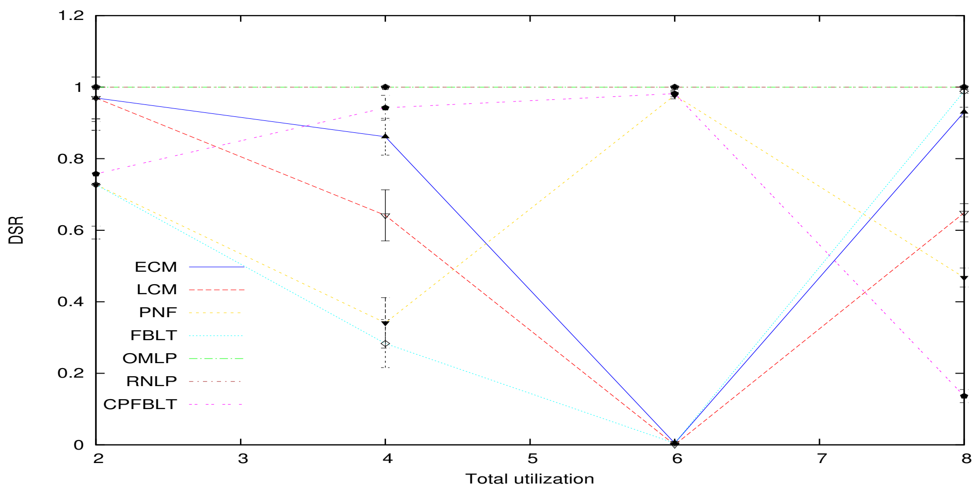


Figure B.105: DSR for Tasksets 105, 375, 645 and 915

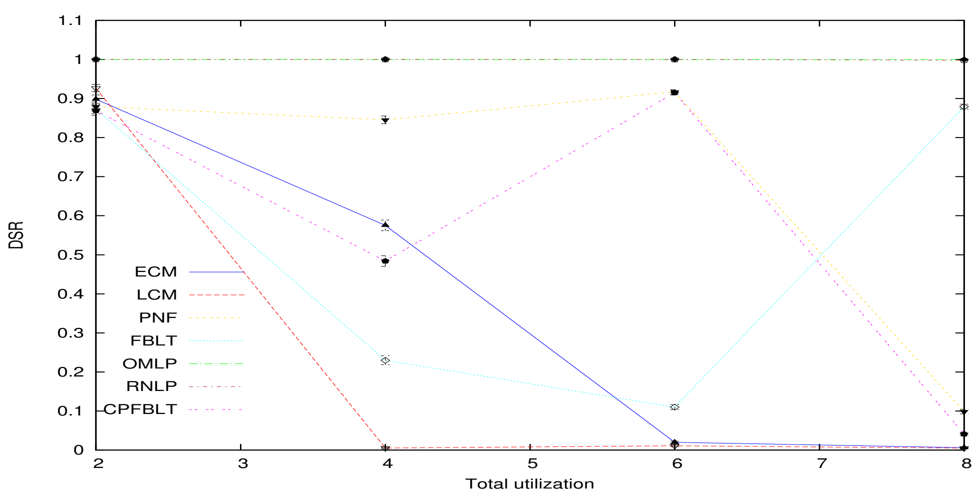


Figure B.106: DSR for Tasksets 106, 376, 646 and 916

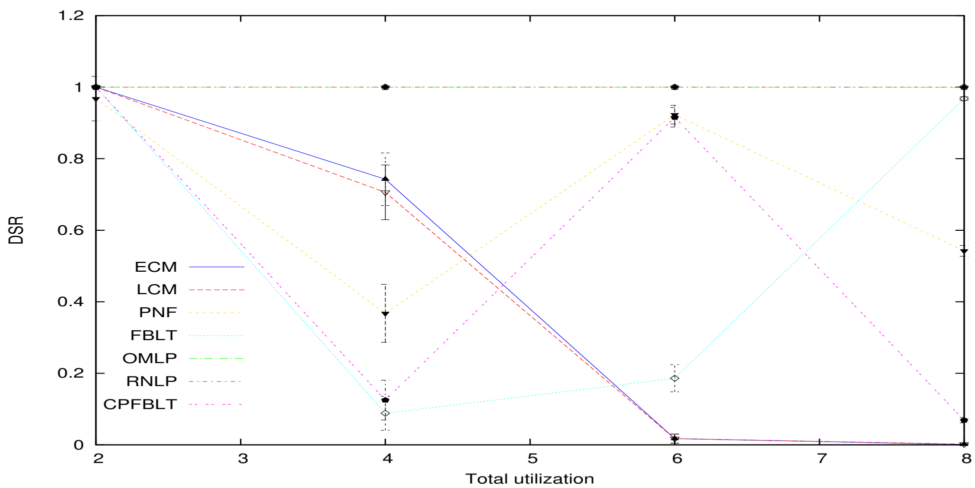


Figure B.107: DSR for Tasksets 107, 377, 647 and 917

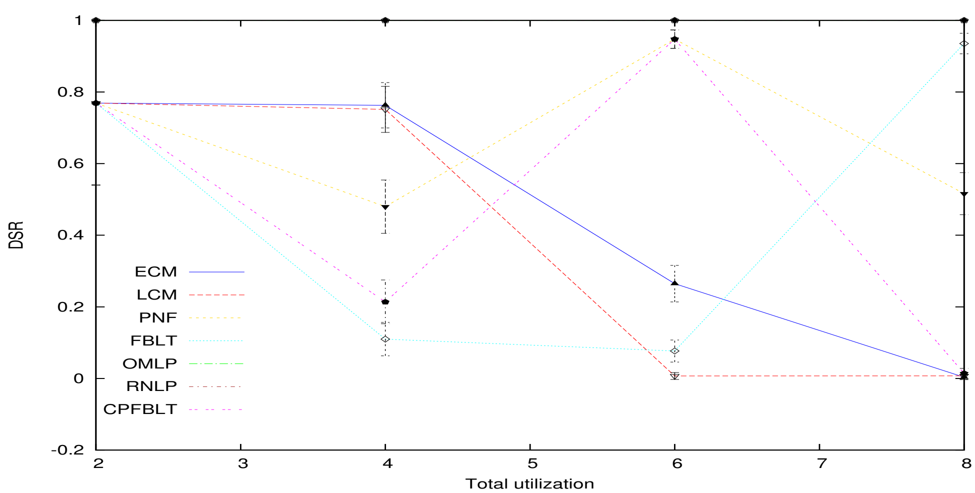


Figure B.108: DSR for Tasksets 108, 378, 648 and 918

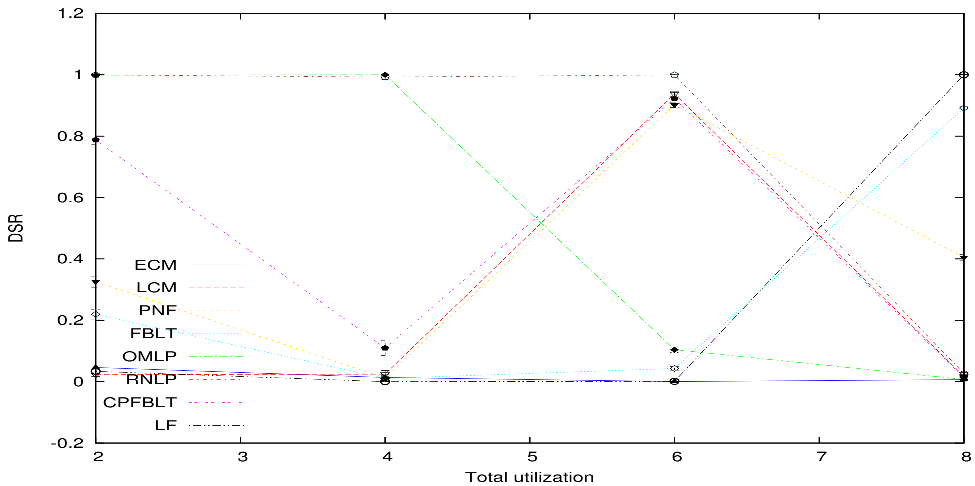


Figure B.109: DSR for Tasksets 109, 379, 649 and 919

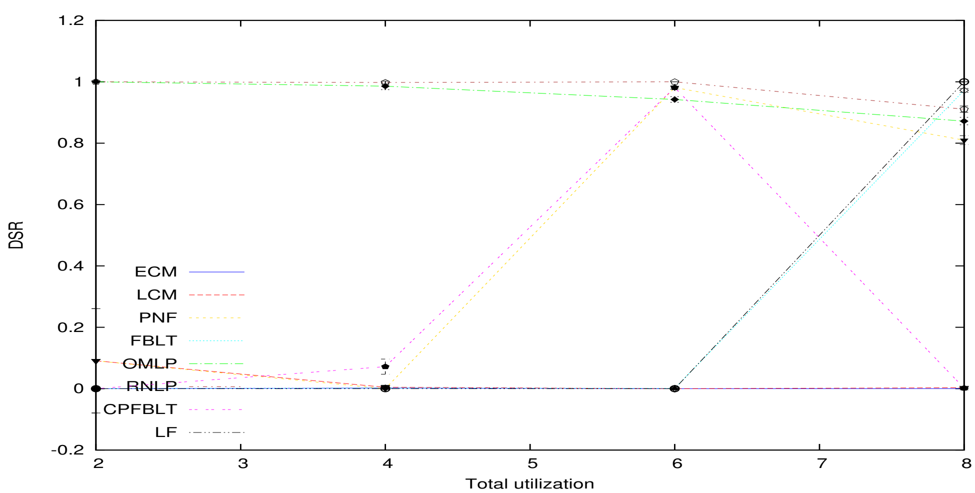


Figure B.110: DSR for Tasksets 110, 380, 650 and 920

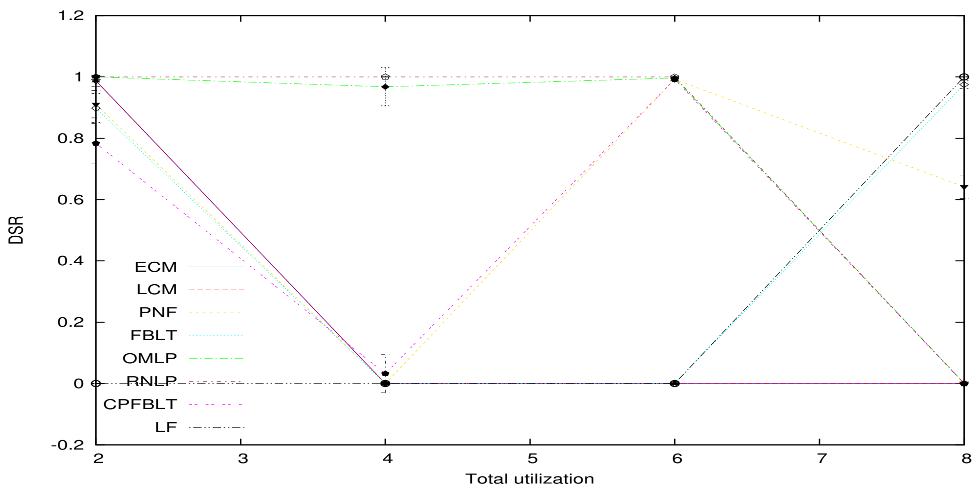


Figure B.111: DSR for Tasksets 111, 381, 651 and 921

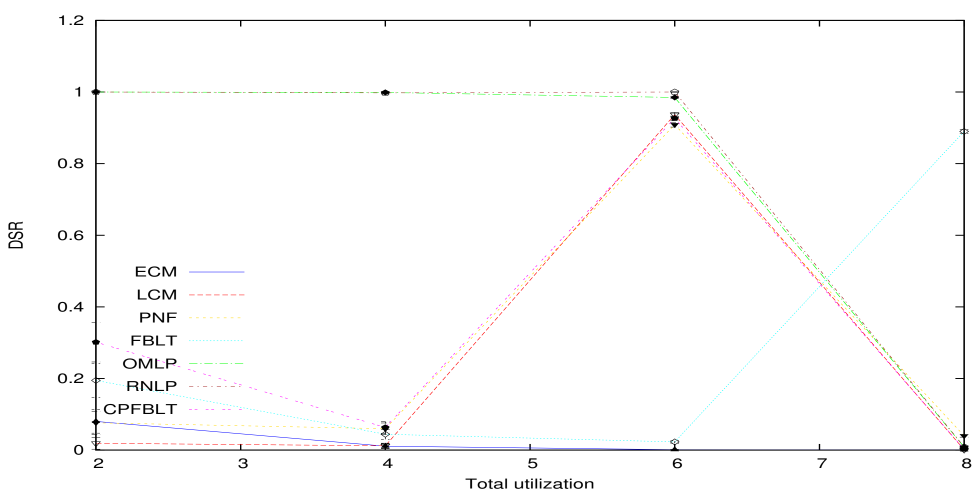


Figure B.112: DSR for Tasksets 112, 382, 652 and 922

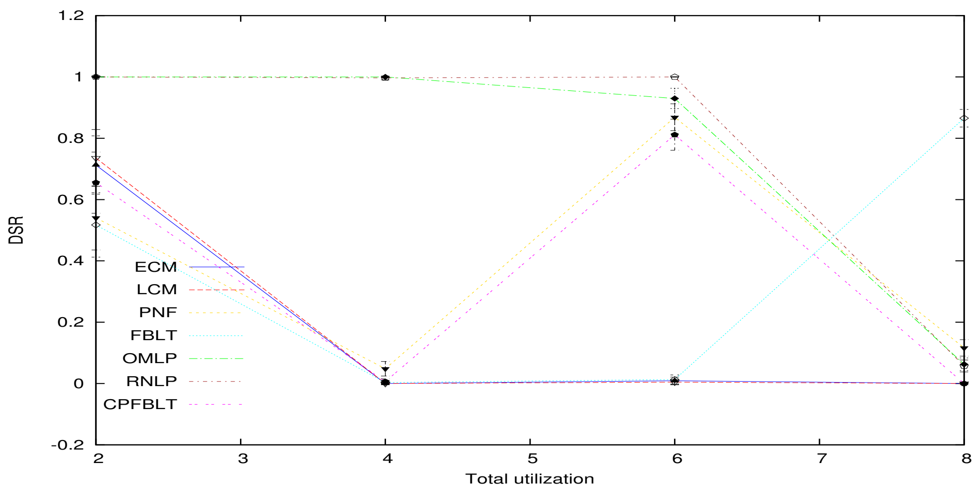


Figure B.113: DSR for Tasksets 113, 383, 653 and 923

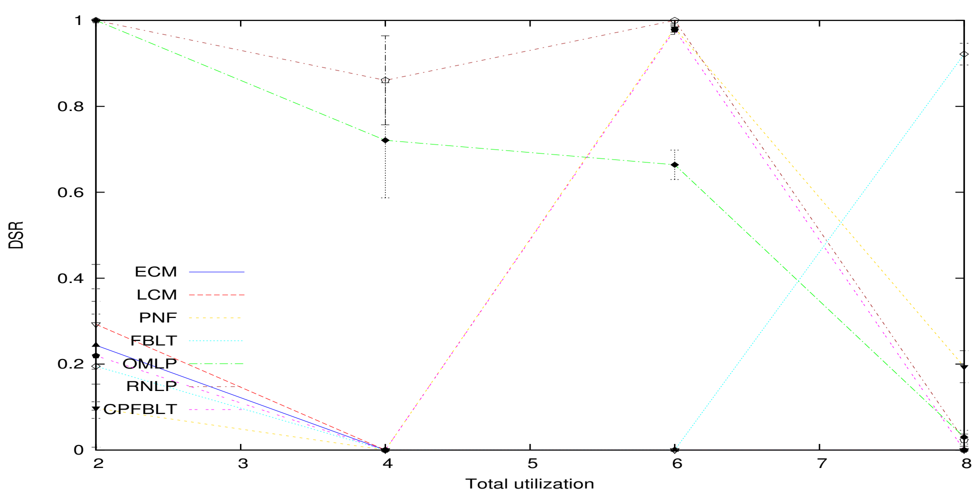


Figure B.114: DSR for Tasksets 114, 384, 654 and 924

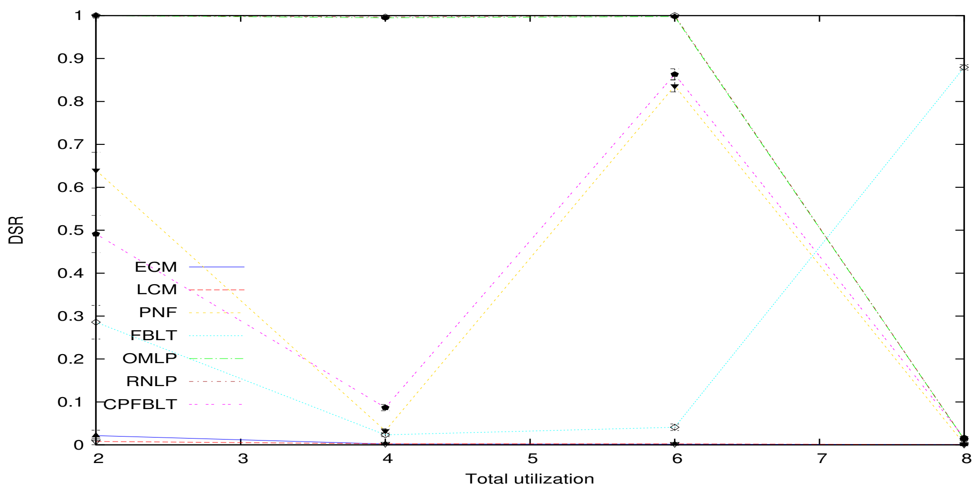


Figure B.115: DSR for Tasksets 115, 385, 655 and 925

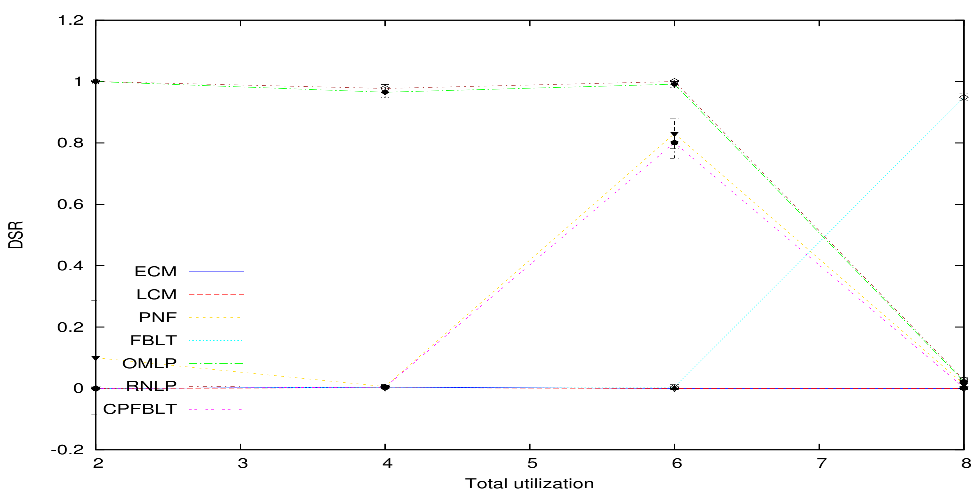


Figure B.116: DSR for Tasksets 116, 386, 656 and 926

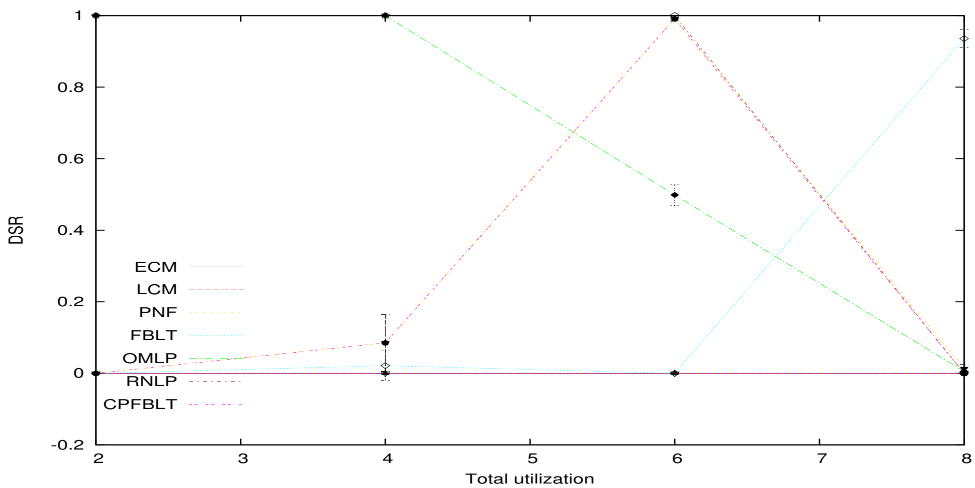


Figure B.117: DSR for Tasksets 117, 387, 657 and 927

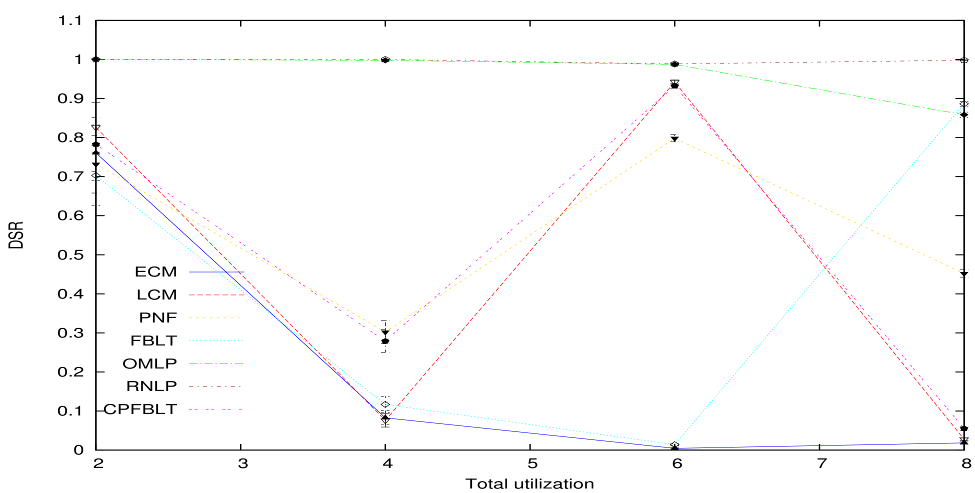


Figure B.118: DSR for Tasksets 118, 388, 658 and 928

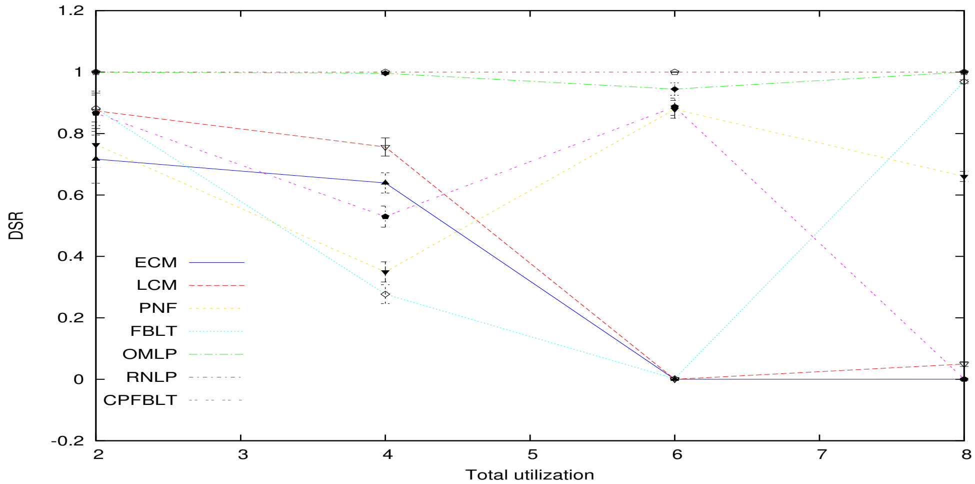


Figure B.119: DSR for Tasksets 119, 389, 659 and 929

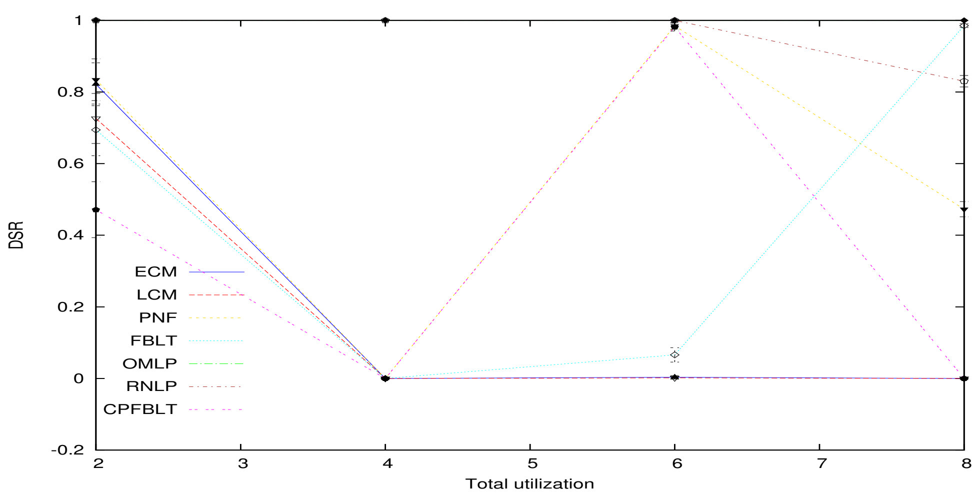


Figure B.120: DSR for Tasksets 120, 390, 660 and 930

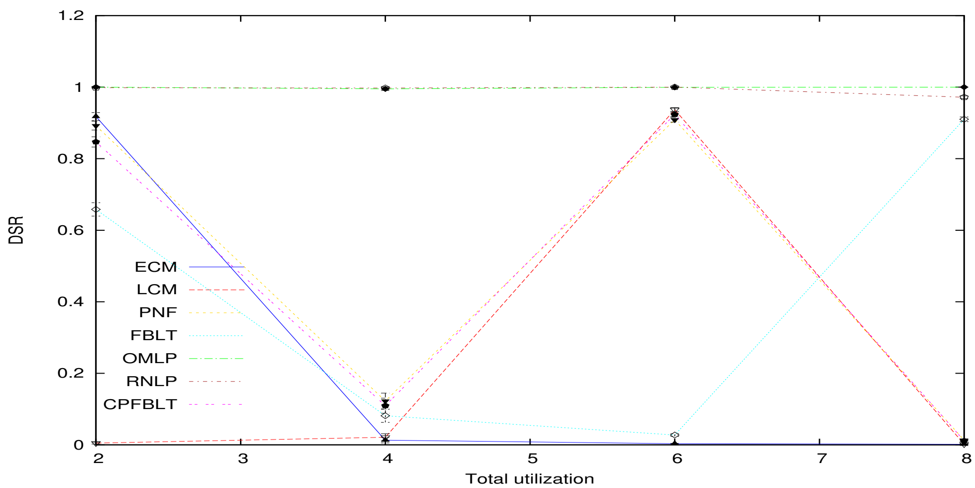


Figure B.121: DSR for Tasksets 121, 391, 661 and 931

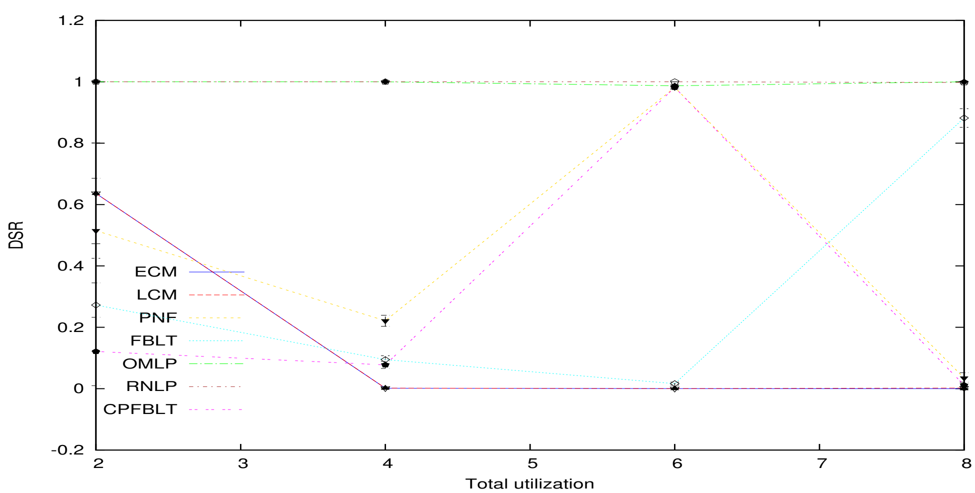


Figure B.122: DSR for Tasksets 122, 392, 662 and 932

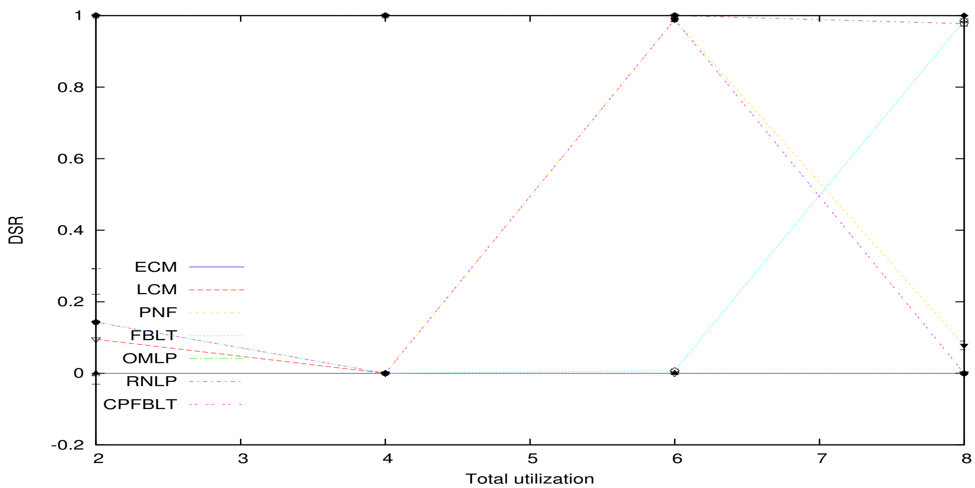


Figure B.123: DSR for Tasksets 123, 393, 663 and 933

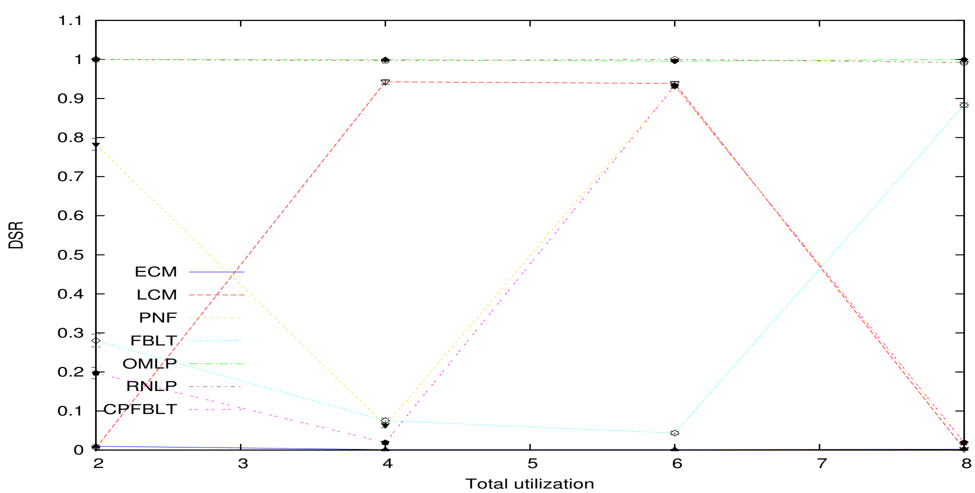


Figure B.124: DSR for Tasksets 124, 394, 664 and 934

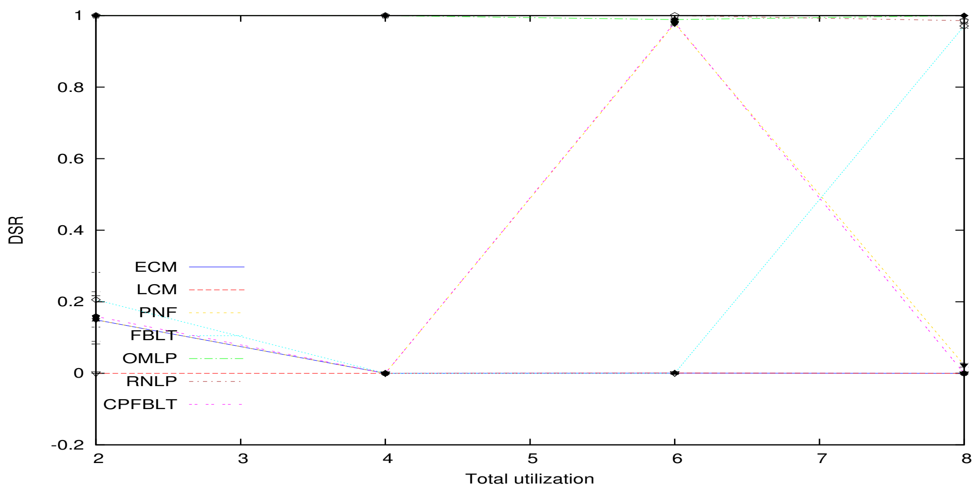


Figure B.125: DSR for Tasksets 125, 395, 665 and 935

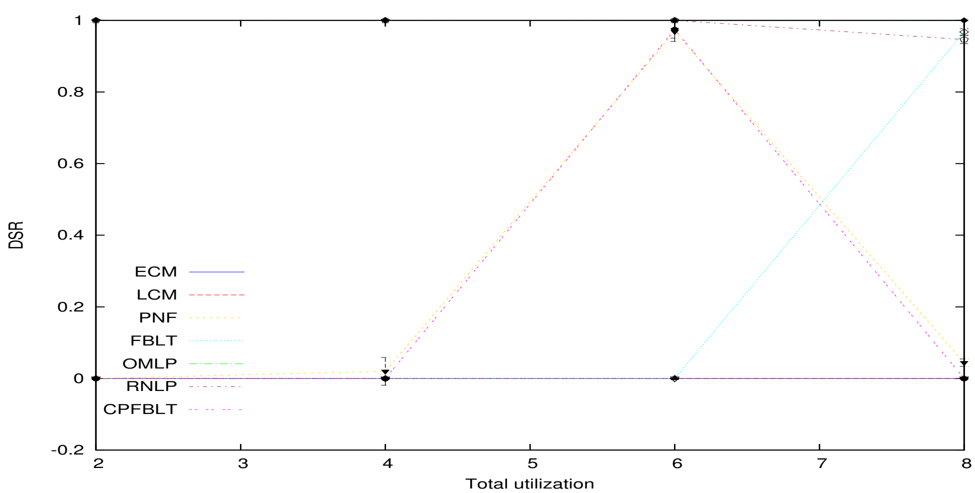


Figure B.126: DSR for Tasksets 126, 396, 666 and 936

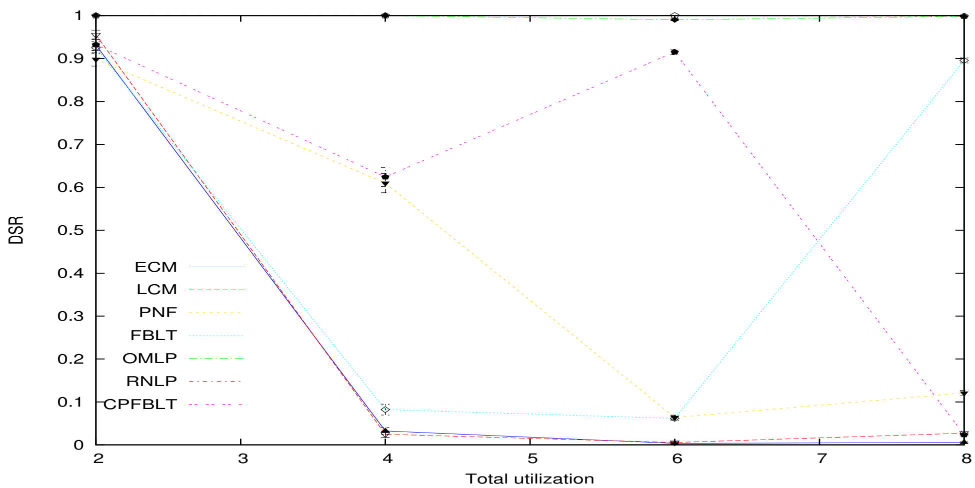


Figure B.127: DSR for Tasksets 127, 397, 667 and 937

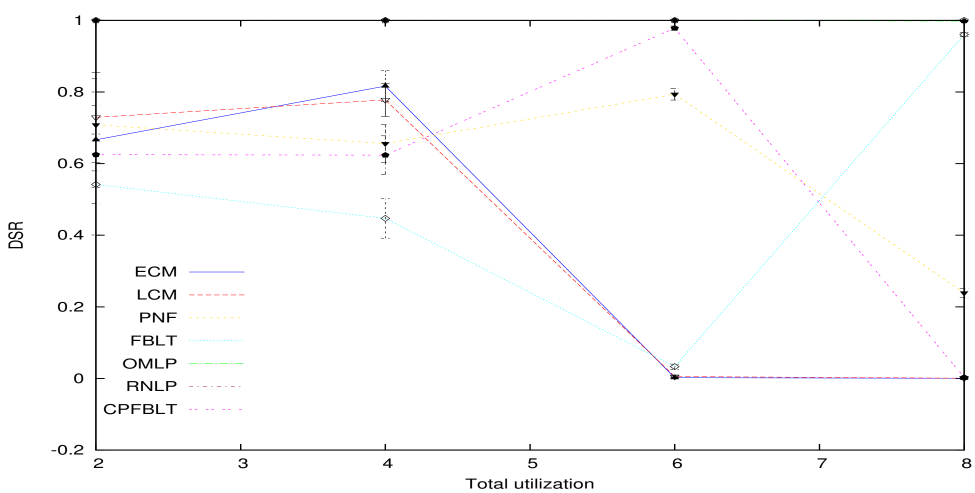


Figure B.128: DSR for Tasksets 128, 398, 668 and 938

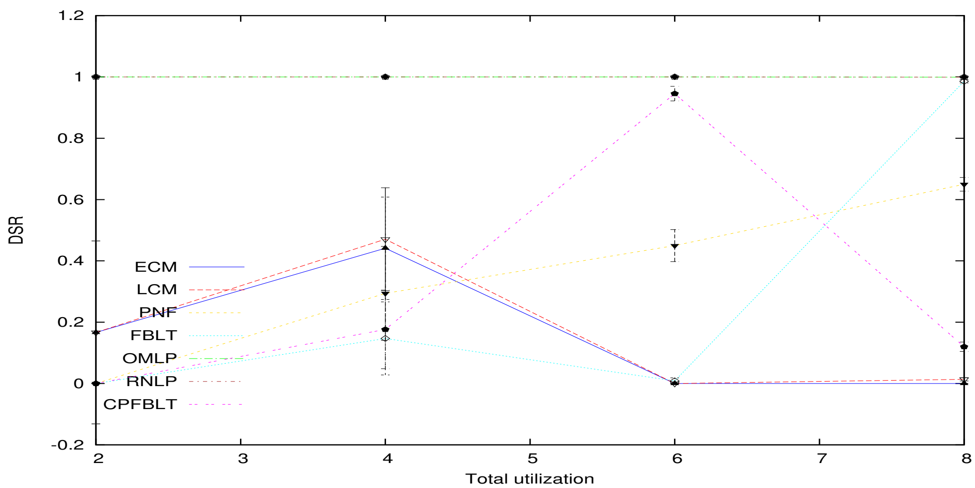


Figure B.129: DSR for Tasksets 129, 399, 669 and 939

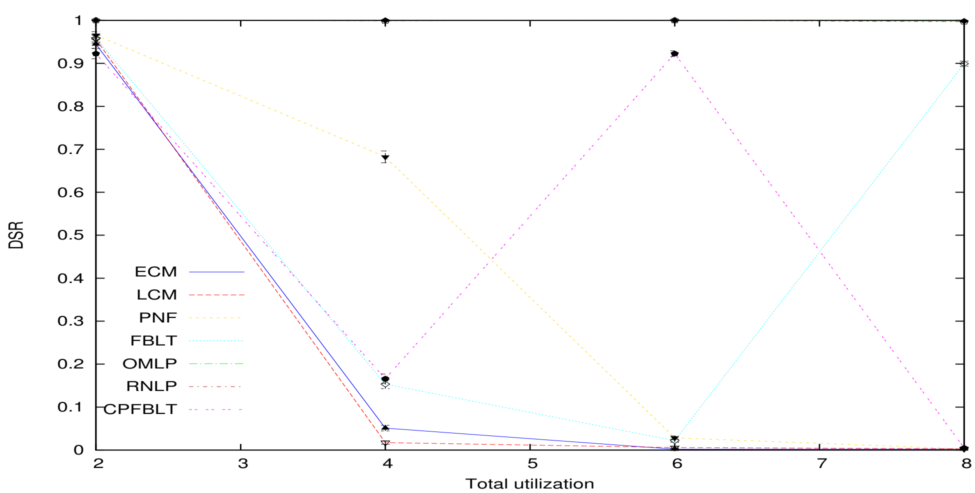


Figure B.130: DSR for Tasksets 130, 400, 670 and 940

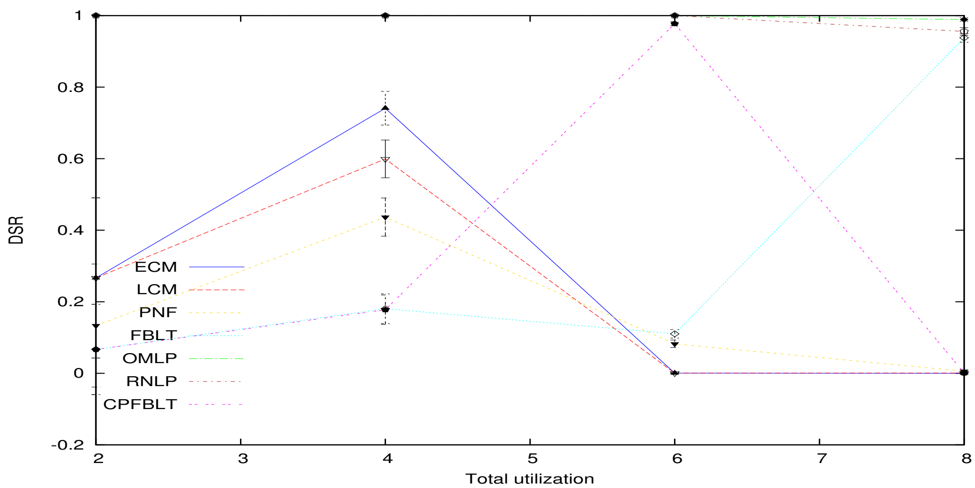


Figure B.131: DSR for Tasksets 131, 401, 671 and 941

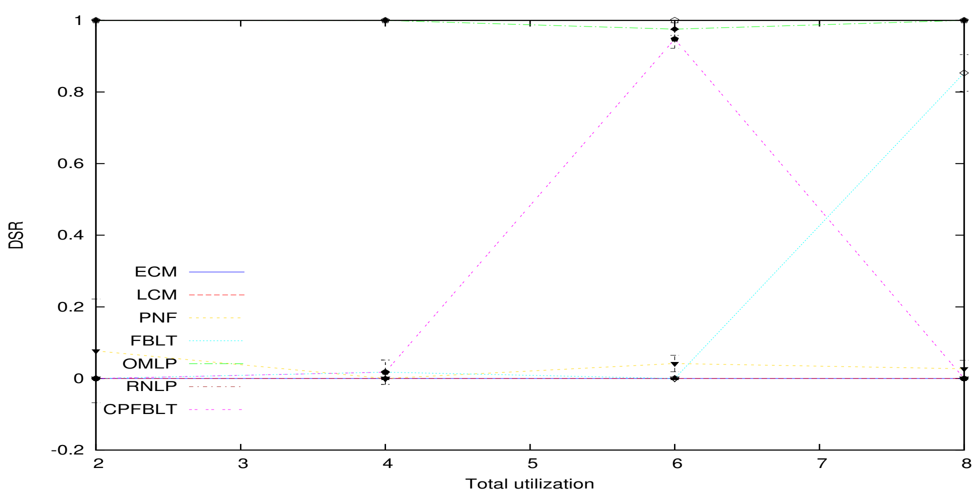


Figure B.132: DSR for Tasksets 132, 402, 672 and 942

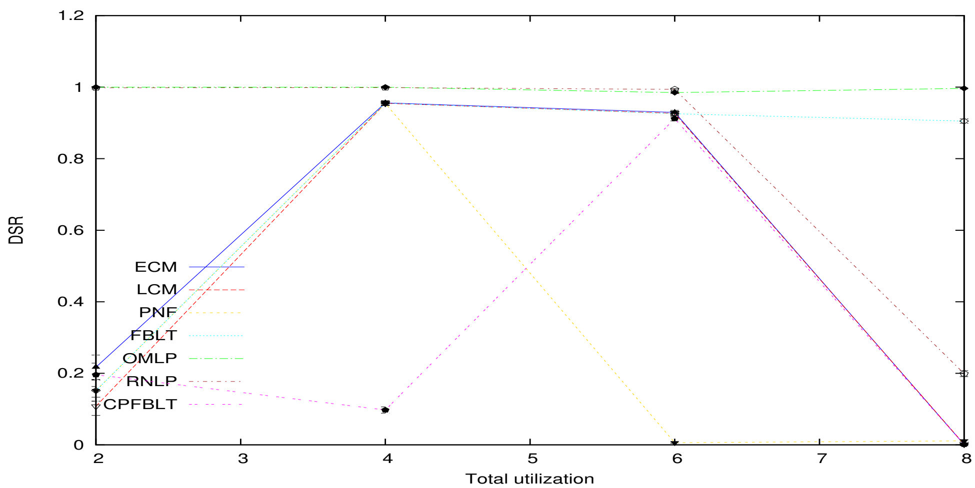


Figure B.133: DSR for Tasksets 133, 403, 673 and 943

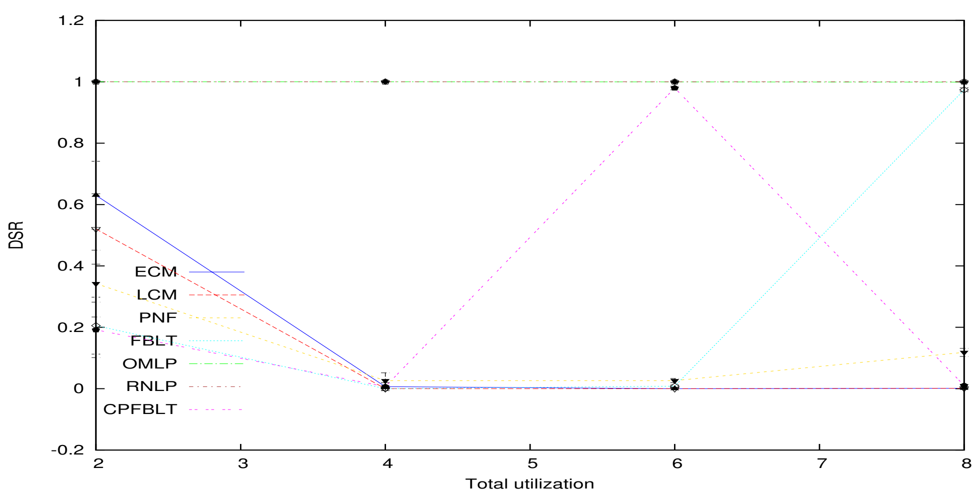


Figure B.134: DSR for Tasksets 134, 404, 674 and 944

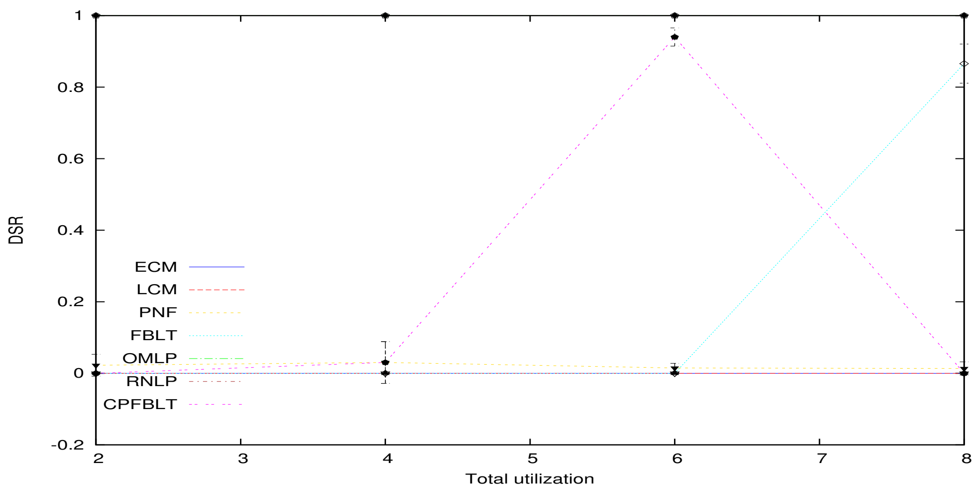


Figure B.135: DSR for Tasksets 135, 405, 675 and 945

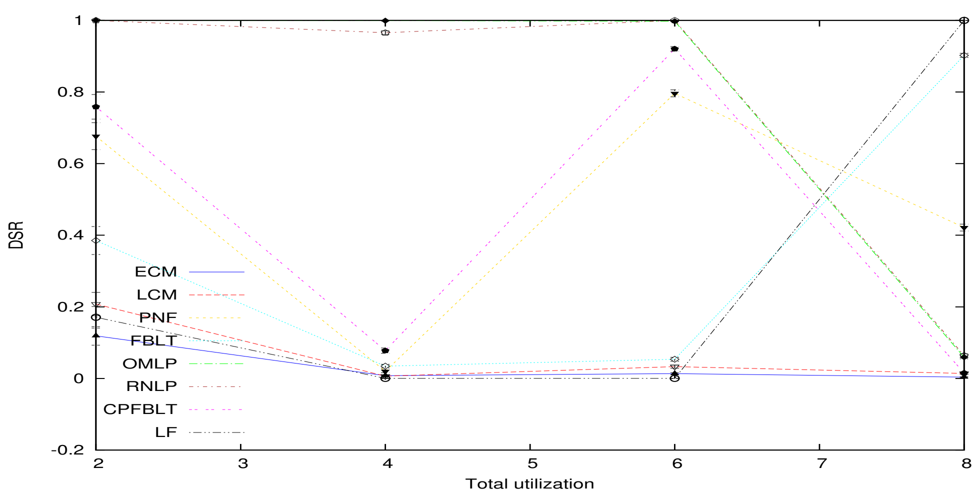


Figure B.136: DSR for Tasksets 136, 406, 676 and 946

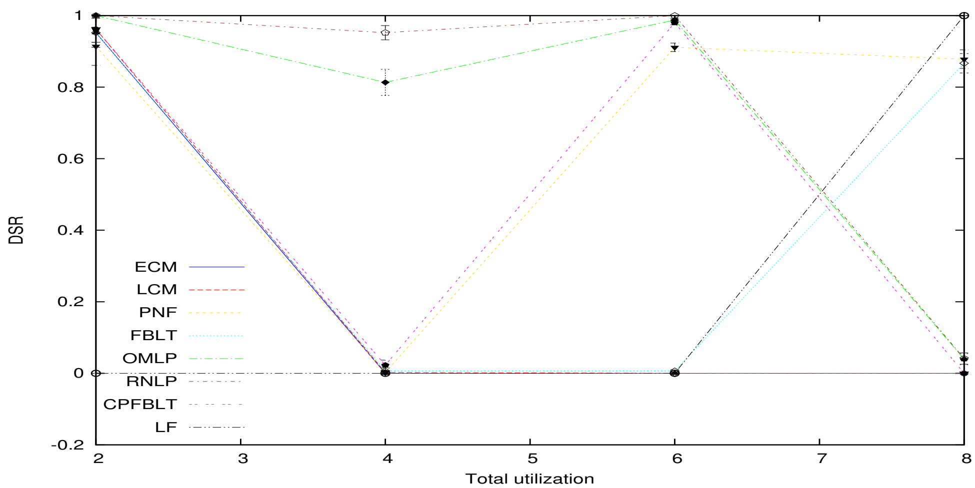


Figure B.137: DSR for Tasksets 137, 407, 677 and 947

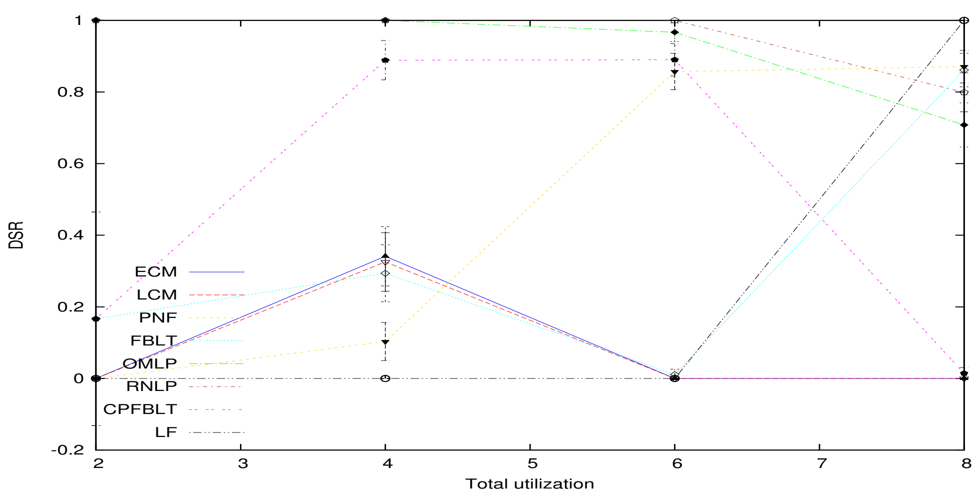


Figure B.138: DSR for Tasksets 138, 408, 678 and 948

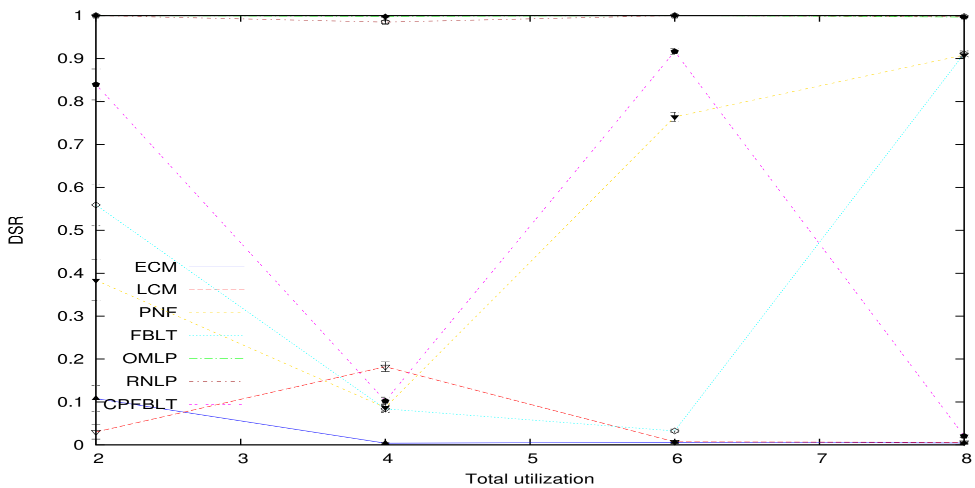


Figure B.139: DSR for Tasksets 139, 409, 679 and 949

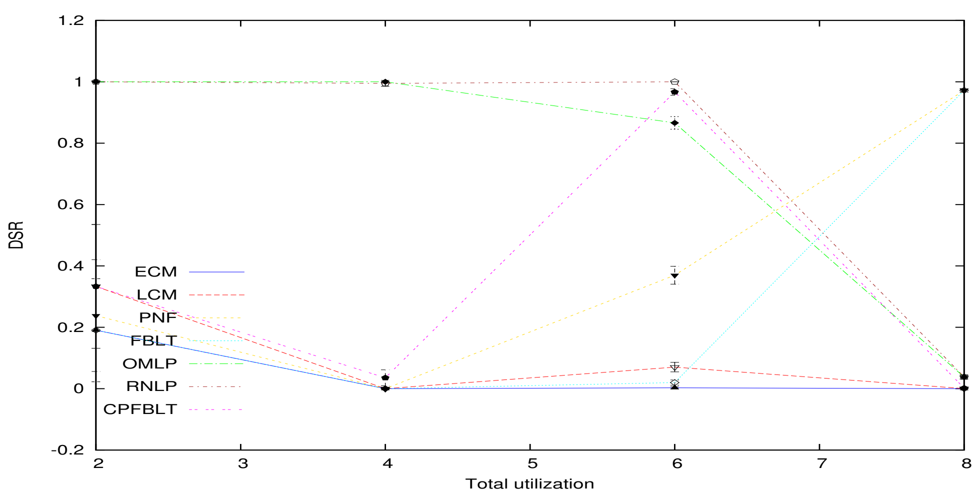


Figure B.140: DSR for Tasksets 140, 410, 680 and 950

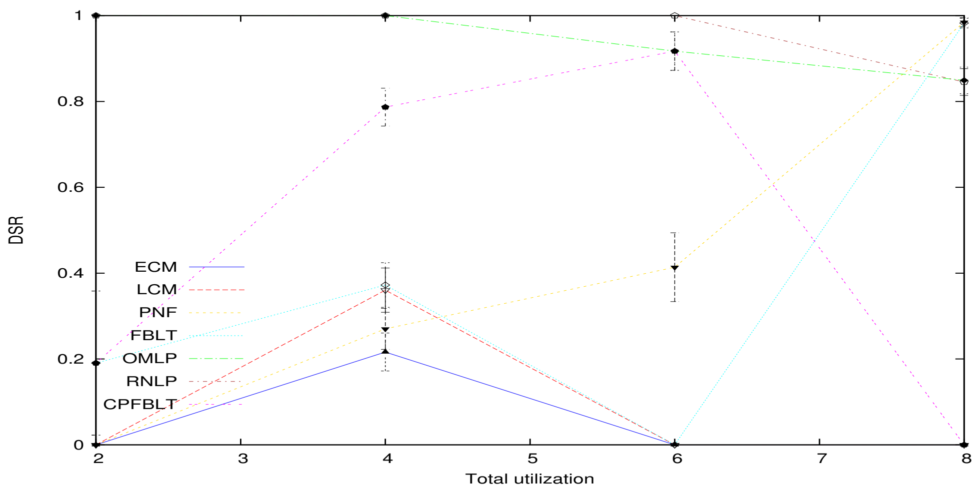


Figure B.141: DSR for Tasksets 141, 411, 681 and 951

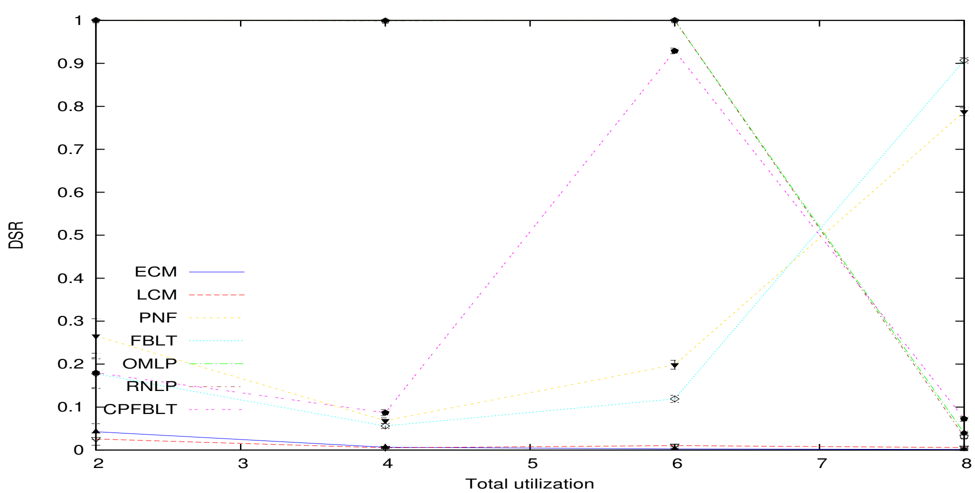


Figure B.142: DSR for Tasksets 142, 412, 682 and 952

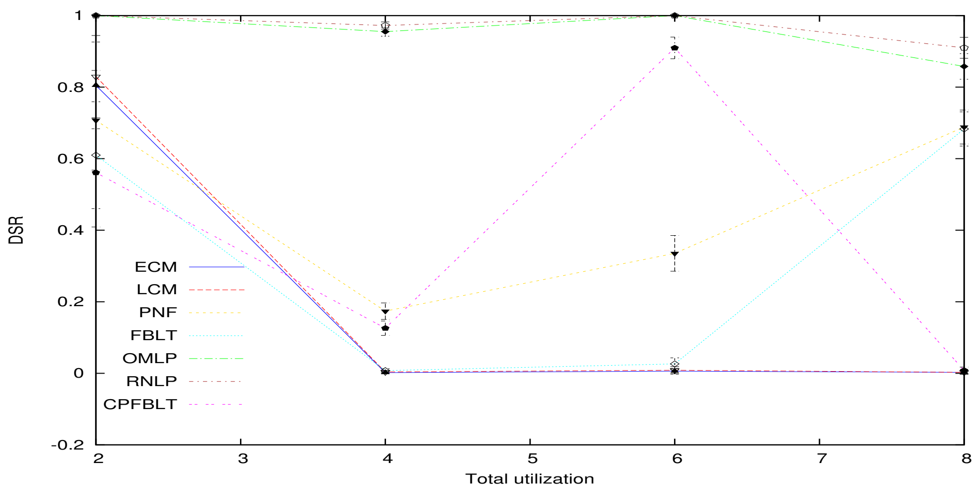


Figure B.143: DSR for Tasksets 143, 413, 683 and 953

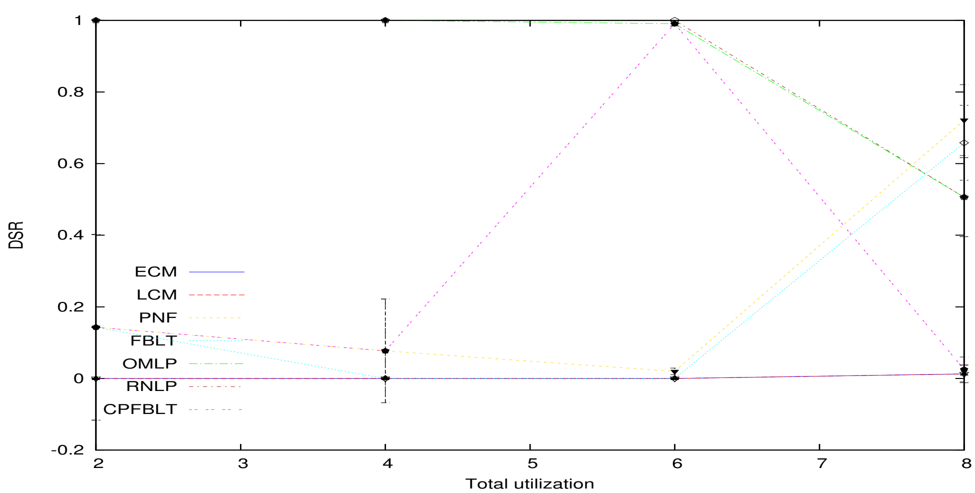


Figure B.144: DSR for Tasksets 144, 414, 684 and 954

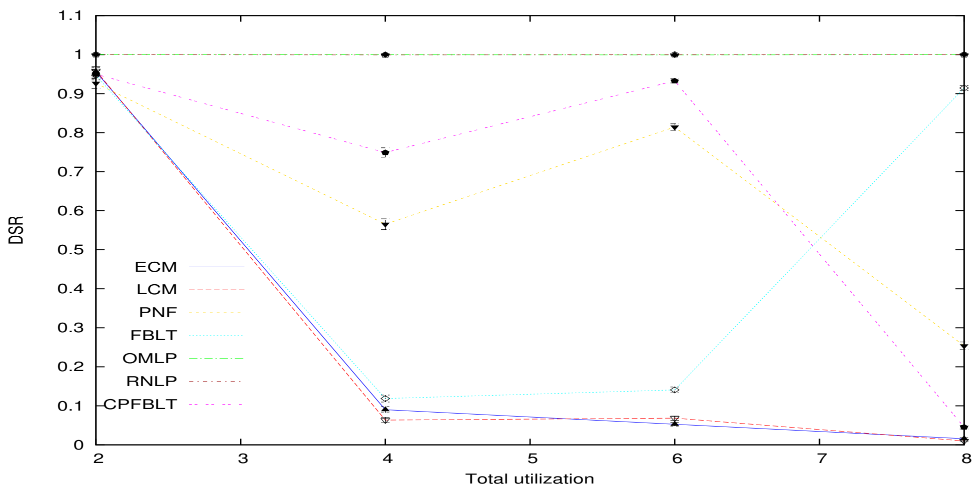


Figure B.145: DSR for Tasksets 145, 415, 685 and 955

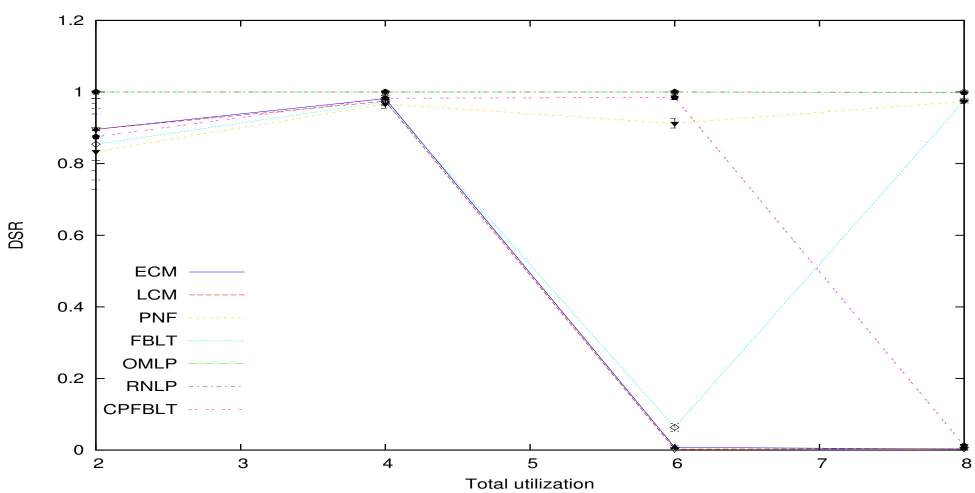


Figure B.146: DSR for Tasksets 146, 416, 686 and 956

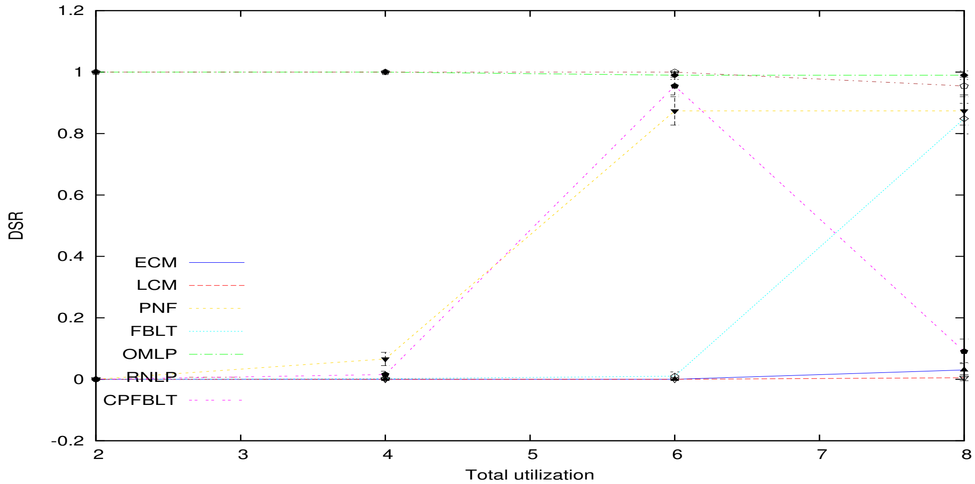


Figure B.147: DSR for Tasksets 147, 417, 687 and 957

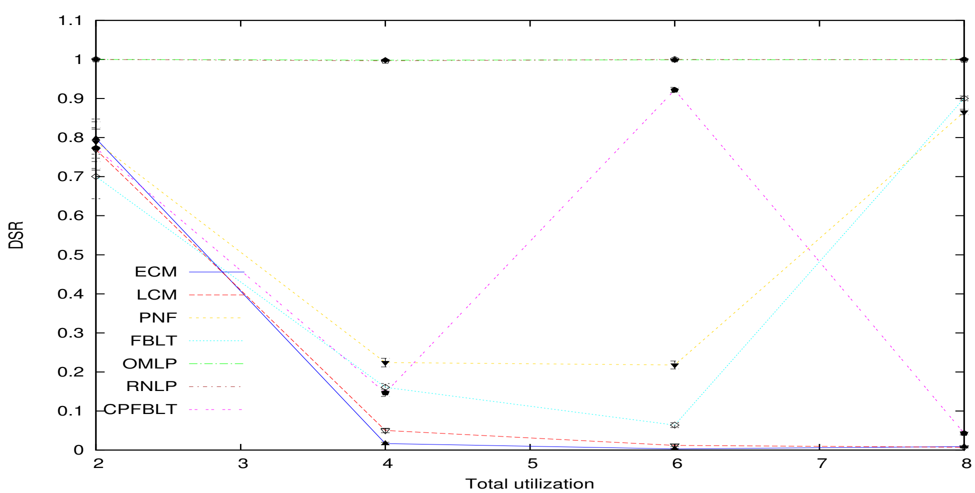


Figure B.148: DSR for Tasksets 148, 418, 688 and 958

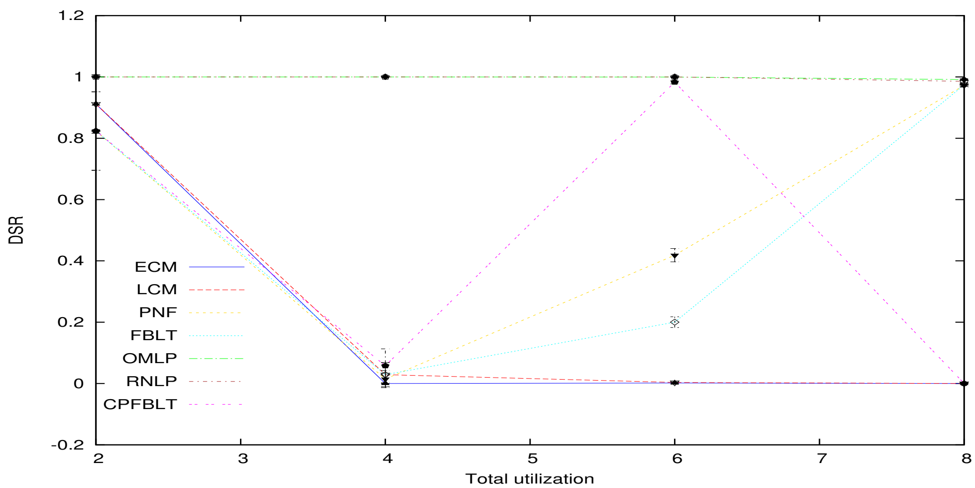


Figure B.149: DSR for Tasksets 149, 419, 689 and 959

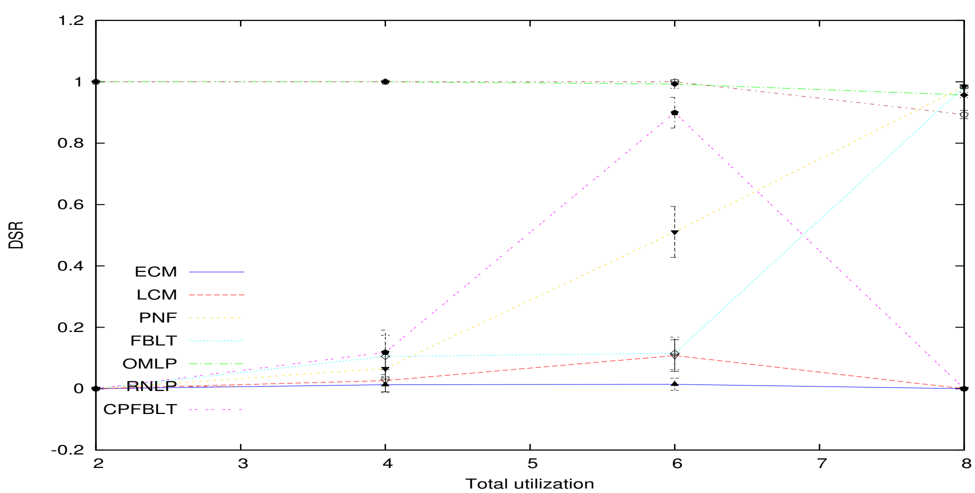


Figure B.150: DSR for Tasksets 150, 420, 690 and 960

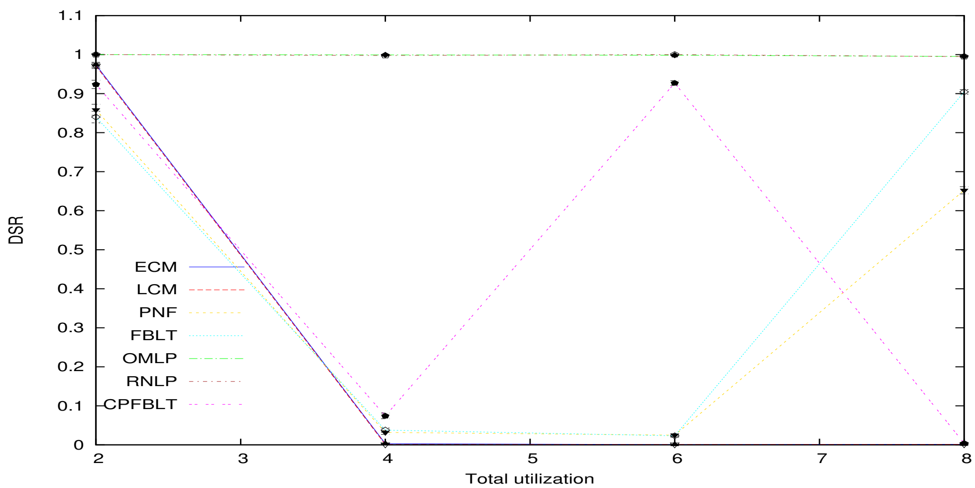


Figure B.151: DSR for Tasksets 151, 421, 691 and 961

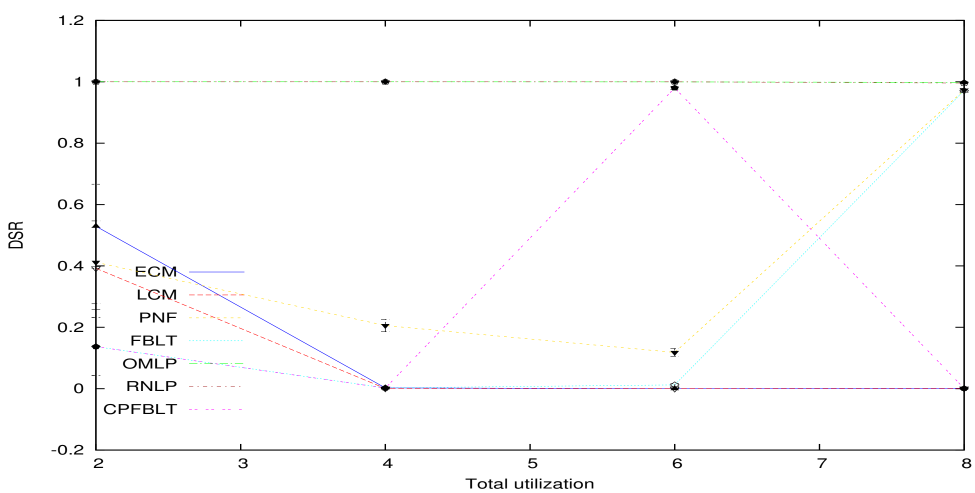


Figure B.152: DSR for Tasksets 152, 422, 692 and 962

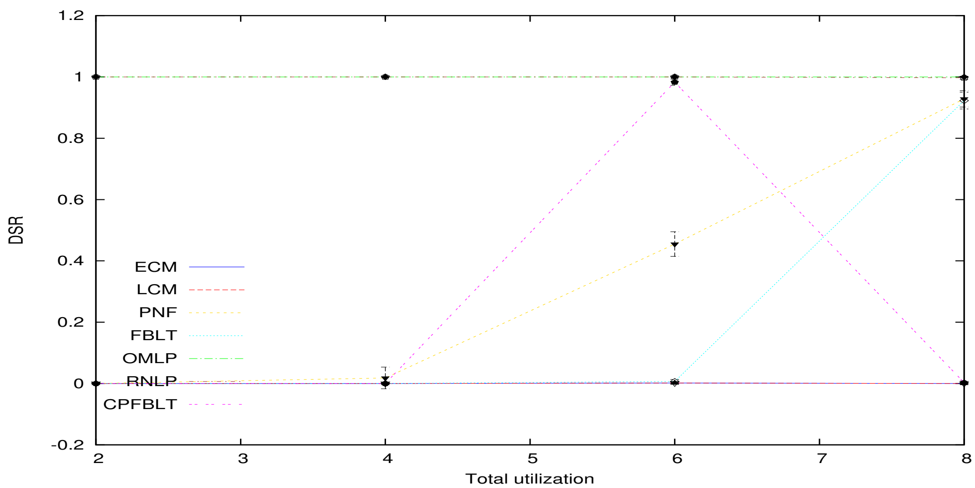


Figure B.153: DSR for Tasksets 153, 423, 693 and 963

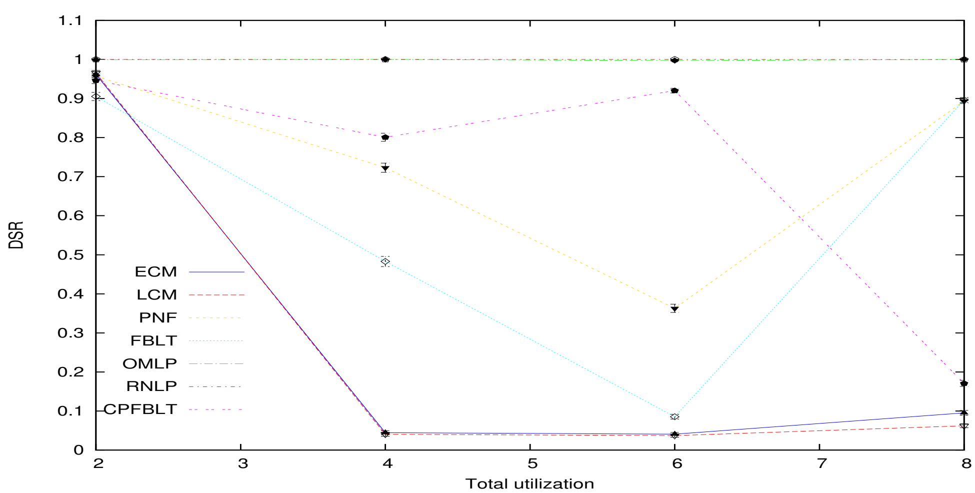


Figure B.154: DSR for Tasksets 154, 424, 694 and 964

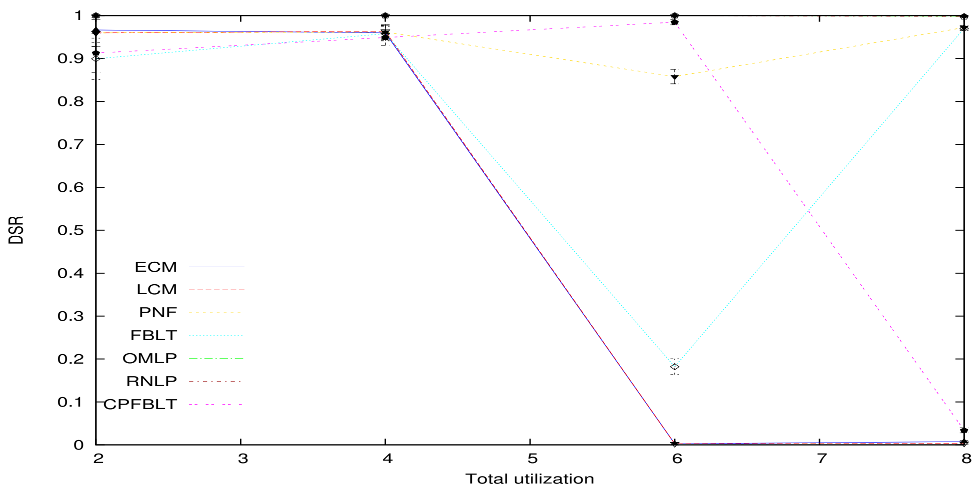


Figure B.155: DSR for Tasksets 155, 425, 695 and 965

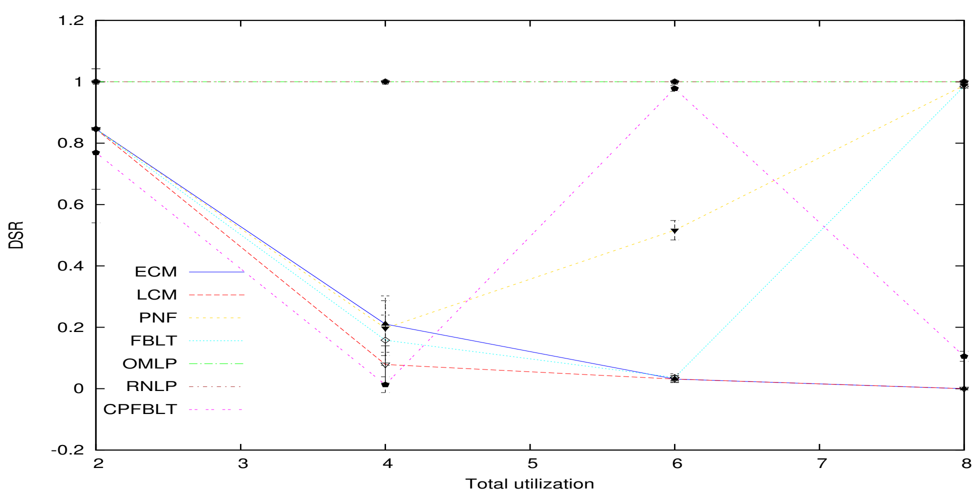


Figure B.156: DSR for Tasksets 156, 426, 696 and 966

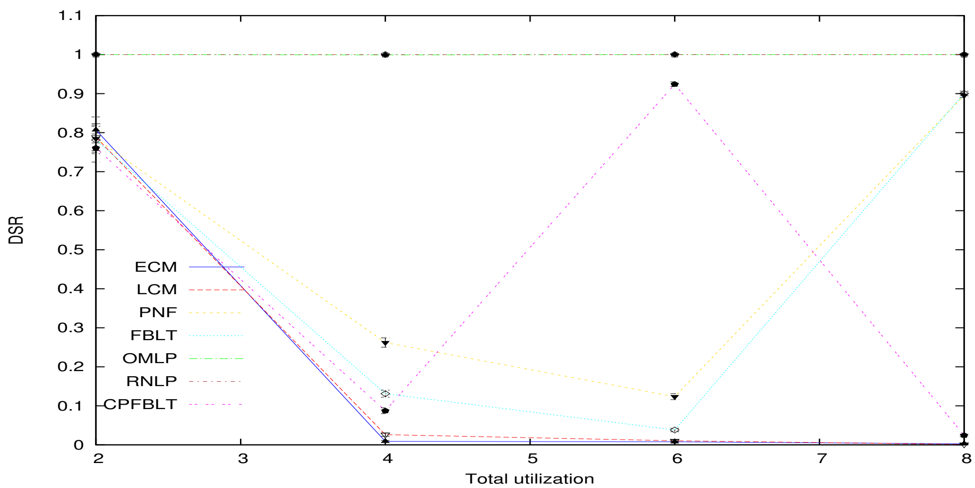


Figure B.157: DSR for Tasksets 157, 427, 697 and 967

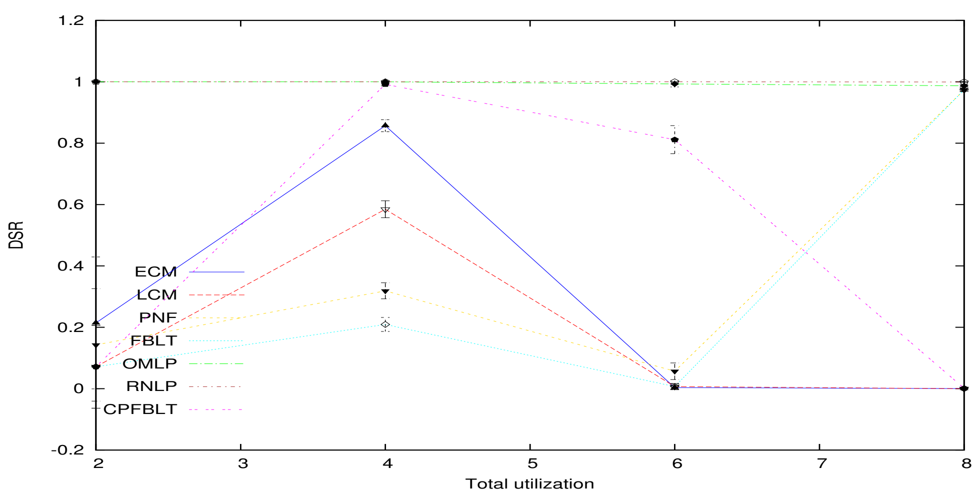


Figure B.158: DSR for Tasksets 158, 428, 698 and 968

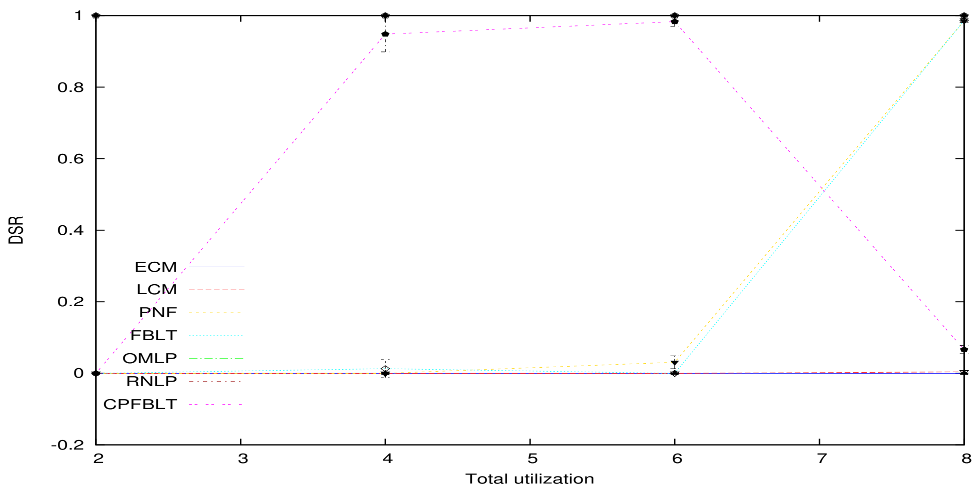


Figure B.159: DSR for Tasksets 159, 429, 699 and 969

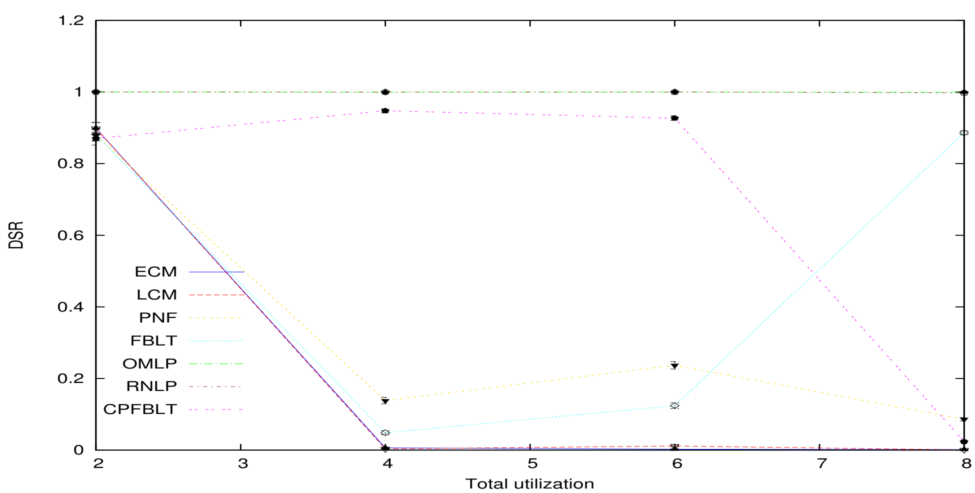


Figure B.160: DSR for Tasksets 160, 430, 700 and 970

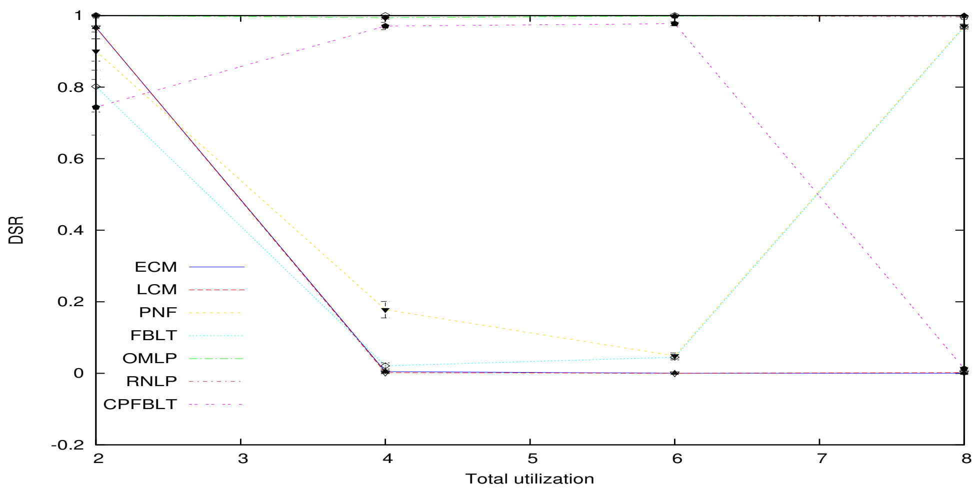


Figure B.161: DSR for Tasksets 161, 431, 701 and 971

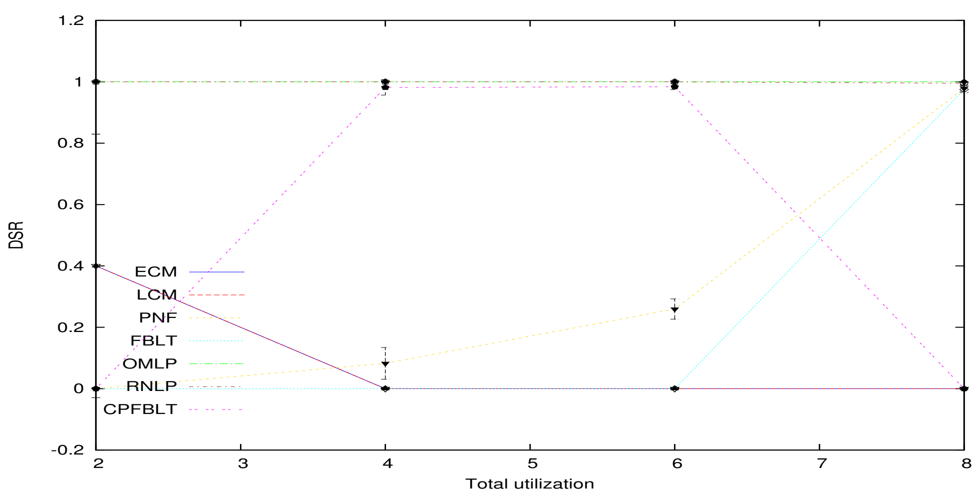


Figure B.162: DSR for Tasksets 162, 432, 702 and 972

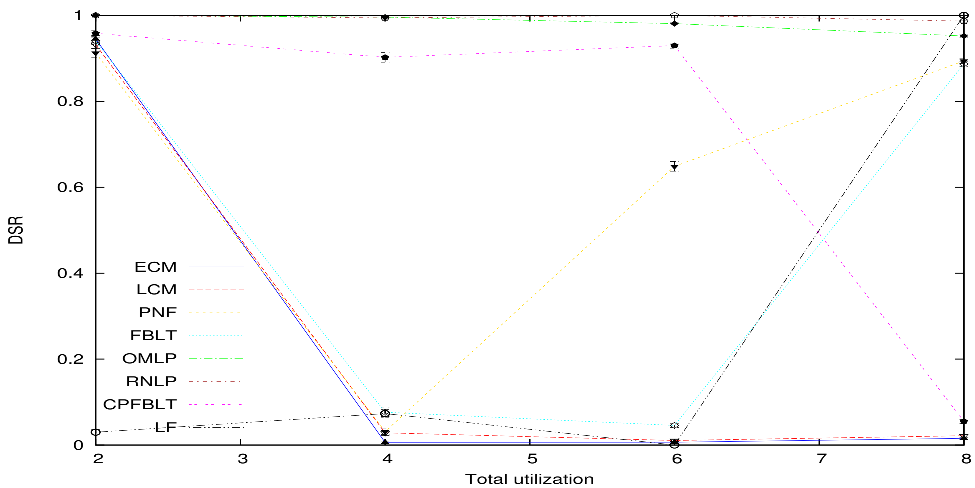


Figure B.163: DSR for Tasksets 163, 433, 703 and 973

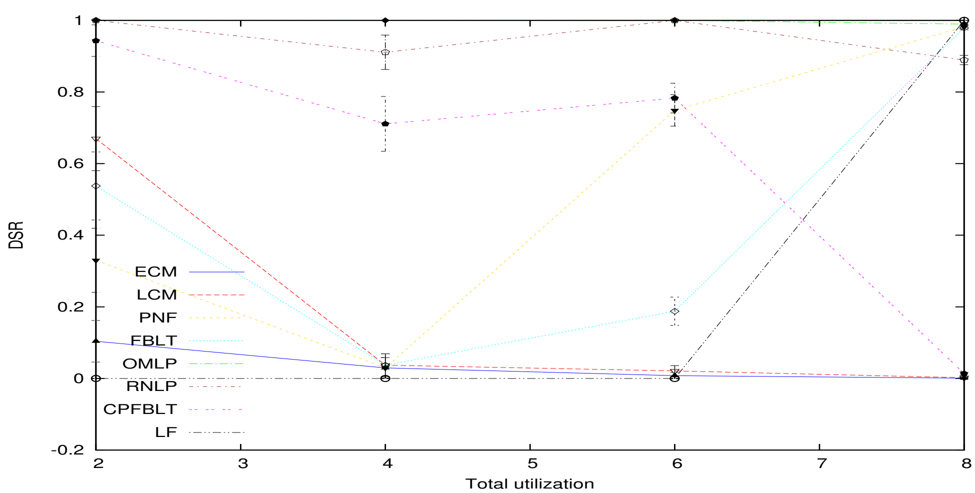


Figure B.164: DSR for Tasksets 164, 434, 704 and 974

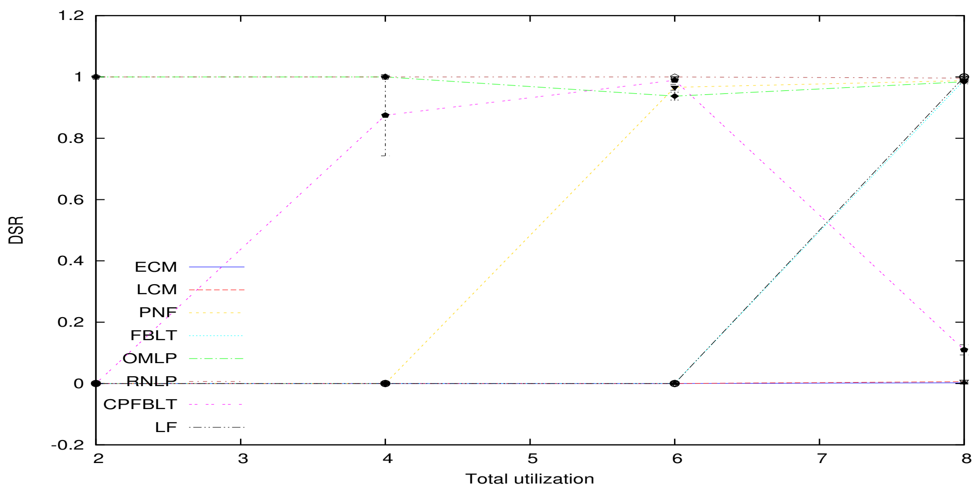


Figure B.165: DSR for Tasksets 165, 435, 705 and 975

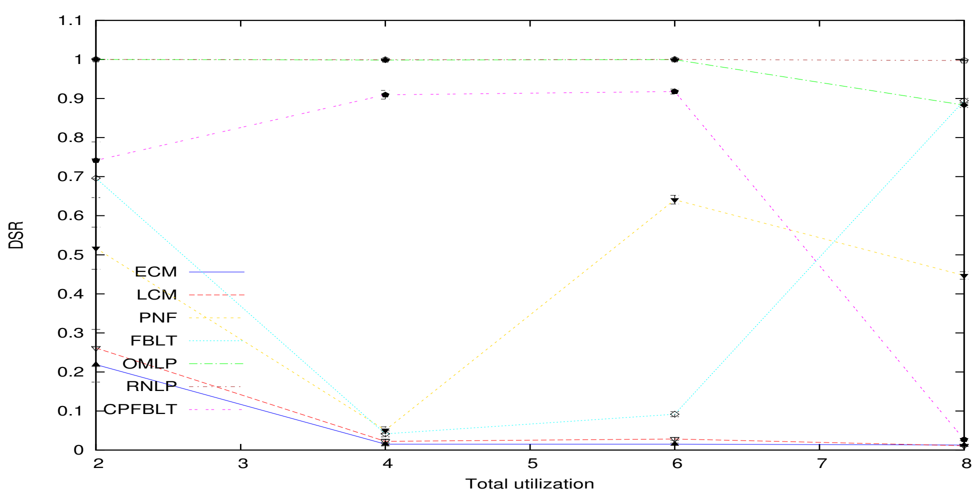


Figure B.166: DSR for Tasksets 166, 436, 706 and 976

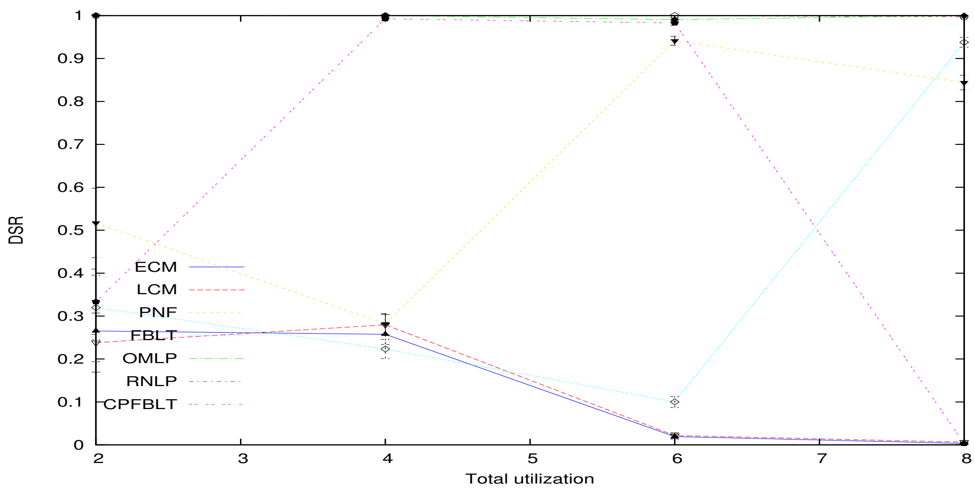


Figure B.167: DSR for Tasksets 167, 437, 707 and 977

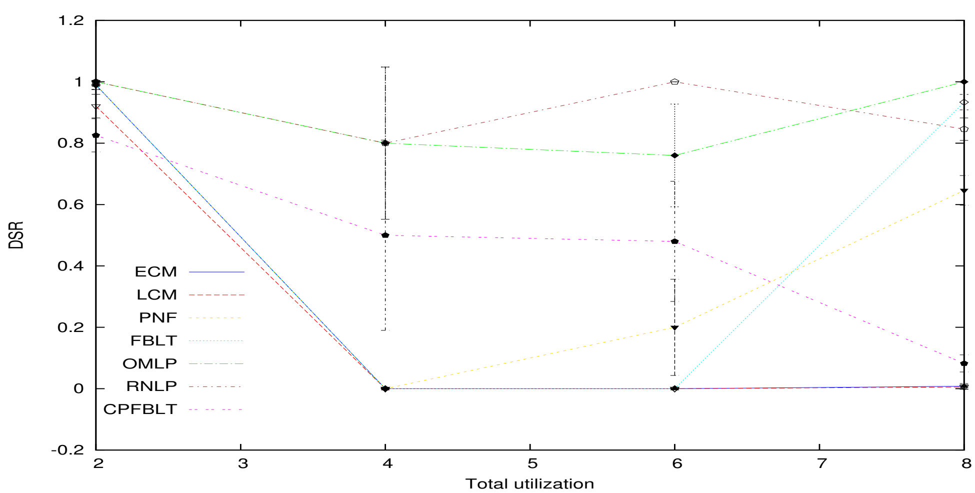


Figure B.168: DSR for Tasksets 168, 438, 708 and 978

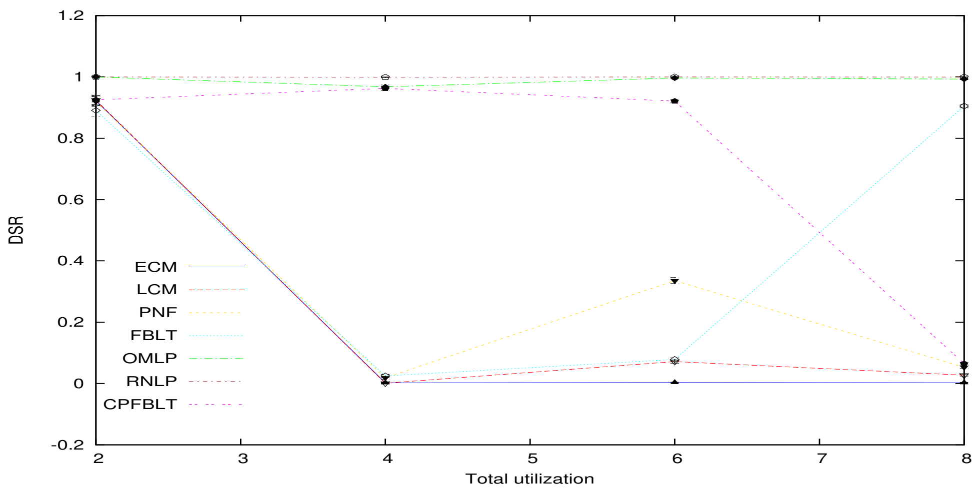


Figure B.169: DSR for Tasksets 169, 439, 709 and 979

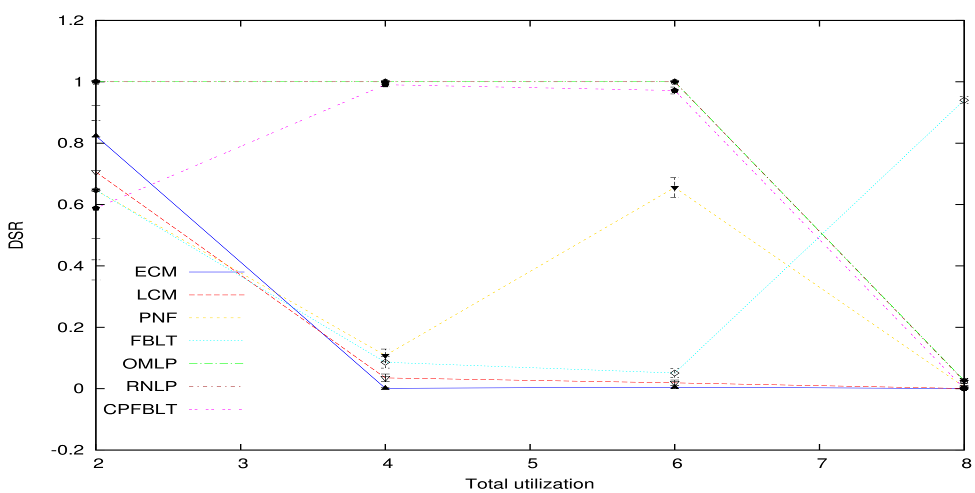


Figure B.170: DSR for Tasksets 170, 440, 710 and 980

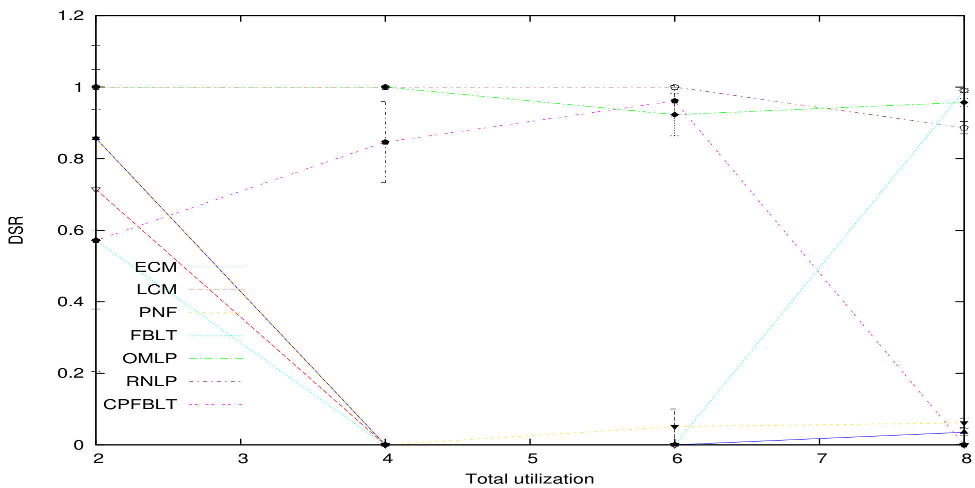


Figure B.171: DSR for Tasksets 171, 441, 711 and 981

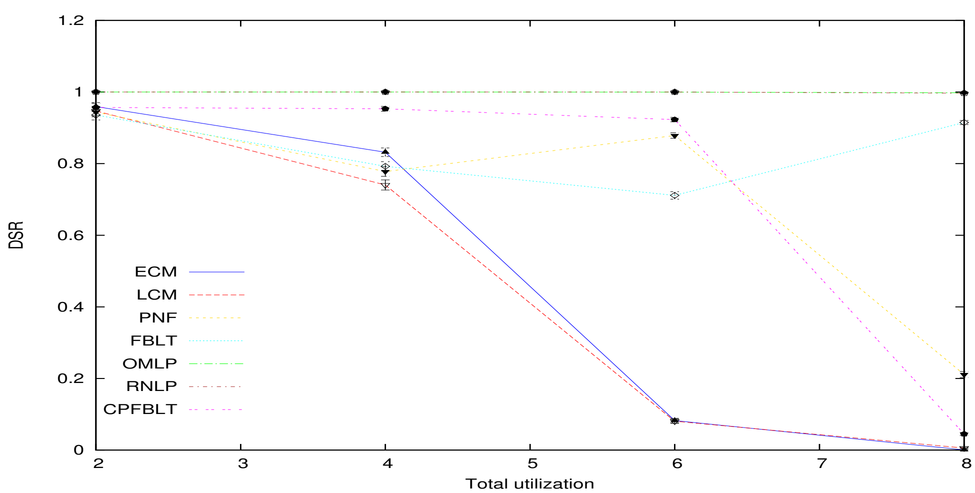


Figure B.172: DSR for Tasksets 172, 442, 712 and 982

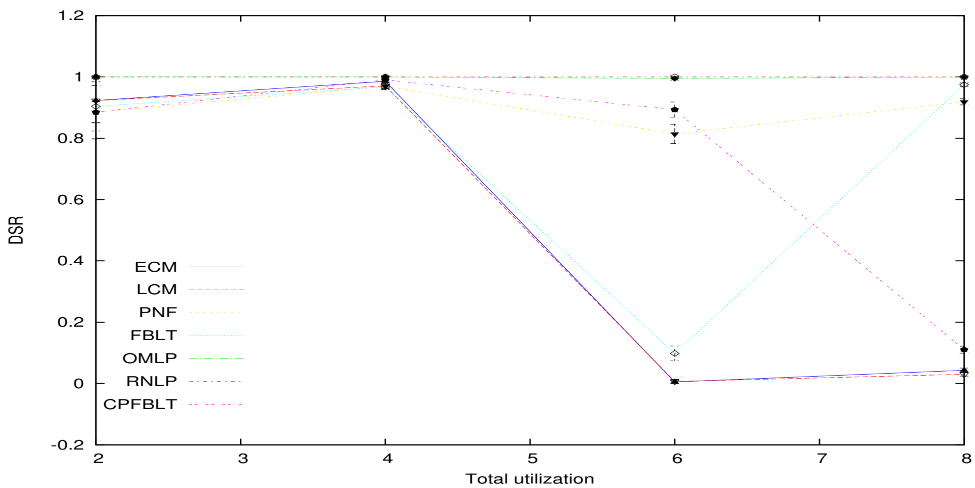


Figure B.173: DSR for Tasksets 173, 443, 713 and 983

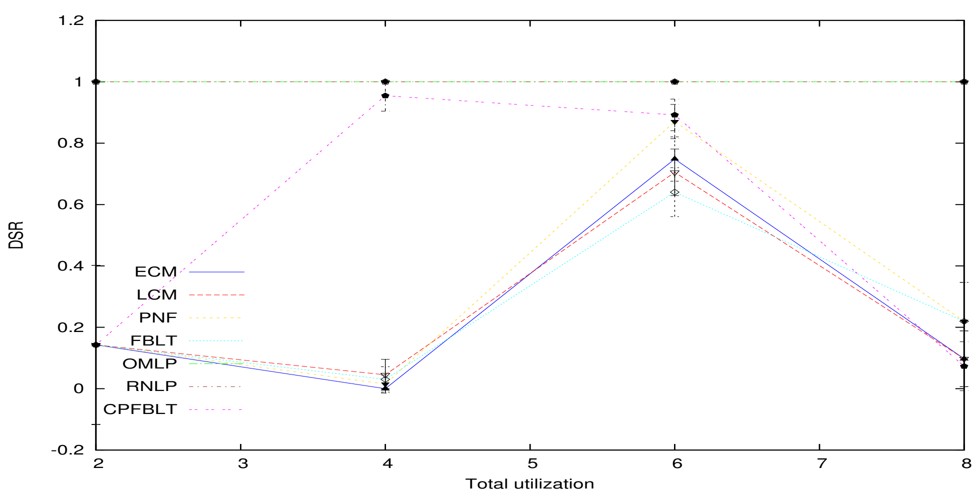


Figure B.174: DSR for Tasksets 174, 444, 714 and 984

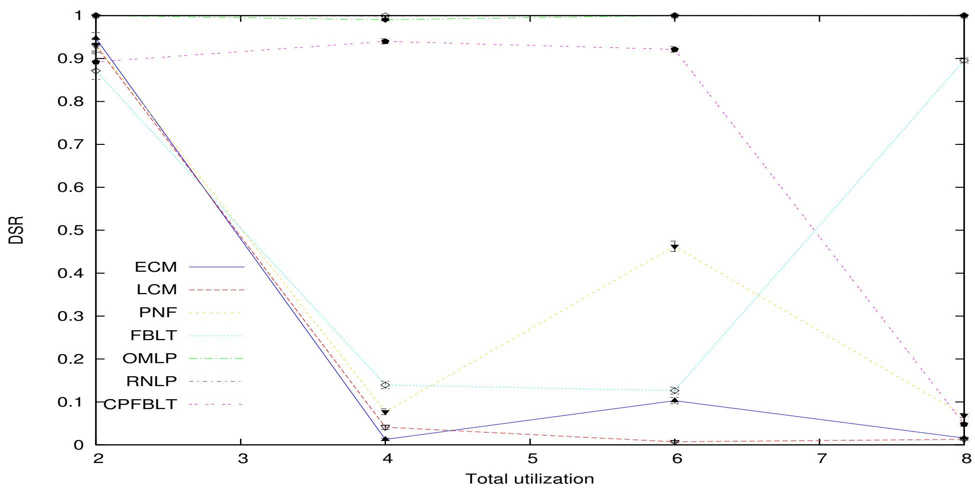


Figure B.175: DSR for Tasksets 175, 445, 715 and 985

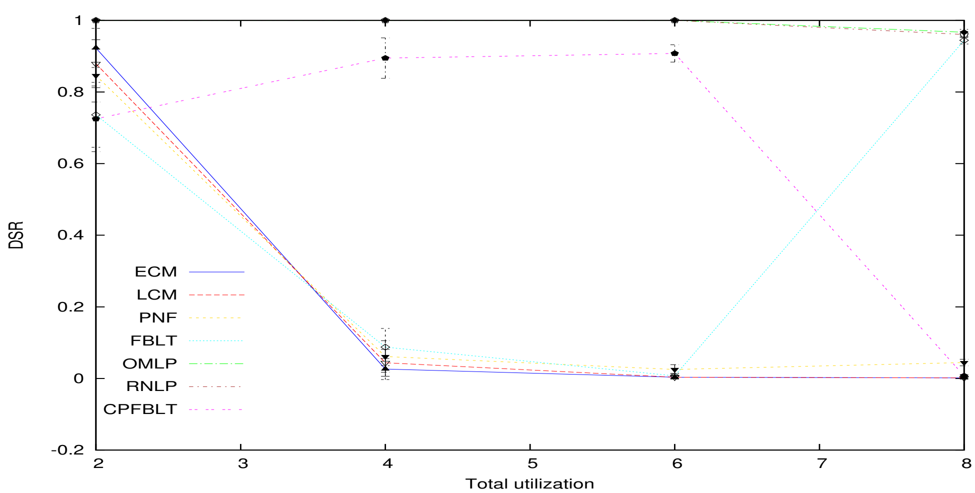


Figure B.176: DSR for Tasksets 176, 446, 716 and 986

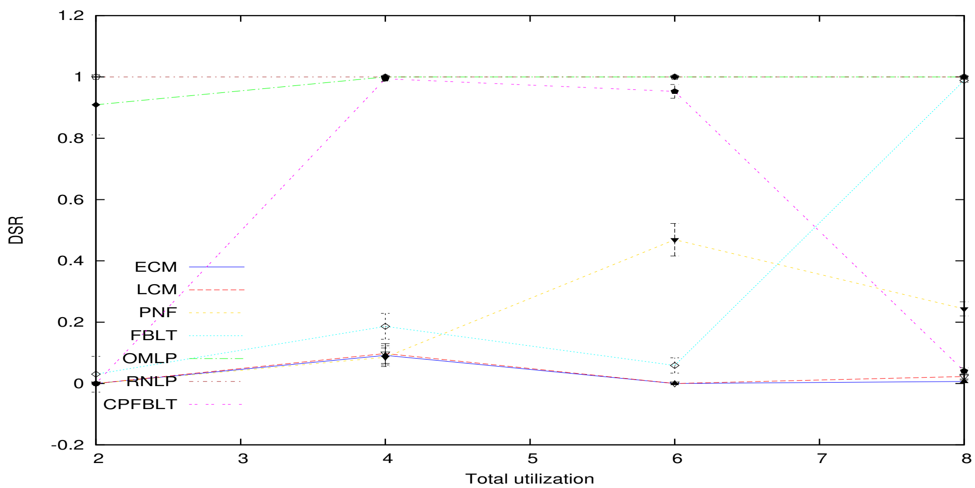


Figure B.177: DSR for Tasksets 177, 447, 717 and 987

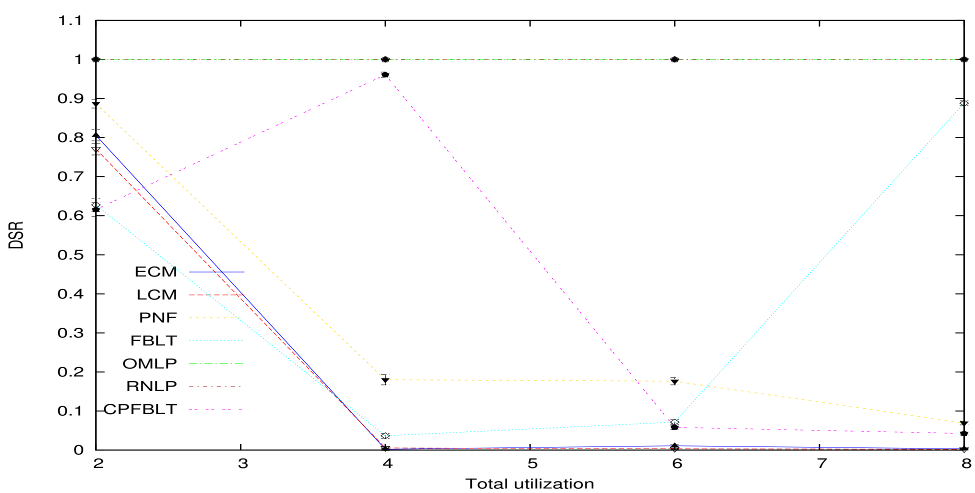


Figure B.178: DSR for Tasksets 178, 448, 718 and 988

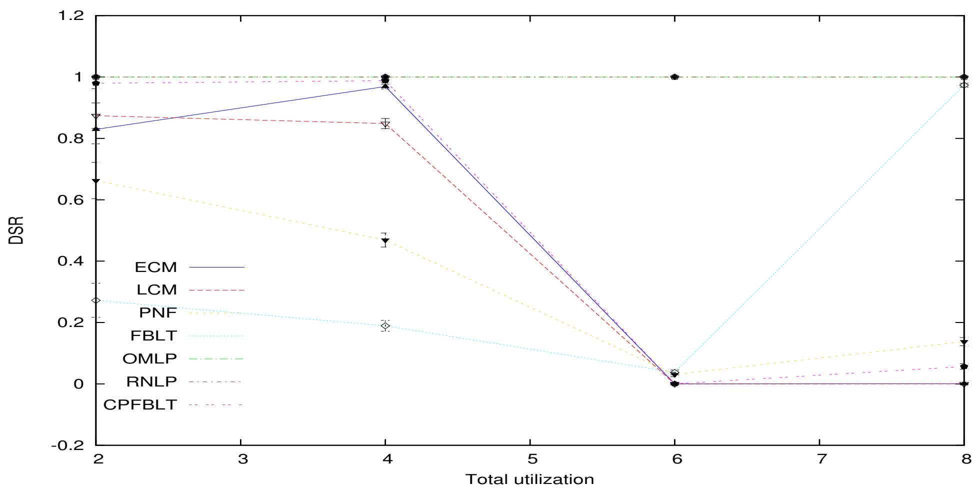


Figure B.179: DSR for Tasksets 179, 449, 719 and 989

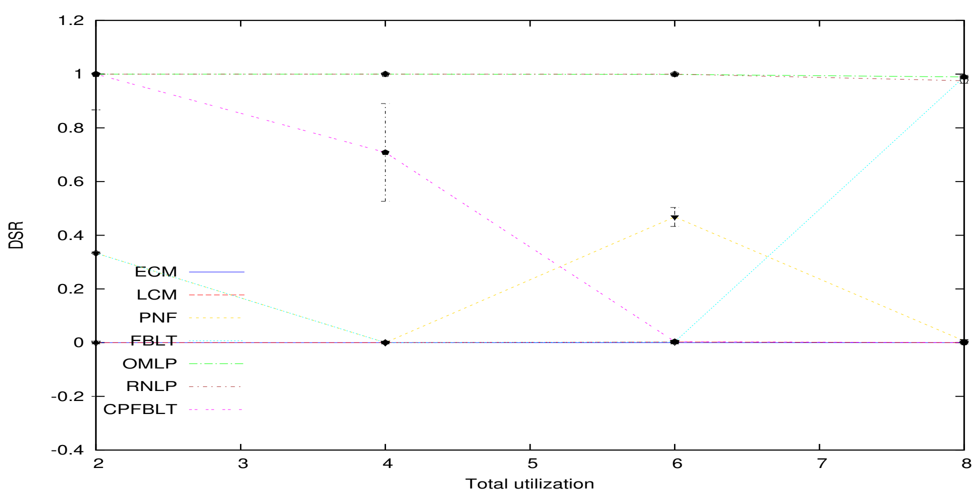


Figure B.180: DSR for Tasksets 180, 450, 720 and 990

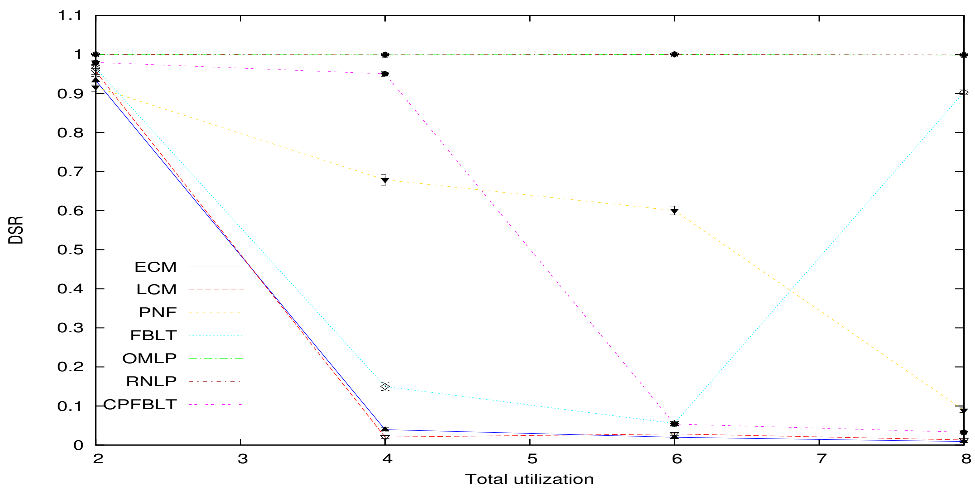


Figure B.181: DSR for Tasksets 181, 451, 721 and 991

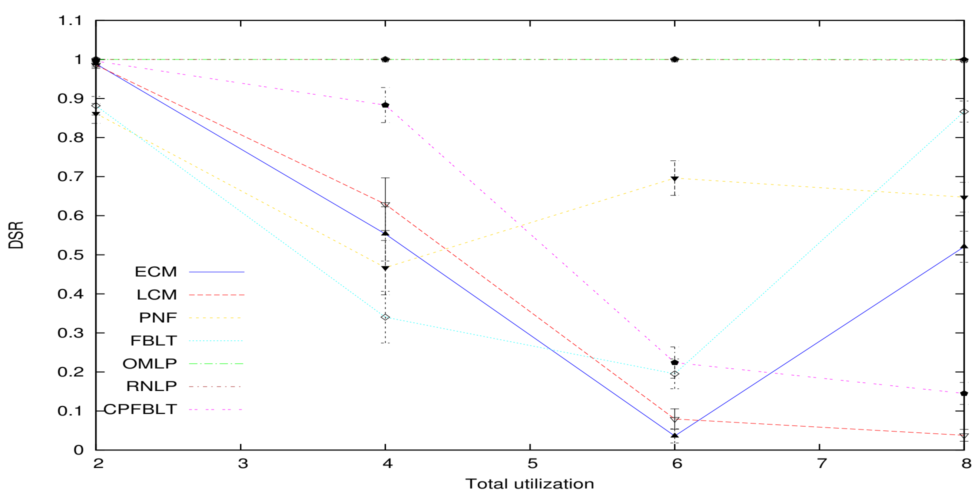


Figure B.182: DSR for Tasksets 182, 452, 722 and 992

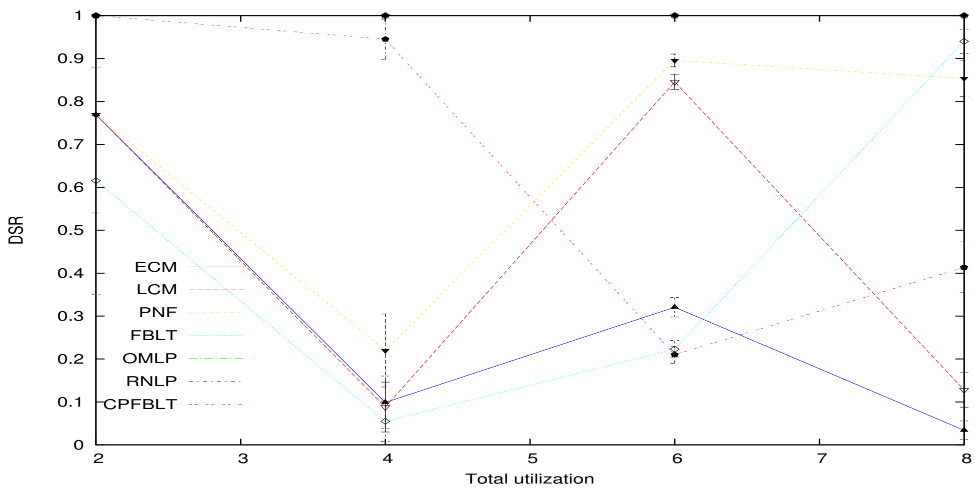


Figure B.183: DSR for Tasksets 183, 453, 723 and 993

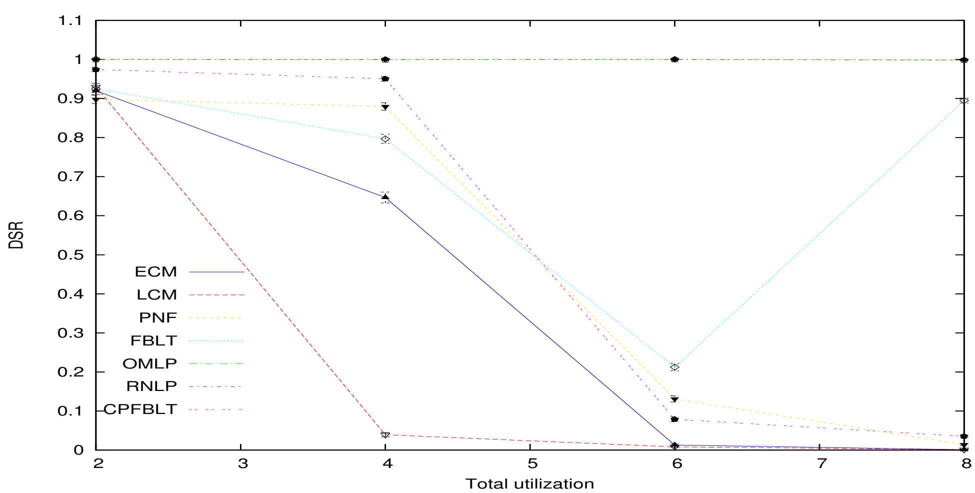


Figure B.184: DSR for Tasksets 184, 454, 724 and 994

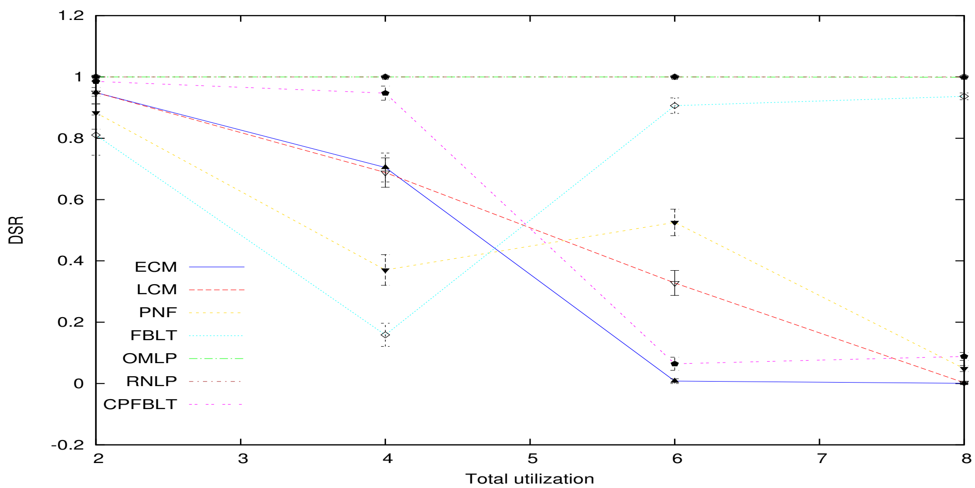


Figure B.185: DSR for Tasksets 185, 455, 725 and 995

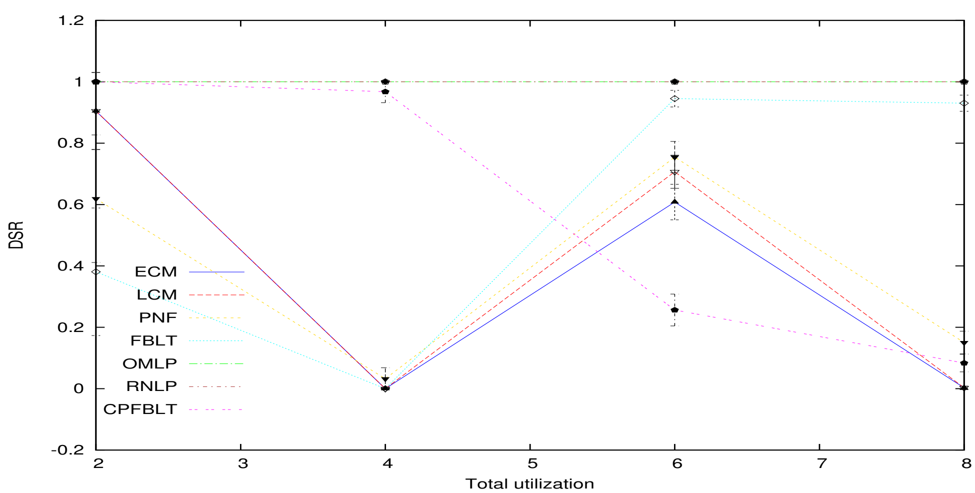


Figure B.186: DSR for Tasksets 186, 456, 726 and 996

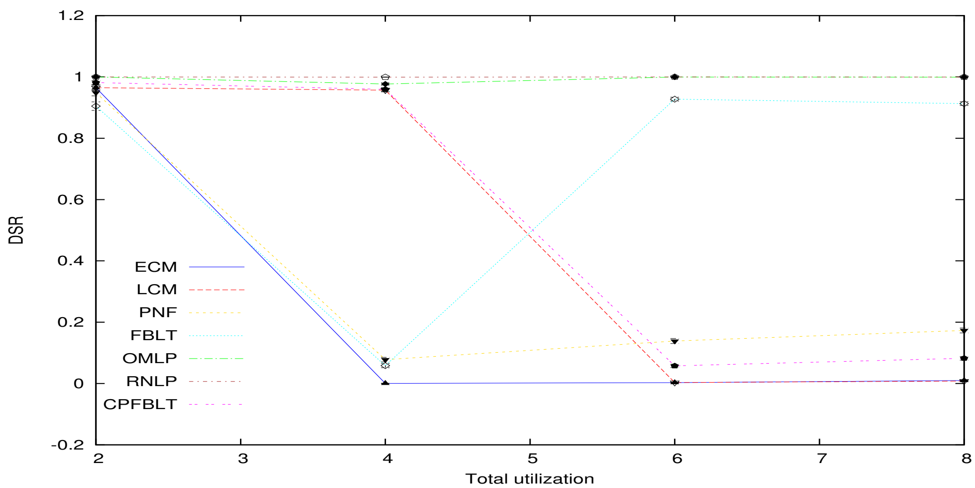


Figure B.187: DSR for Tasksets 187, 457, 727 and 997

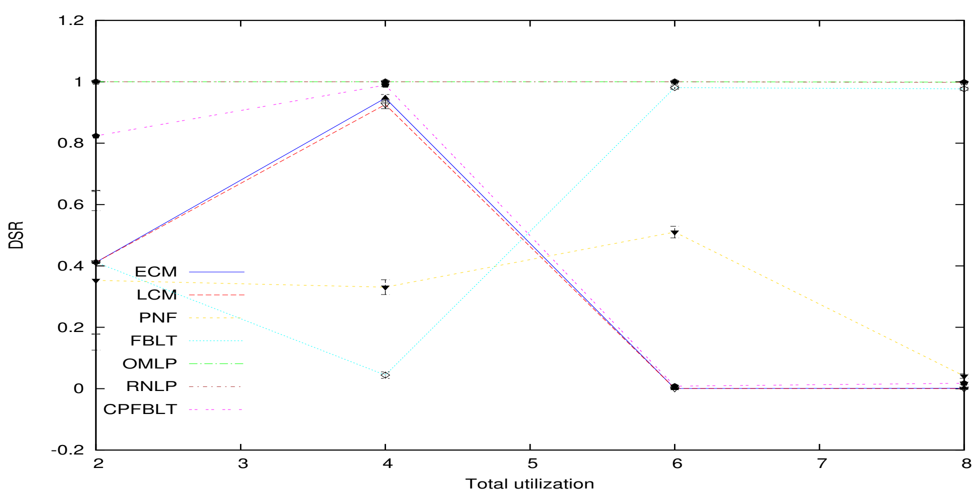


Figure B.188: DSR for Tasksets 188, 458, 728 and 998

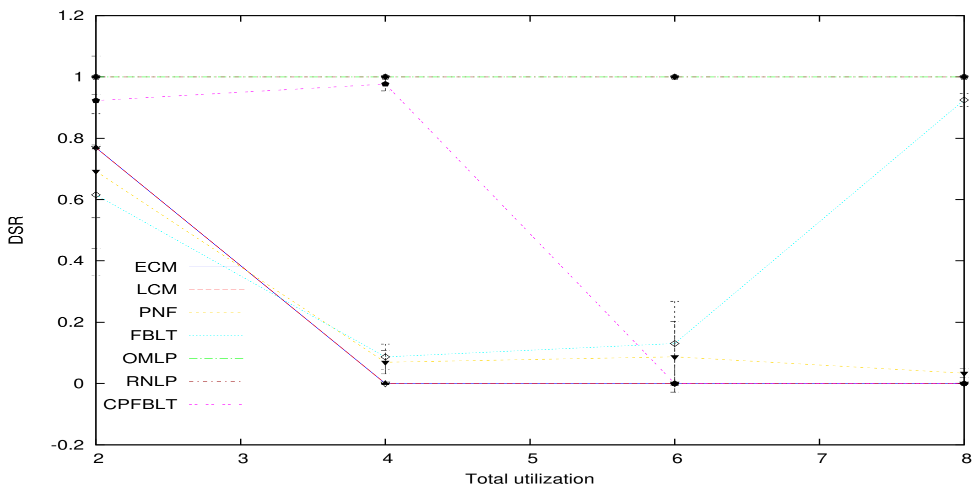


Figure B.189: DSR for Tasksets 189, 459, 729 and 999

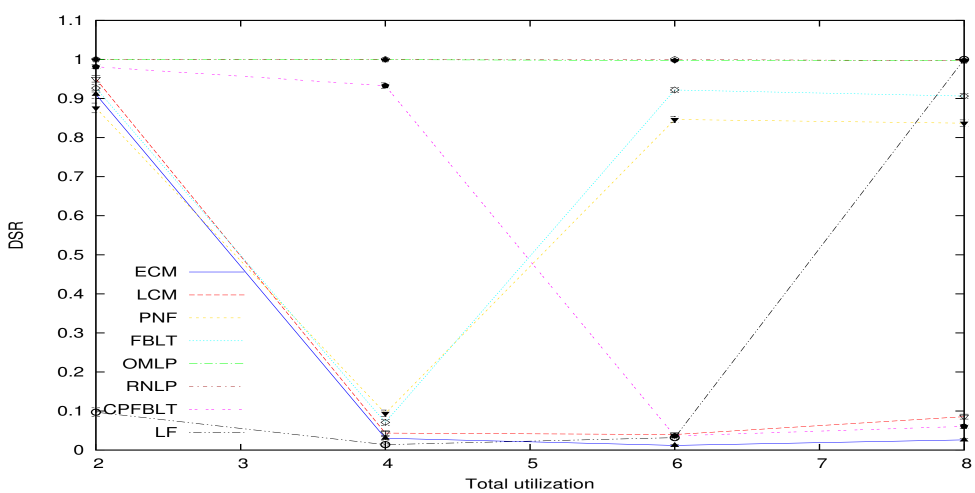


Figure B.190: DSR for Tasksets 190, 460, 730 and 1000

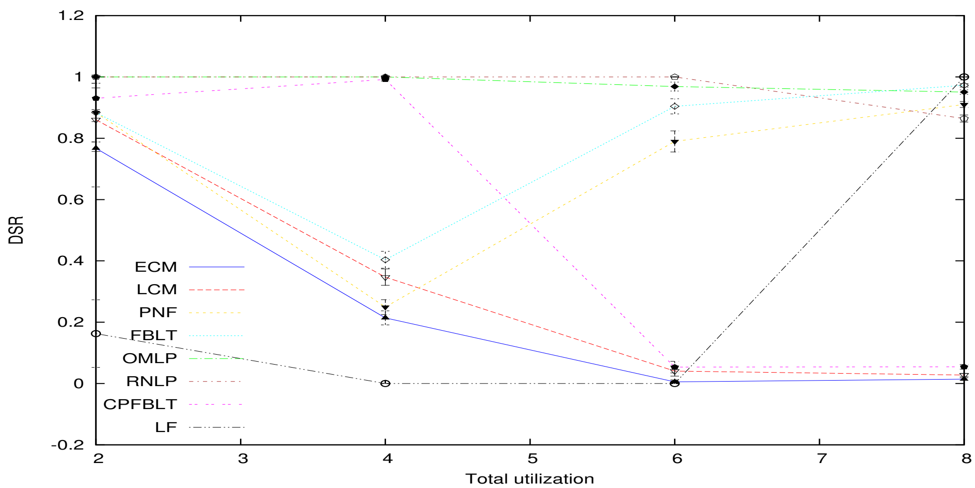


Figure B.191: DSR for Tasksets 191, 461, 731 and 1001

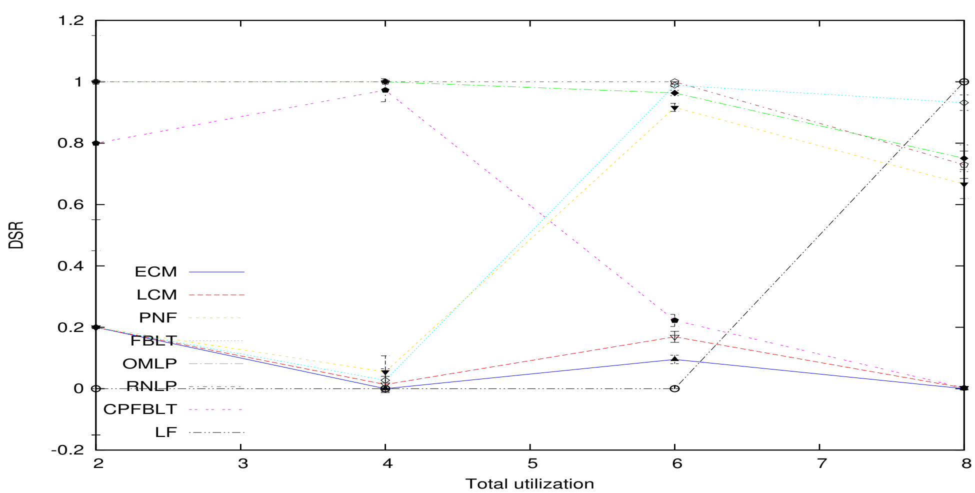


Figure B.192: DSR for Tasksets 192, 462, 732 and 1002

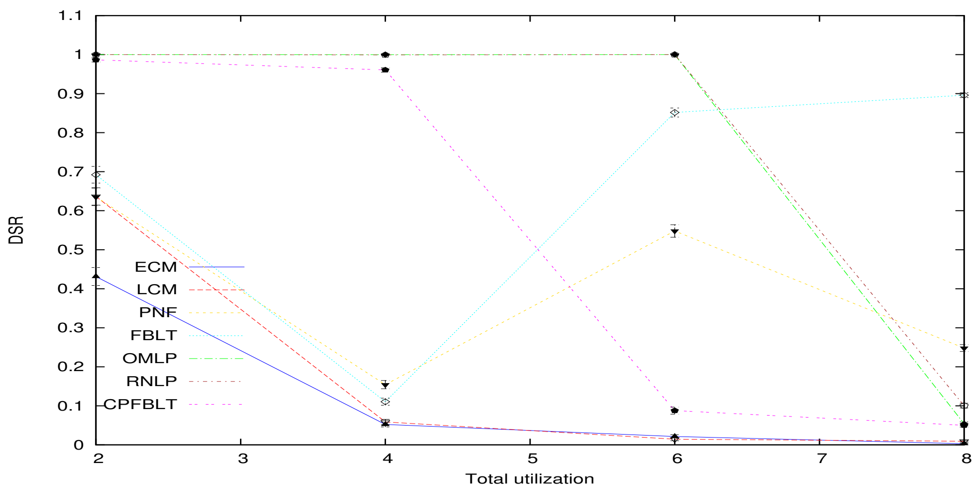


Figure B.193: DSR for Tasksets 193, 463, 733 and 1003

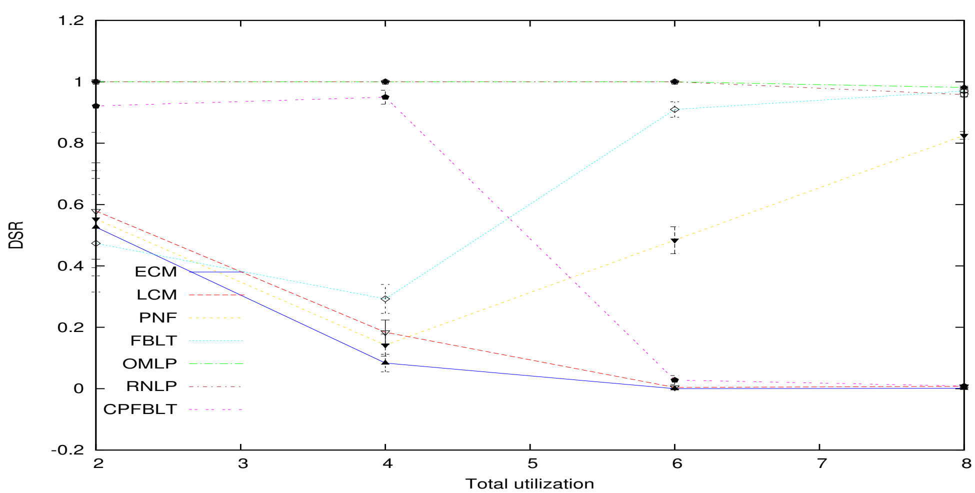


Figure B.194: DSR for Tasksets 194, 464, 734 and 1004

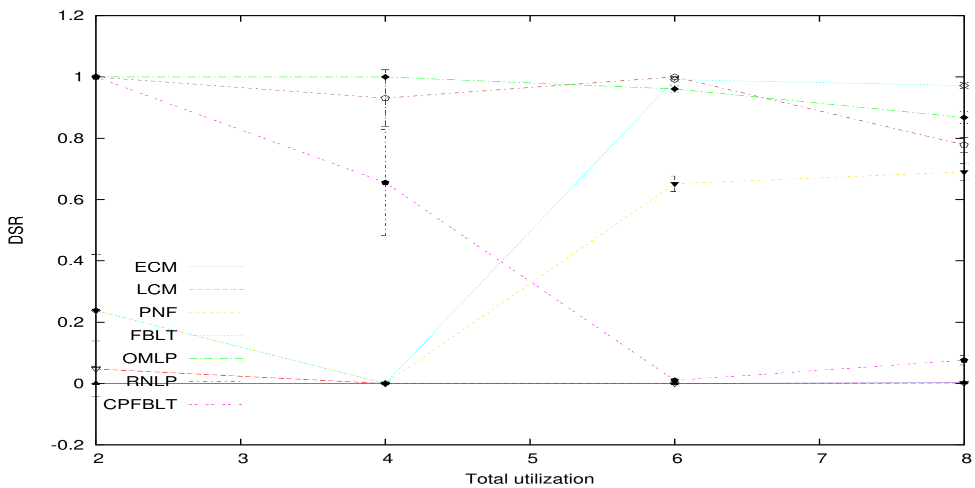


Figure B.195: DSR for Tasksets 195, 465, 735 and 1005

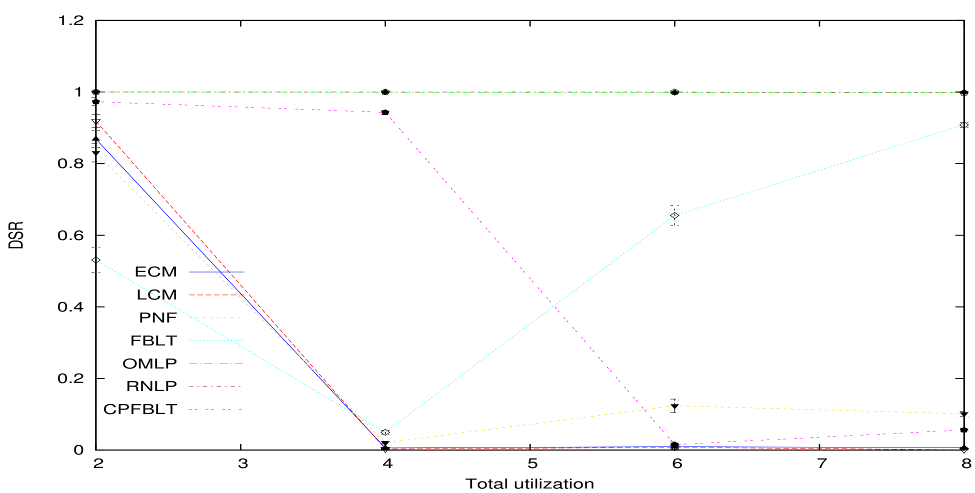


Figure B.196: DSR for Tasksets 196, 466, 736 and 1006

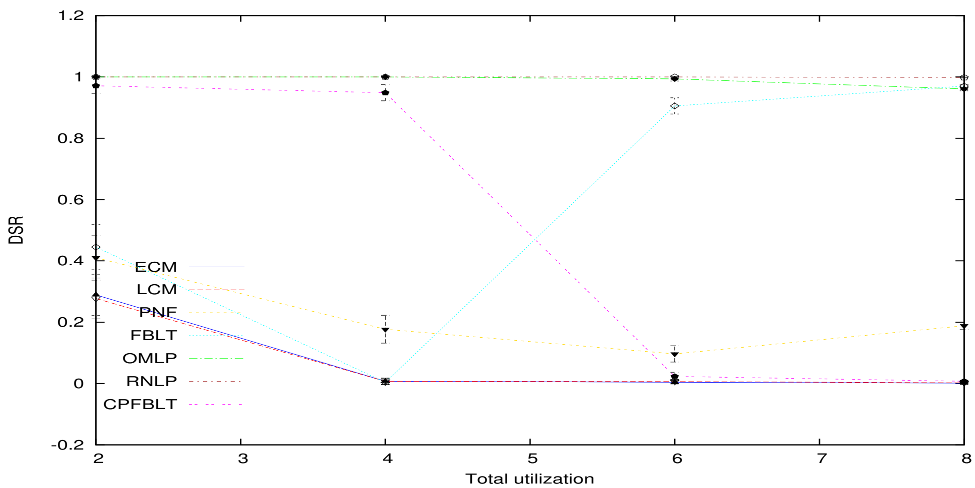


Figure B.197: DSR for Tasksets 197, 467, 737 and 1007

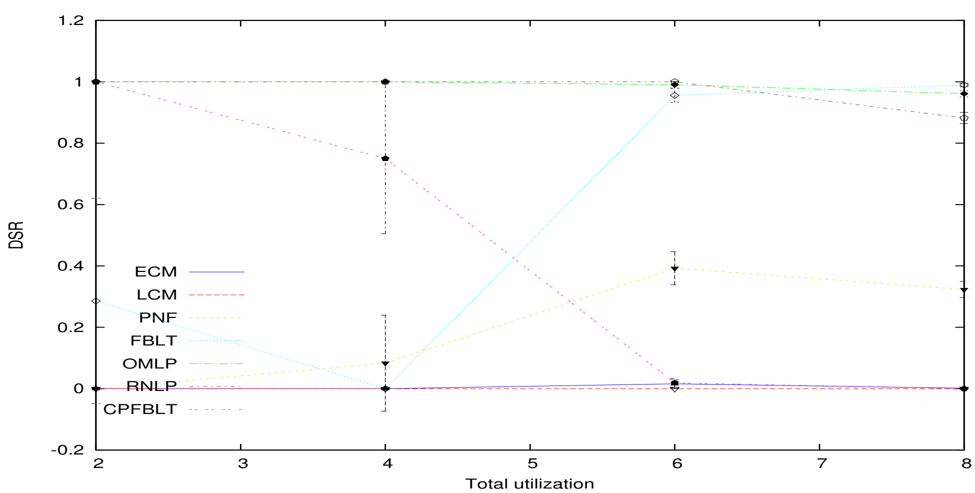


Figure B.198: DSR for Tasksets 198, 468, 738 and 1008

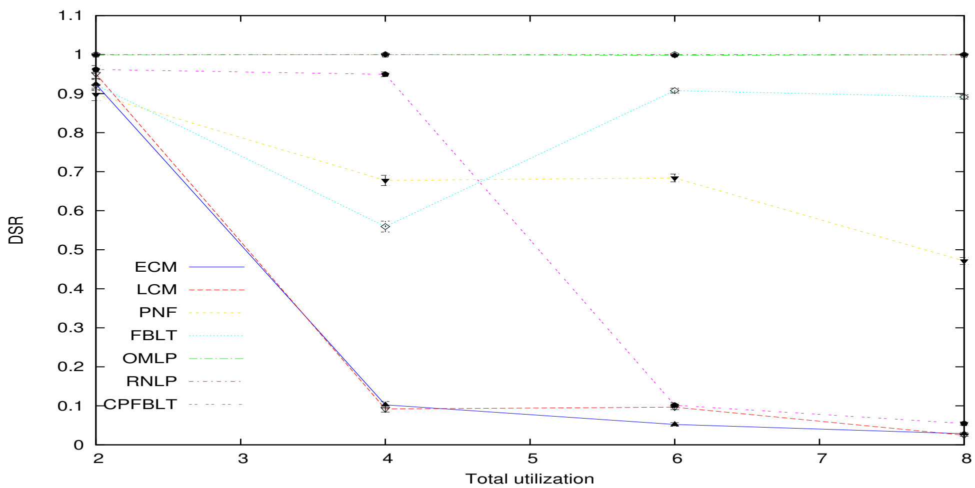


Figure B.199: DSR for Tasksets 199, 469, 739 and 1009

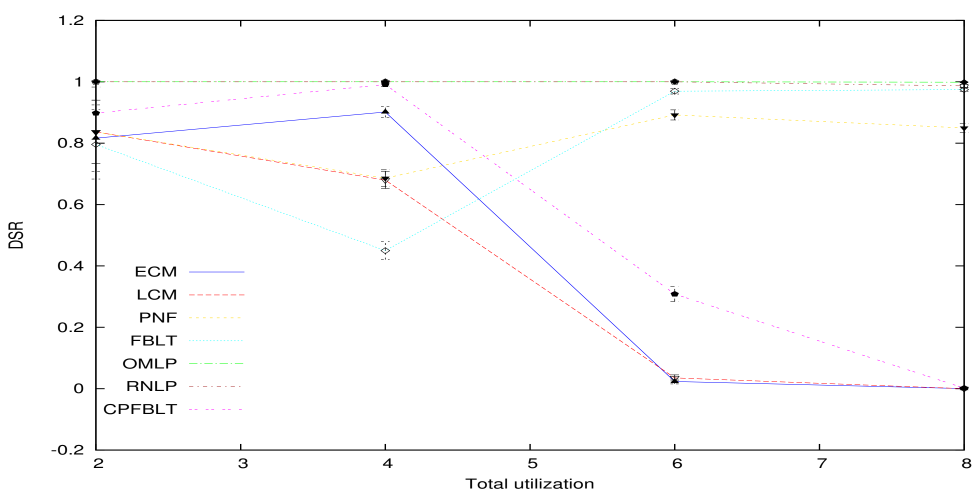


Figure B.200: DSR for Tasksets 200, 470, 740 and 1010

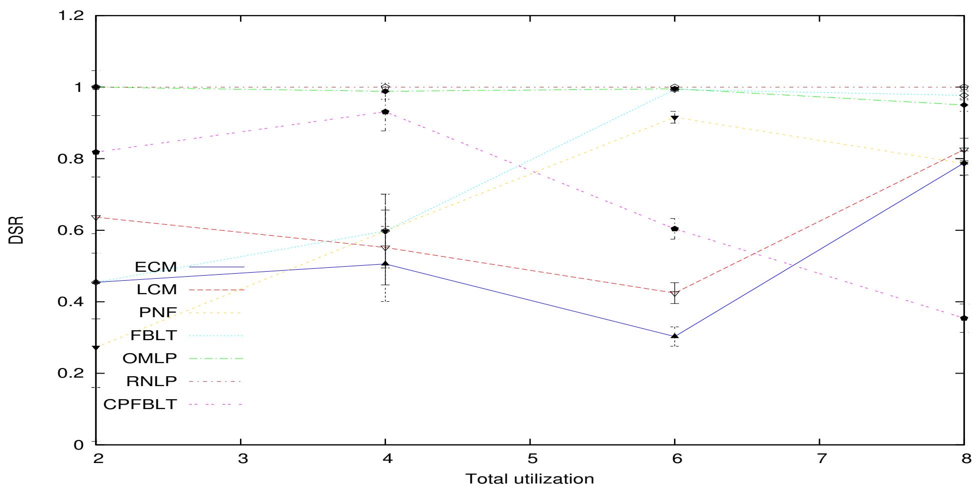


Figure B.201: DSR for Tasksets 201, 471, 741 and 1011

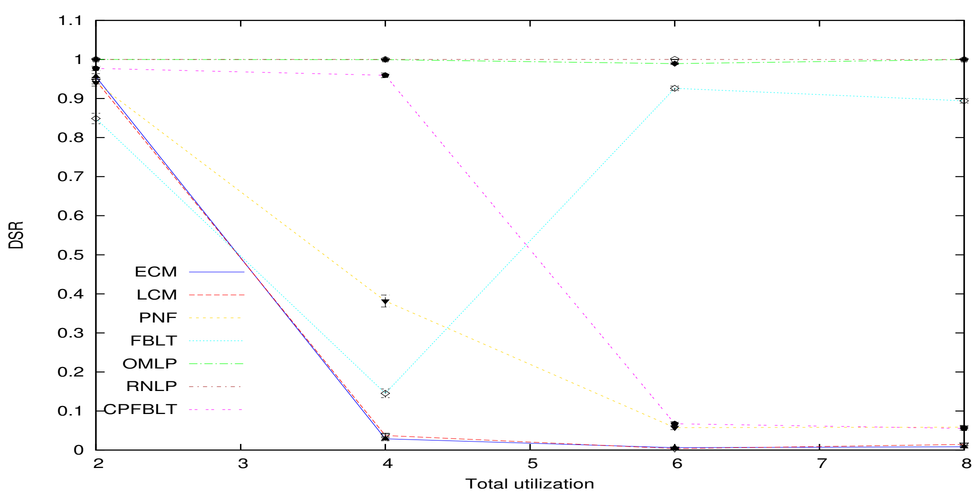


Figure B.202: DSR for Tasksets 202, 472, 742 and 1012

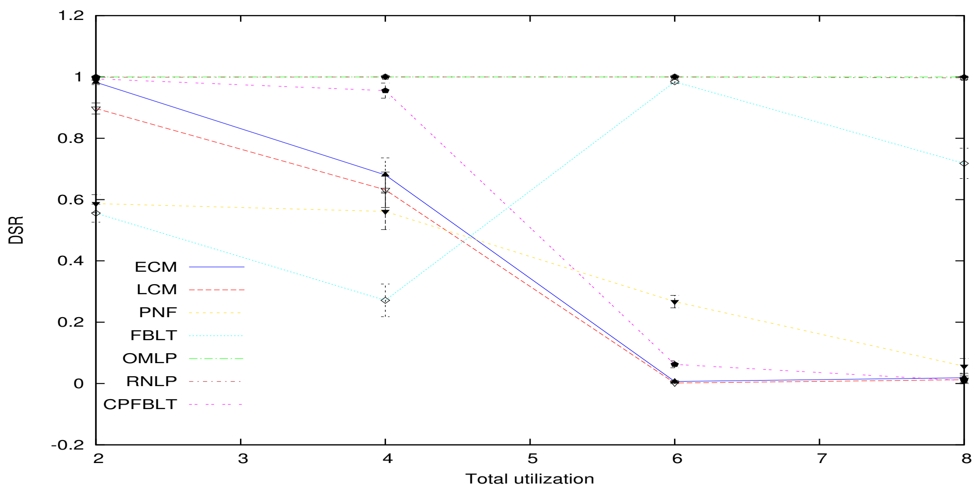


Figure B.203: DSR for Tasksets 203, 473, 743 and 1013

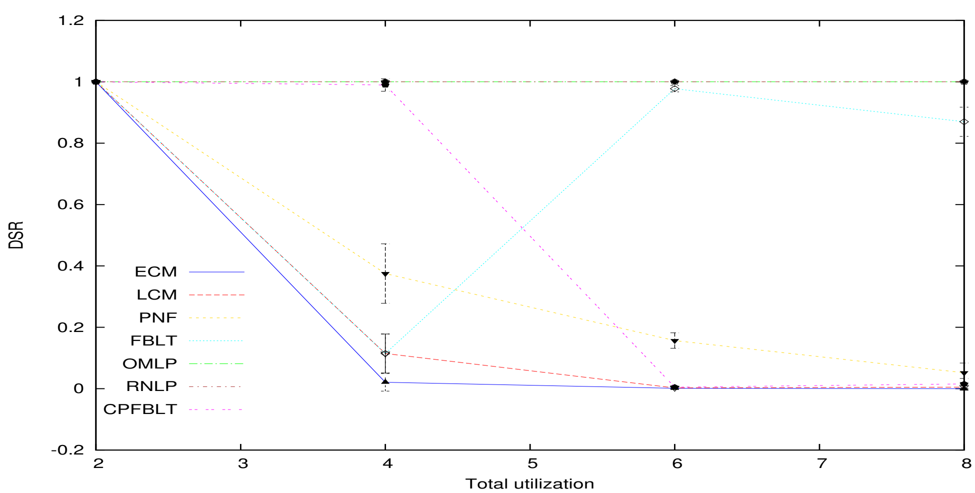


Figure B.204: DSR for Tasksets 204, 474, 744 and 1014

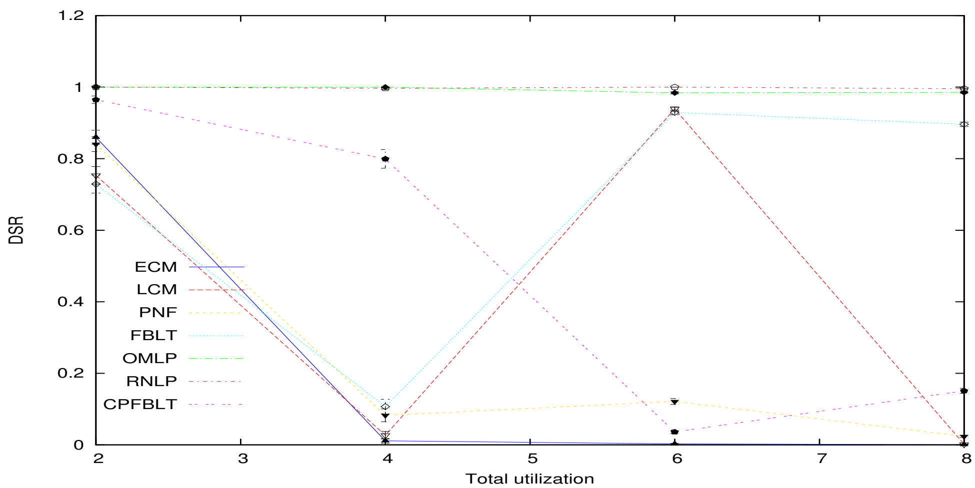


Figure B.205: DSR for Tasksets 205, 475, 745 and 1015

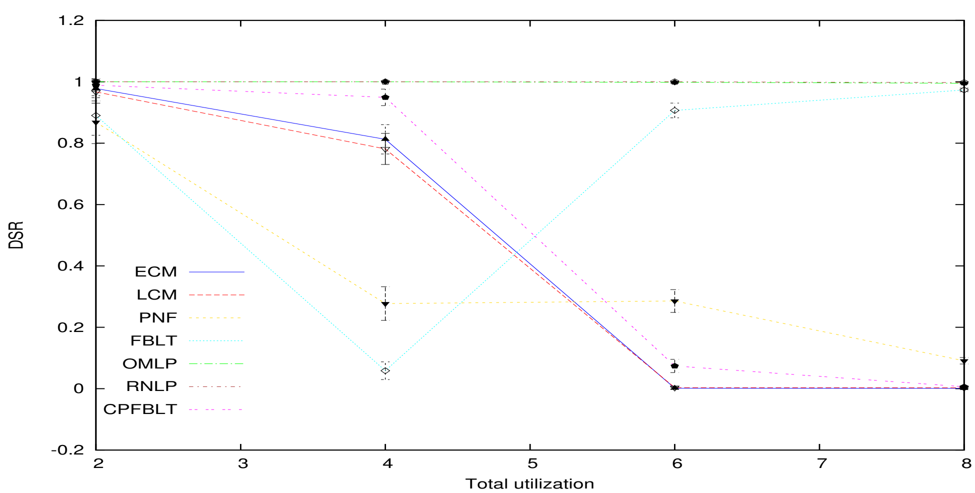


Figure B.206: DSR for Tasksets 206, 476, 746 and 1016

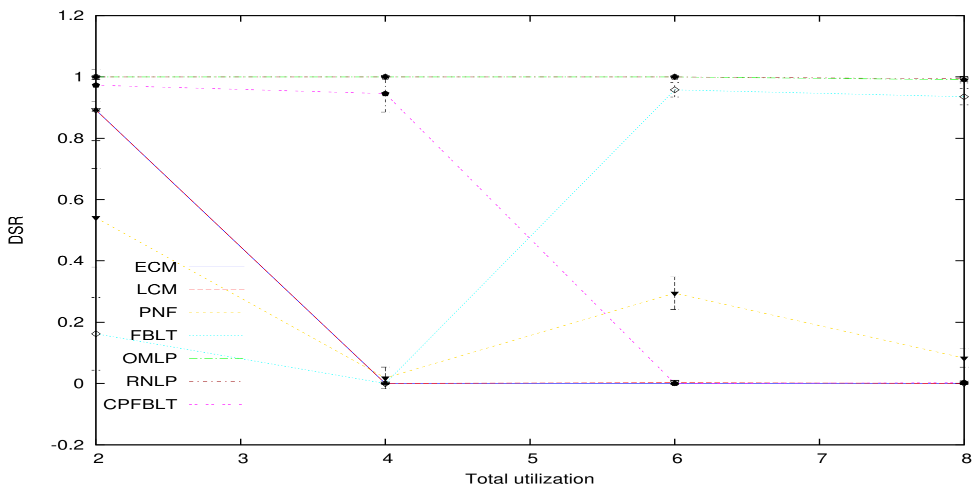


Figure B.207: DSR for Tasksets 207, 477, 747 and 1017

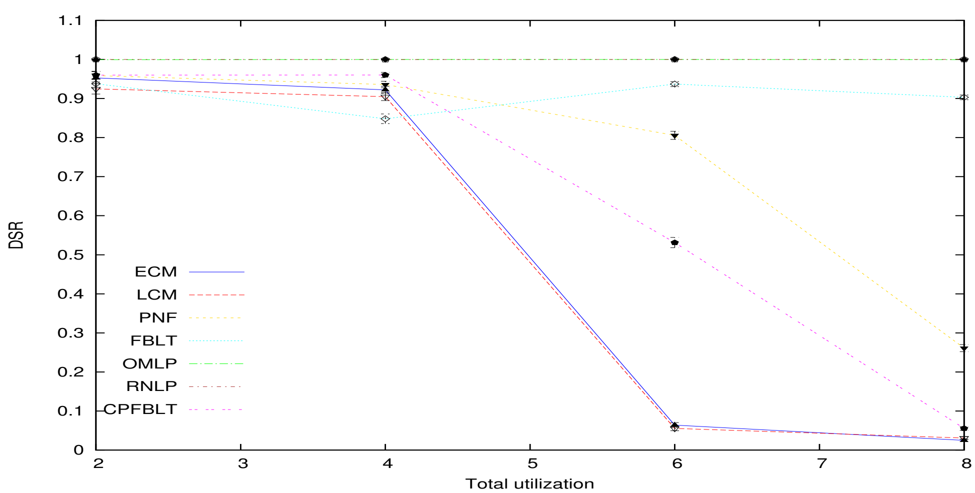


Figure B.208: DSR for Tasksets 208, 478, 748 and 1018

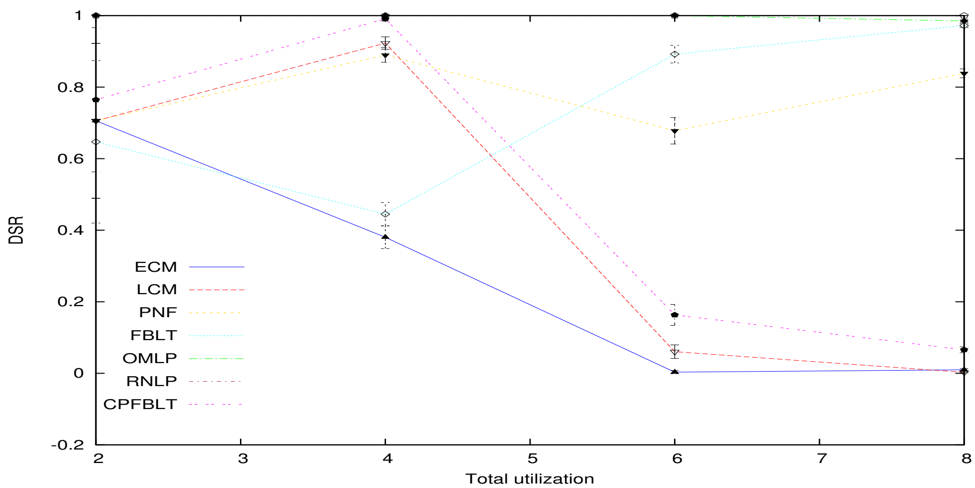


Figure B.209: DSR for Tasksets 209, 479, 749 and 1019

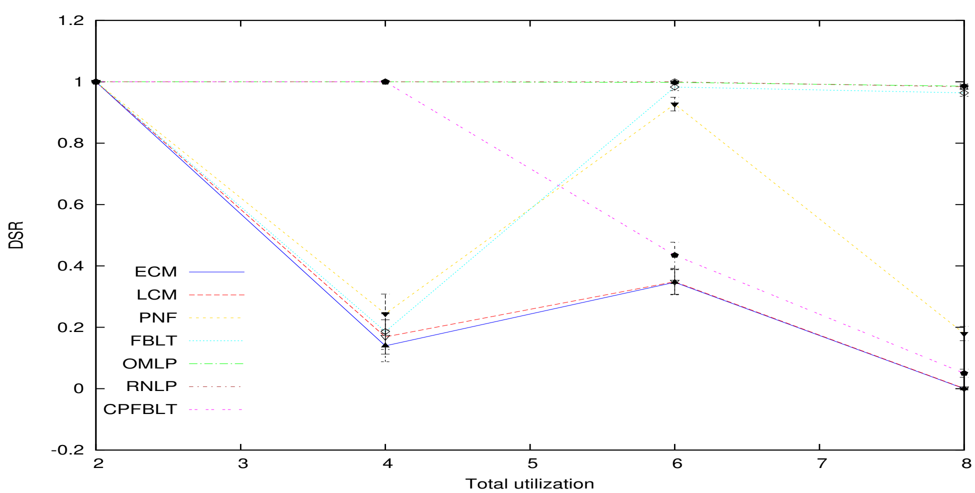


Figure B.210: DSR for Tasksets 210, 480, 750 and 1020

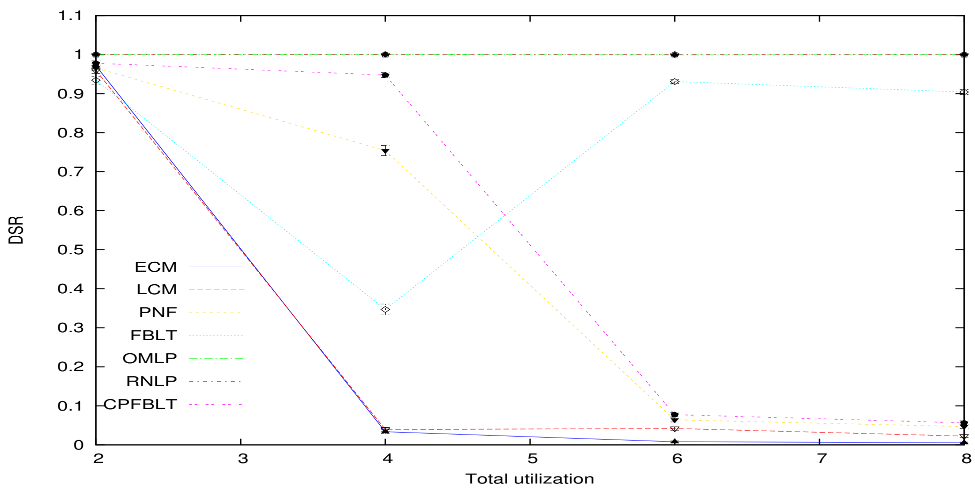


Figure B.211: DSR for Tasksets 211, 481, 751 and 1021

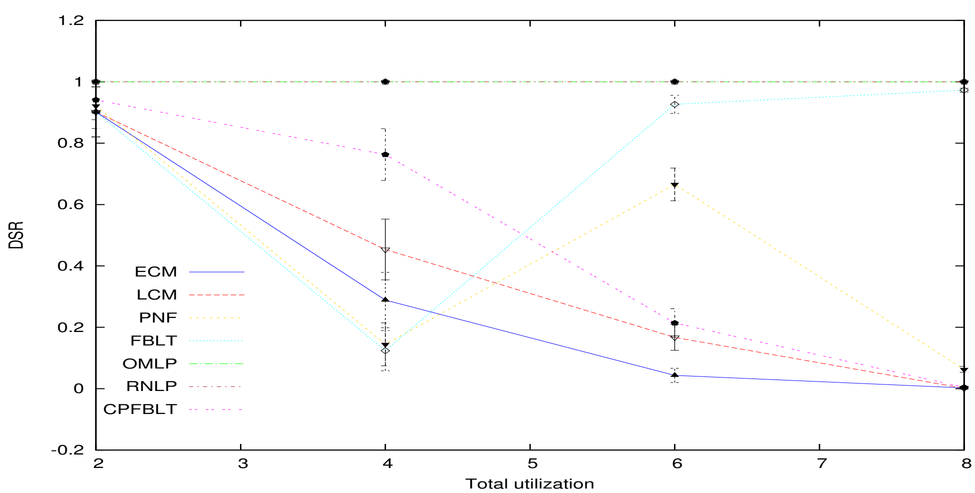


Figure B.212: DSR for Tasksets 212, 482, 752 and 1022

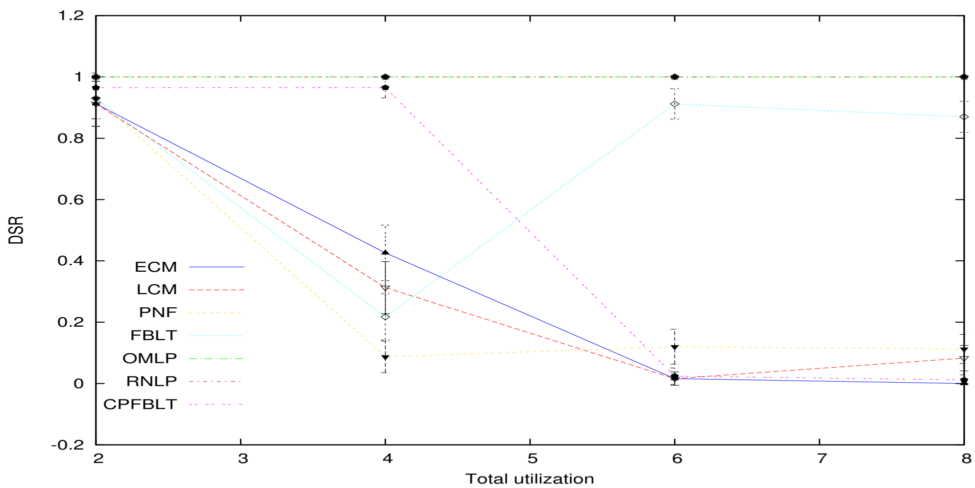


Figure B.213: DSR for Tasksets 213, 483, 753 and 1023

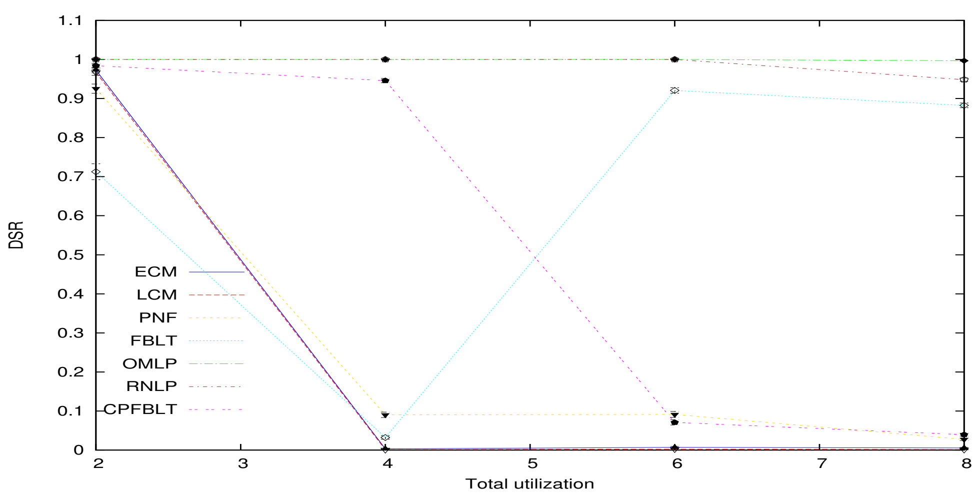


Figure B.214: DSR for Tasksets 214, 484, 754 and 1024

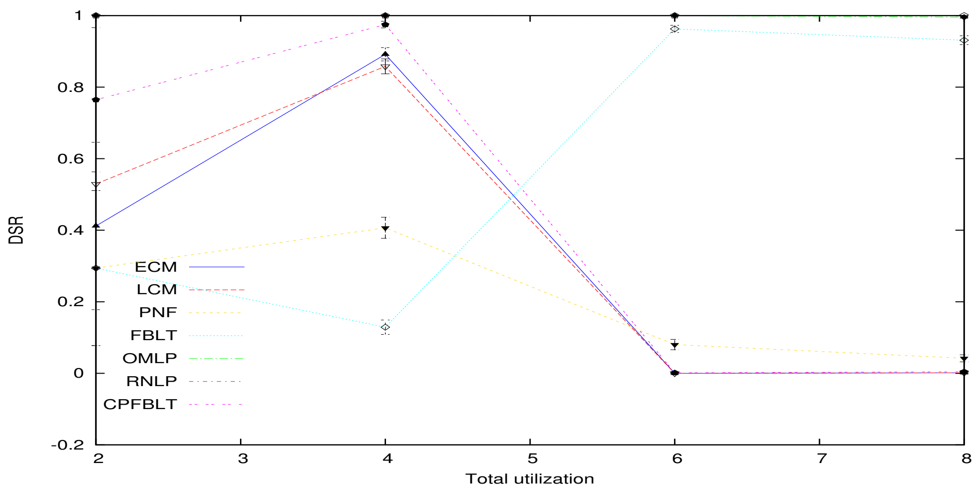


Figure B.215: DSR for Tasksets 215, 485, 755 and 1025

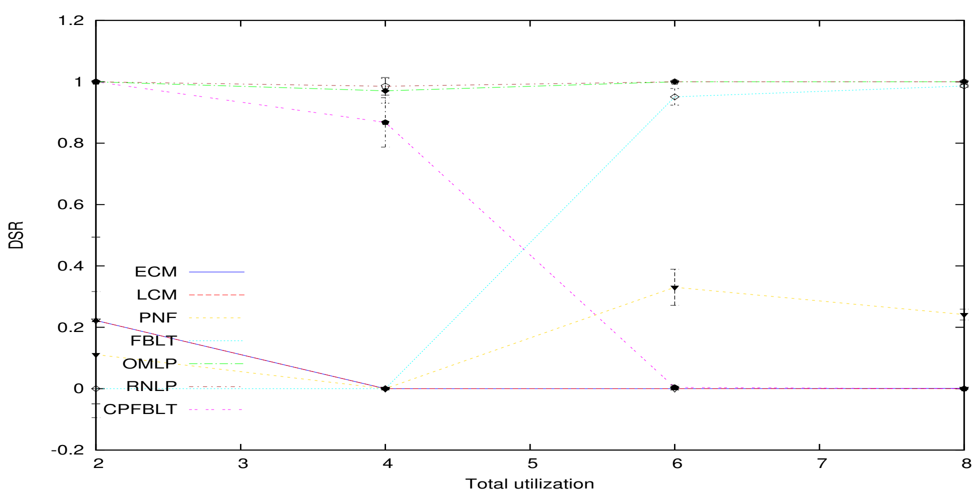


Figure B.216: DSR for Tasksets 216, 486, 756 and 1026

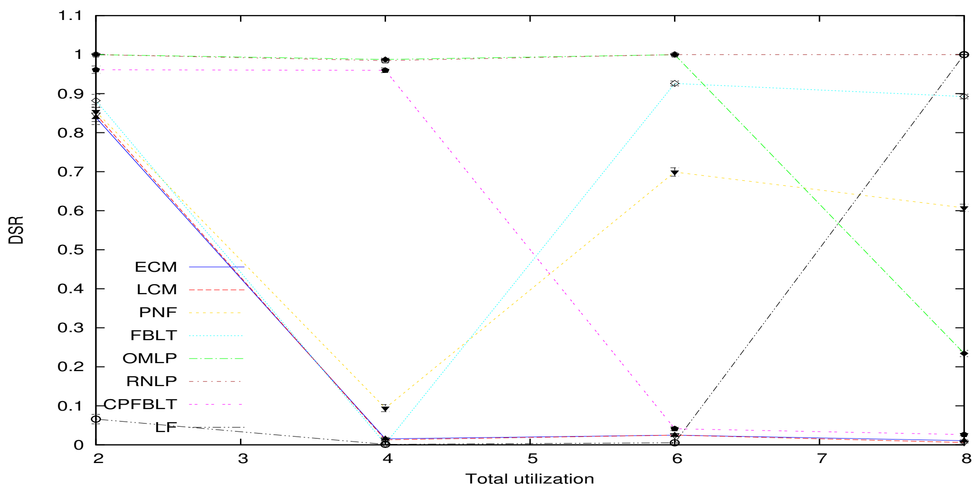


Figure B.217: DSR for Tasksets 217, 487, 757 and 1027

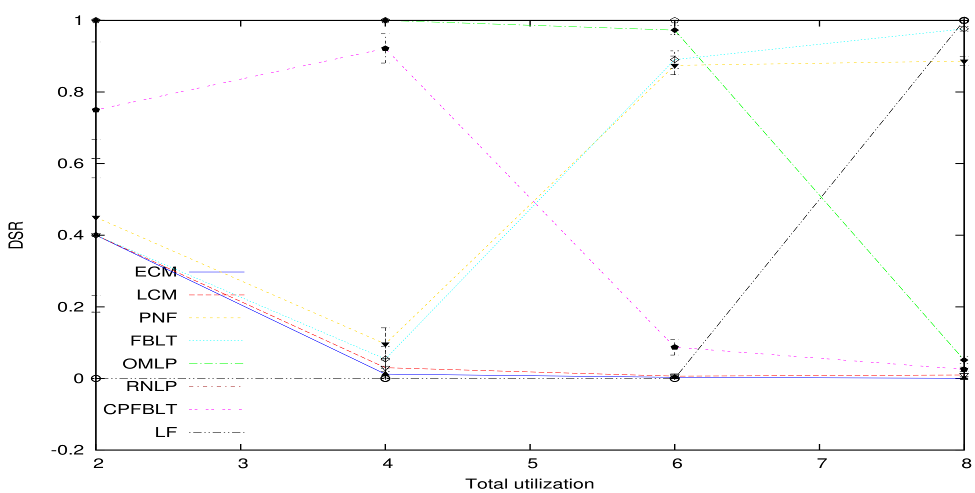


Figure B.218: DSR for Tasksets 218, 488, 758 and 1028

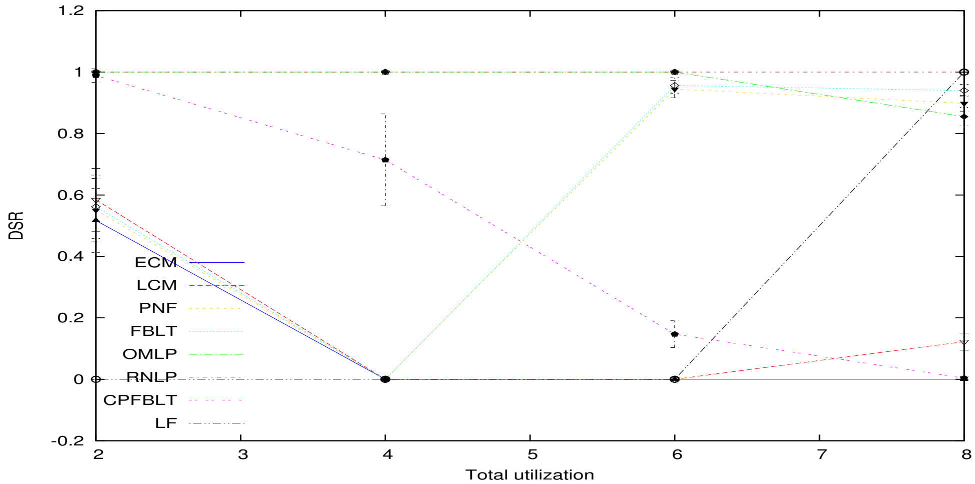


Figure B.219: DSR for Tasksets 219, 489, 759 and 1029

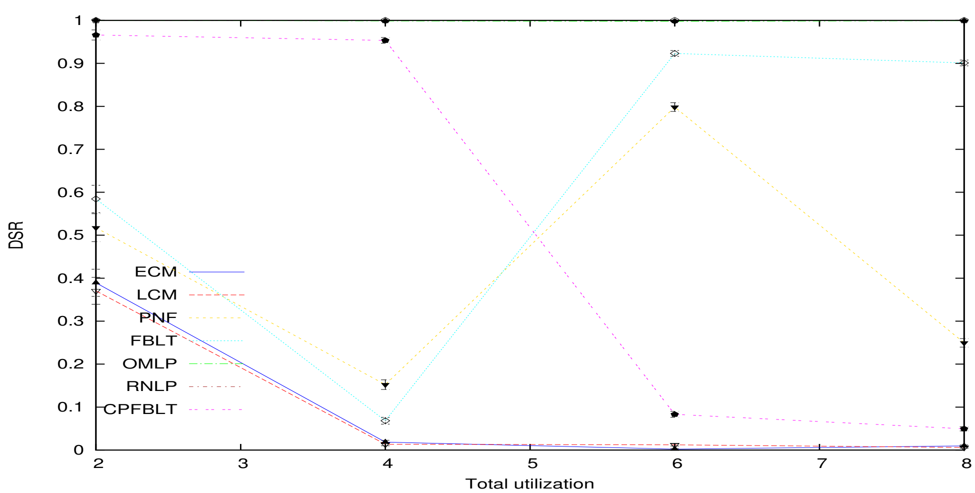


Figure B.220: DSR for Tasksets 220, 490, 760 and 1030

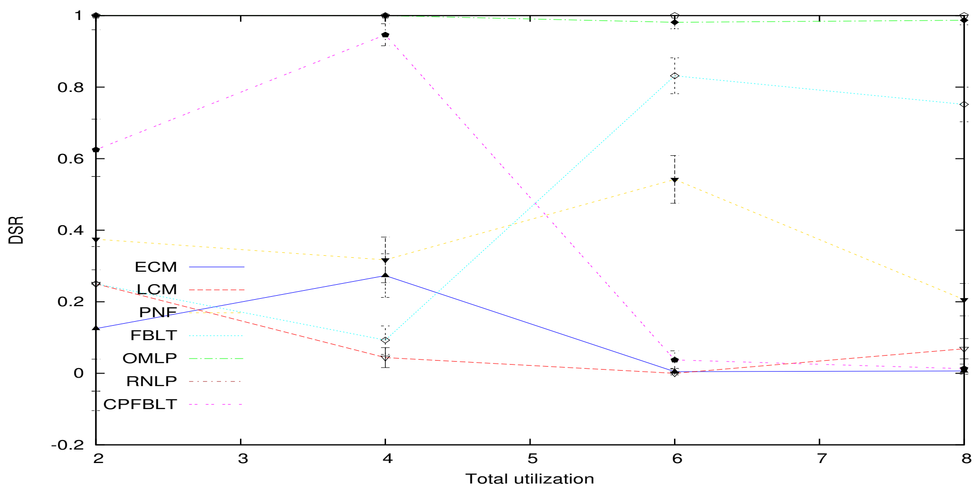


Figure B.221: DSR for Tasksets 221, 491, 761 and 1031

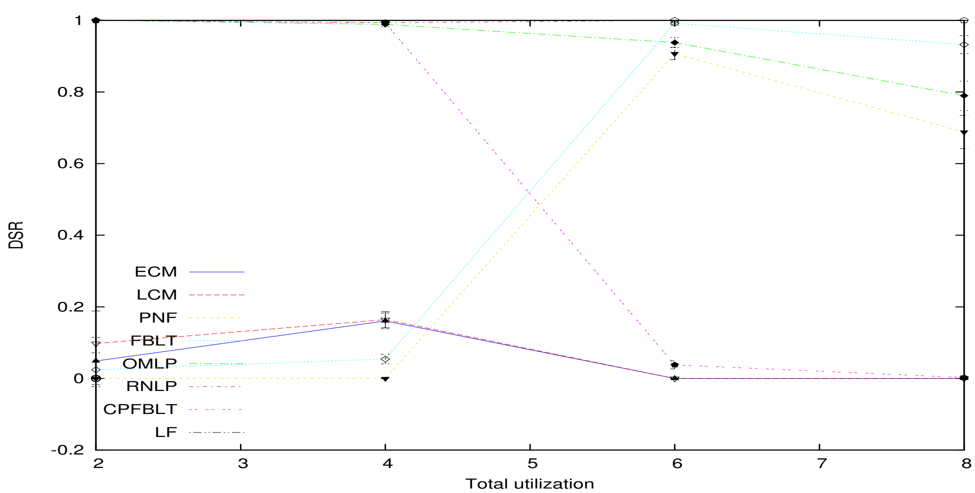


Figure B.222: DSR for Tasksets 222, 492, 762 and 1032

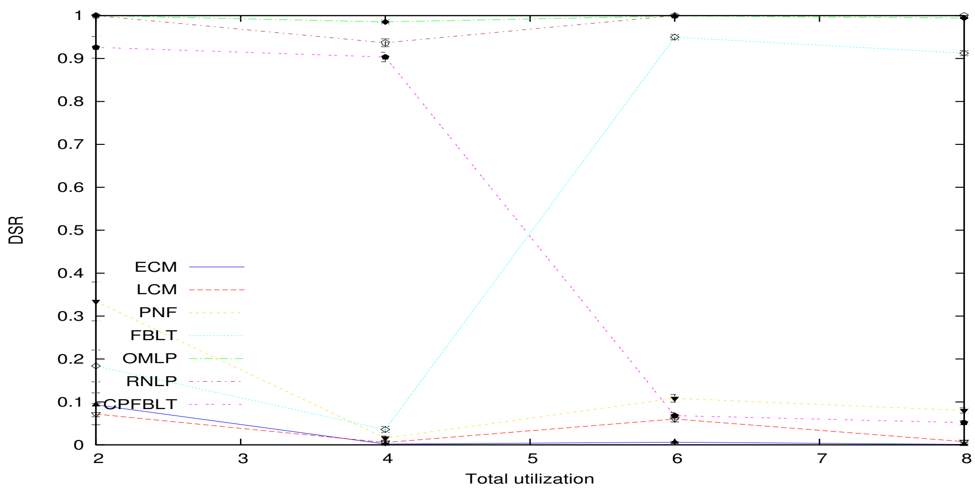


Figure B.223: DSR for Tasksets 223, 493, 763 and 1033

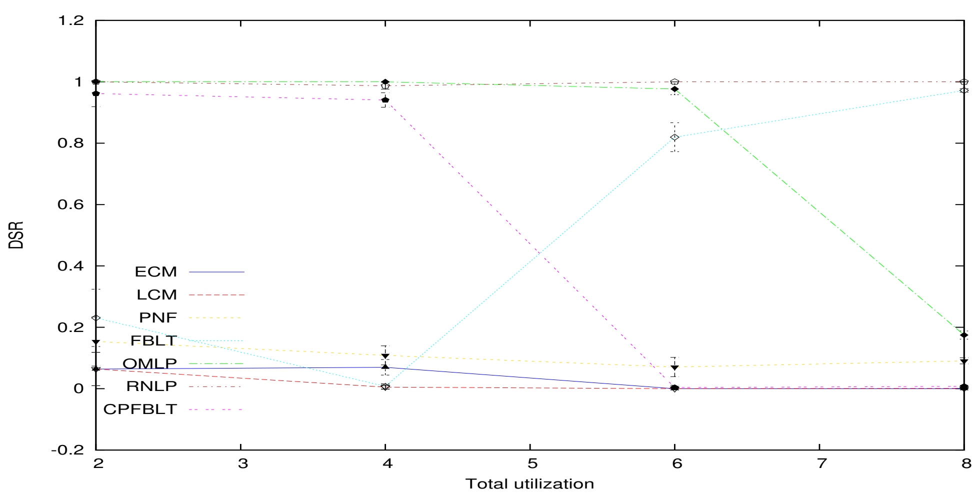


Figure B.224: DSR for Tasksets 224, 494, 764 and 1034

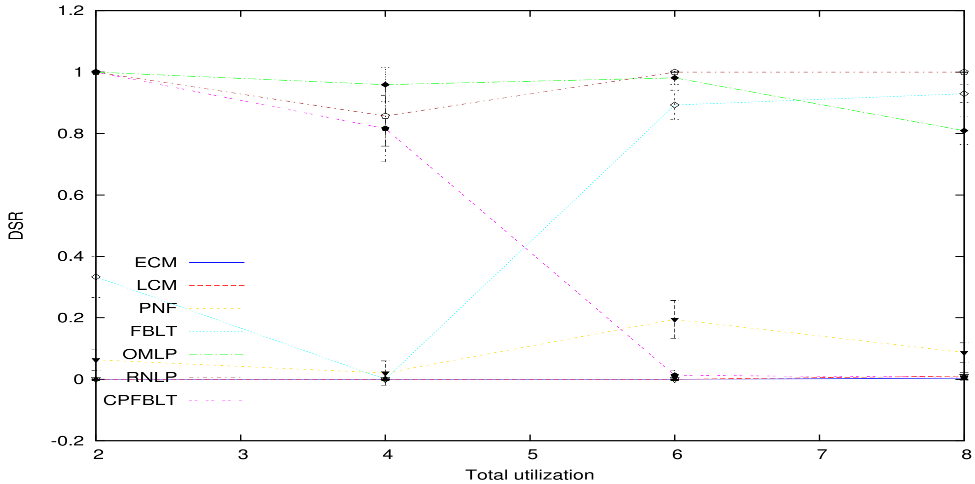


Figure B.225: DSR for Tasksets 225, 495, 765 and 1035

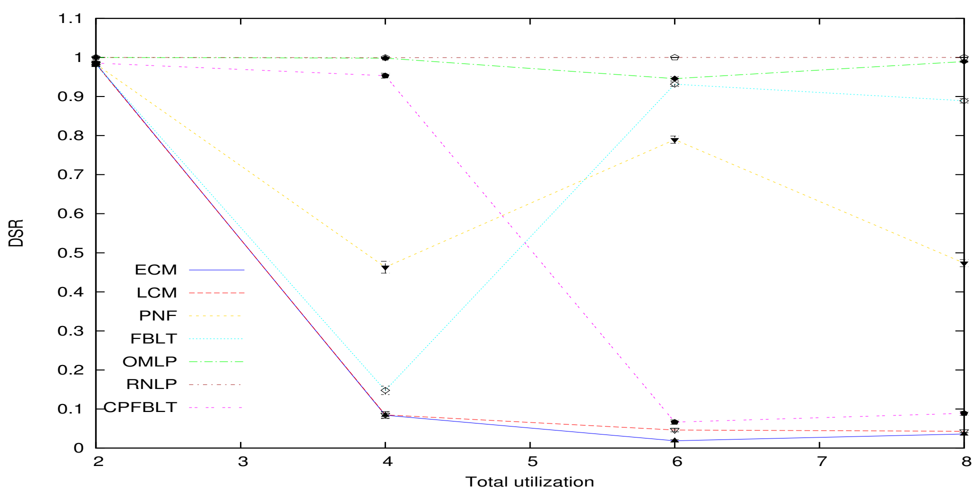


Figure B.226: DSR for Tasksets 226, 496, 766 and 1036

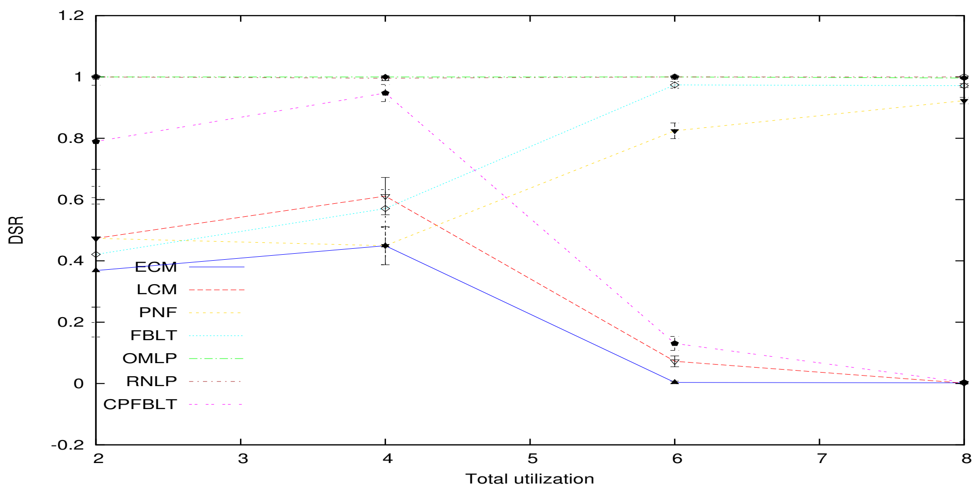


Figure B.227: DSR for Tasksets 227, 497, 767 and 1037

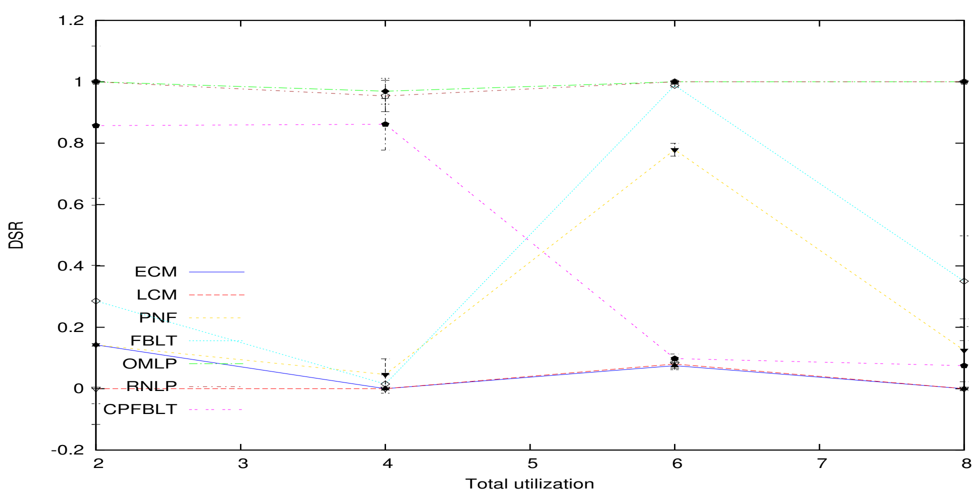


Figure B.228: DSR for Tasksets 228, 498, 768 and 1038

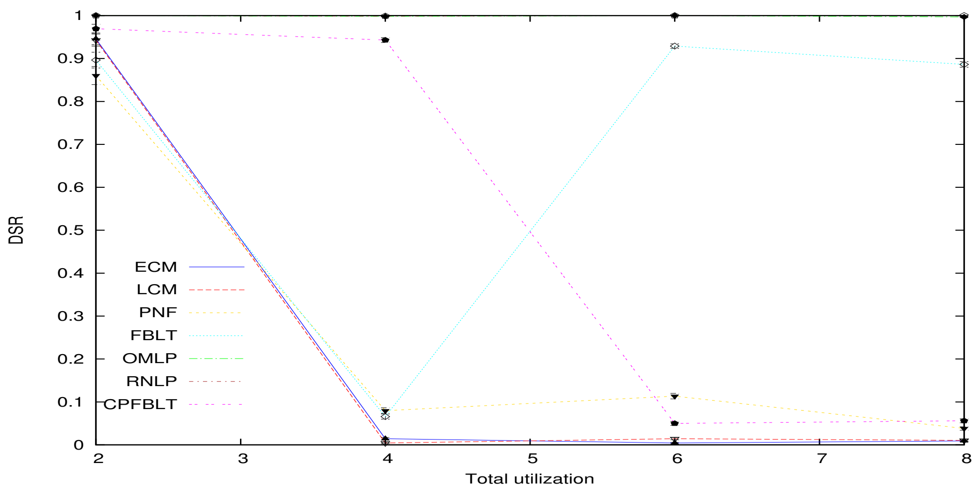


Figure B.229: DSR for Tasksets 229, 499, 769 and 1039

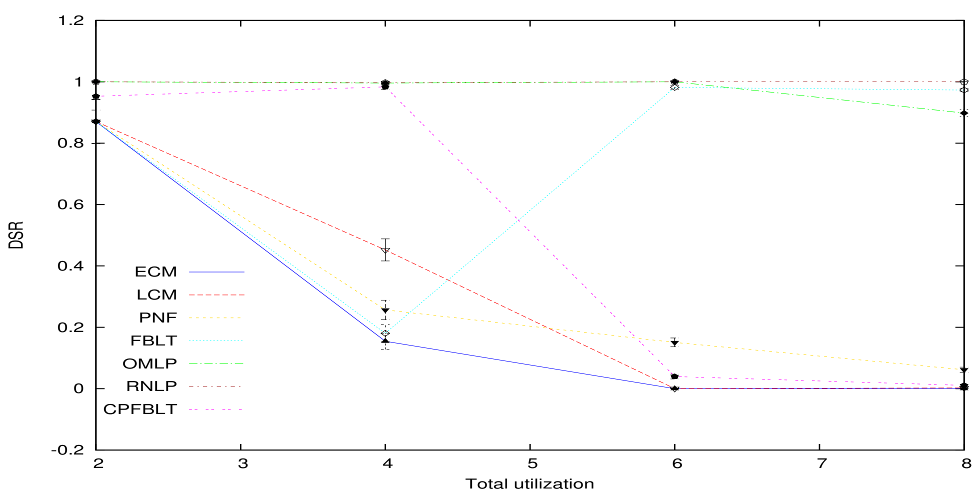


Figure B.230: DSR for Tasksets 230, 500, 770 and 1040

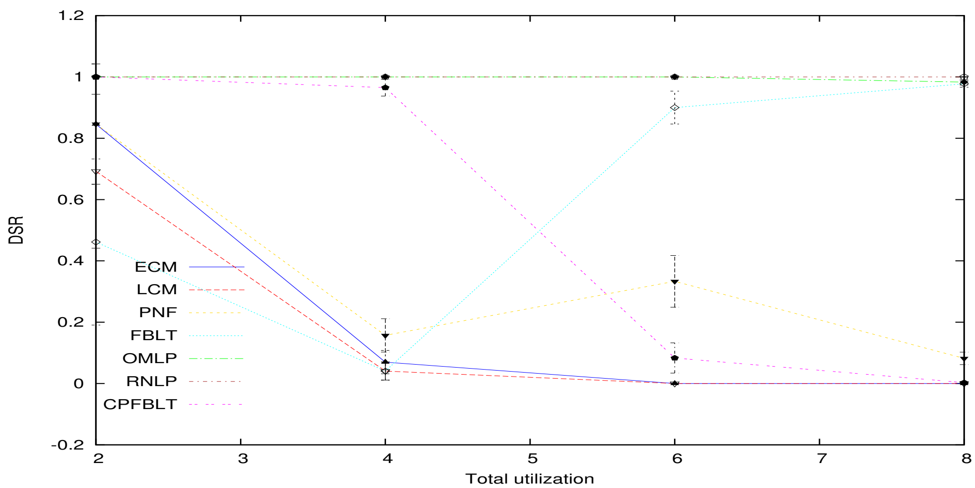


Figure B.231: DSR for Tasksets 231, 501, 771 and 1041

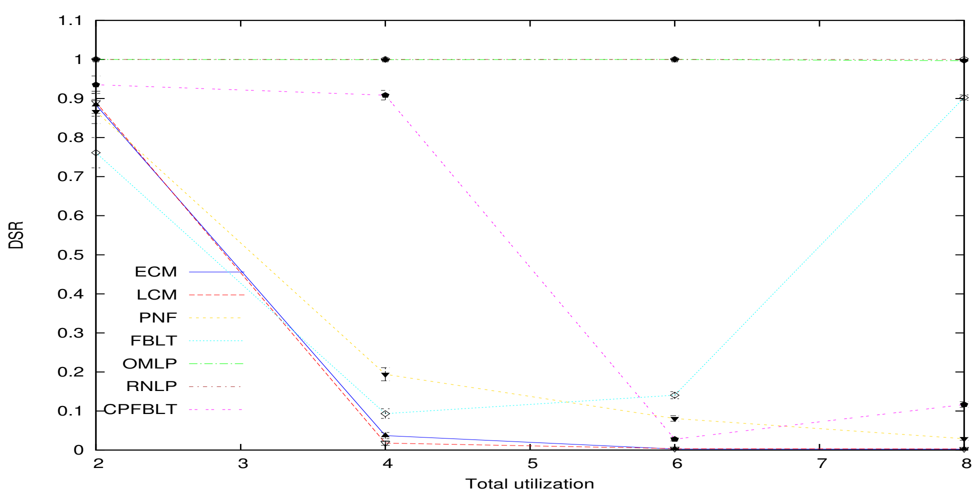


Figure B.232: DSR for Tasksets 232, 502, 772 and 1042

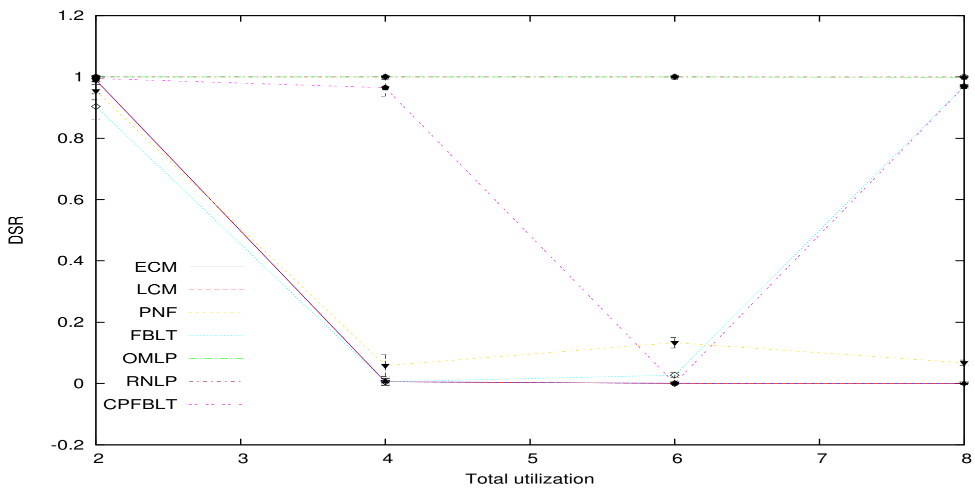


Figure B.233: DSR for Tasksets 233, 503, 773 and 1043

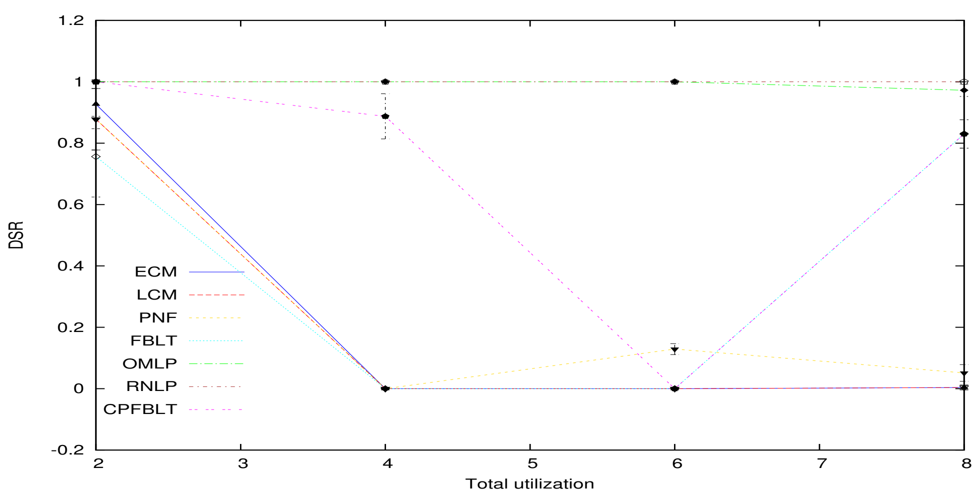


Figure B.234: DSR for Tasksets 234, 504, 774 and 1044

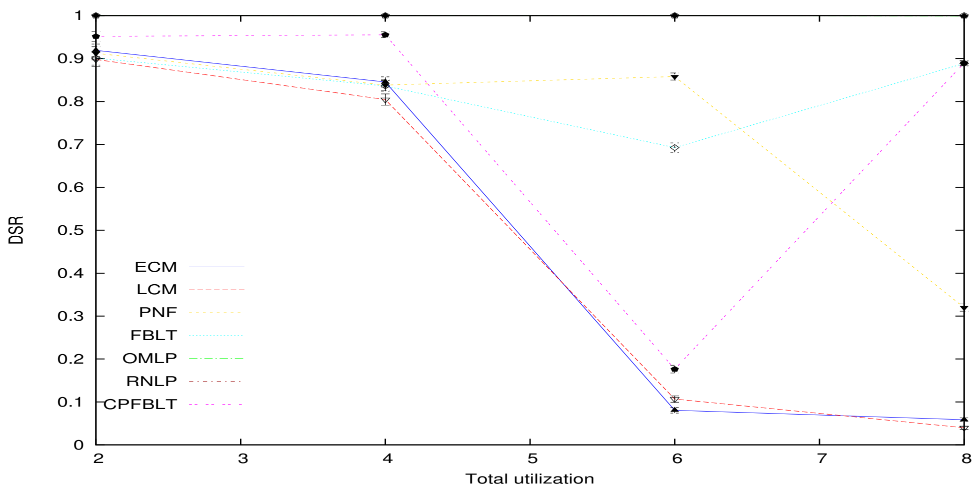


Figure B.235: DSR for Tasksets 235, 505, 775 and 1045

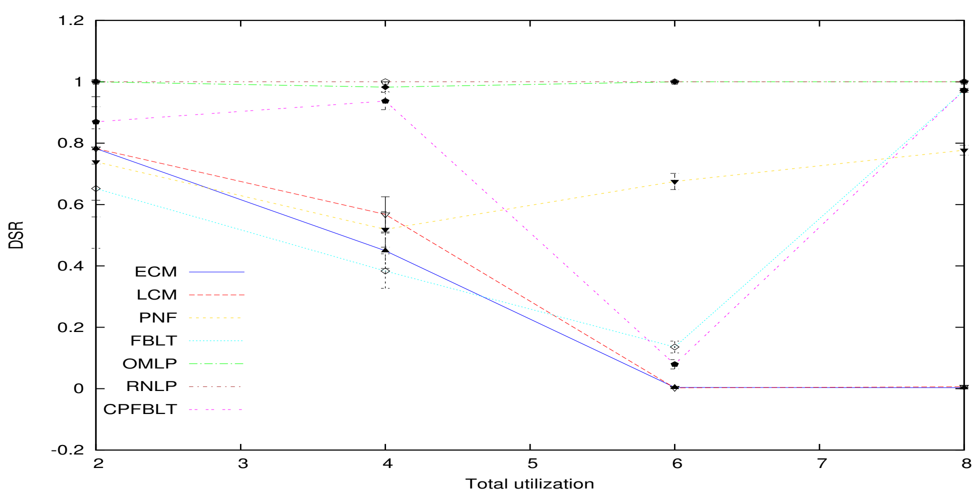


Figure B.236: DSR for Tasksets 236, 506, 776 and 1046

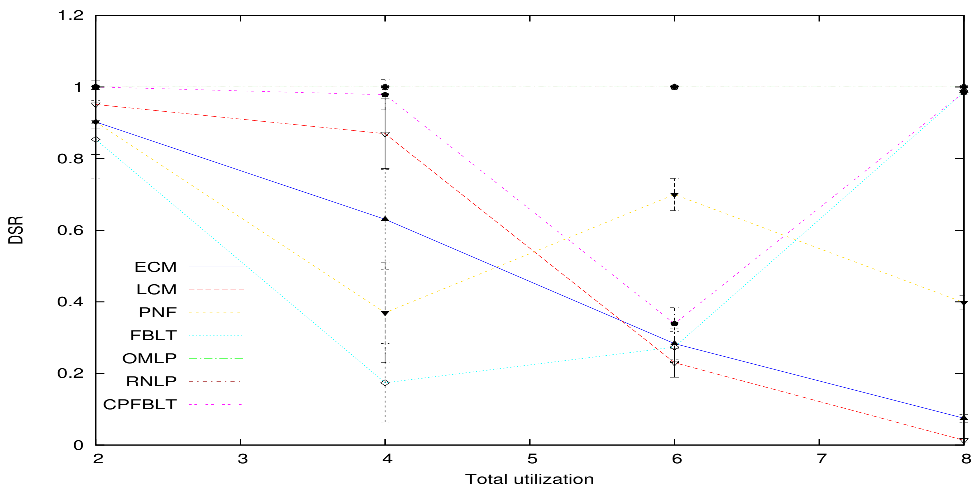


Figure B.237: DSR for Tasksets 237, 507, 777 and 1047

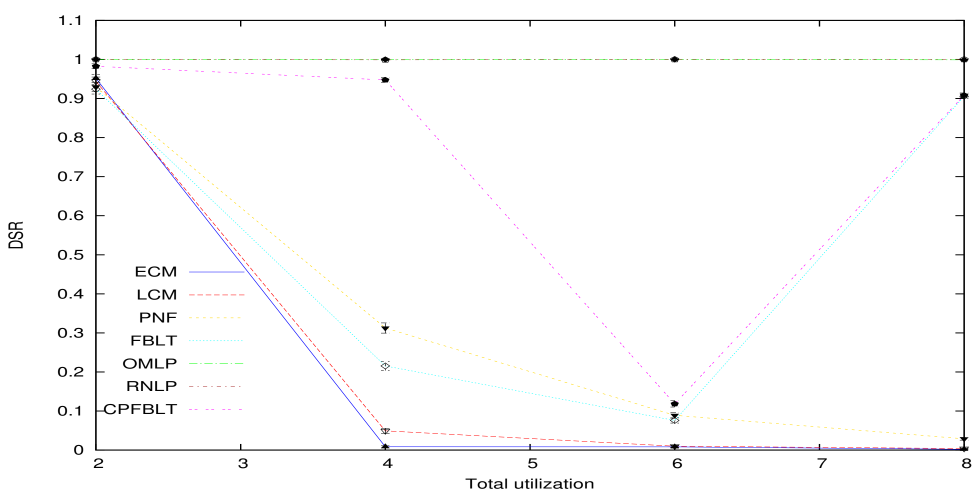


Figure B.238: DSR for Tasksets 238, 508, 778 and 1048

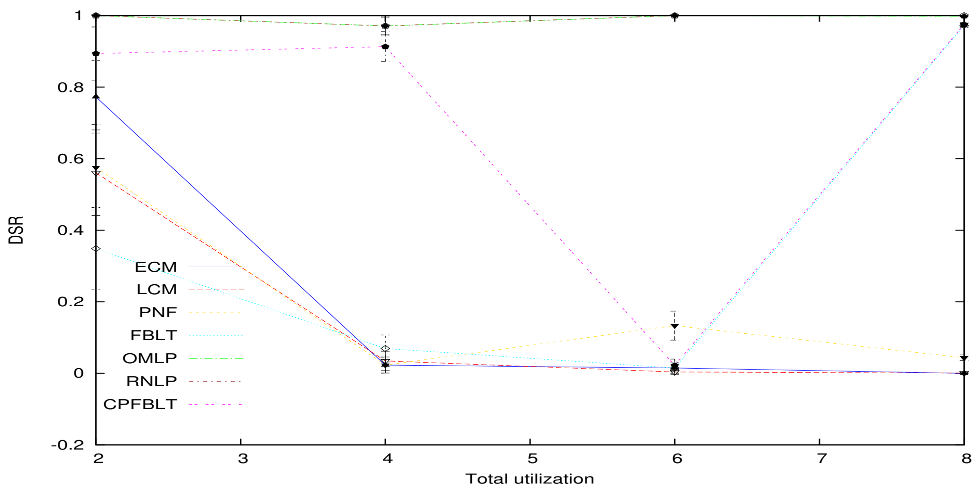


Figure B.239: DSR for Tasksets 239, 509, 779 and 1049

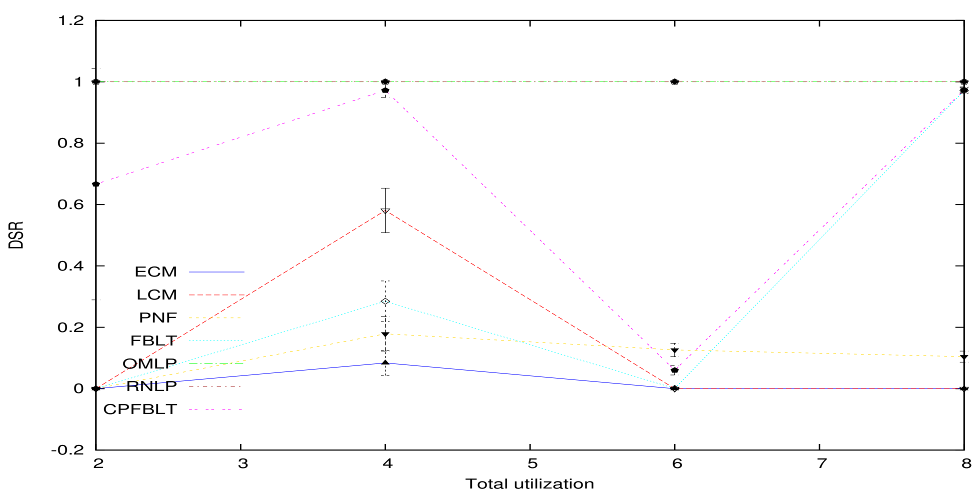


Figure B.240: DSR for Tasksets 240, 510, 780 and 1050

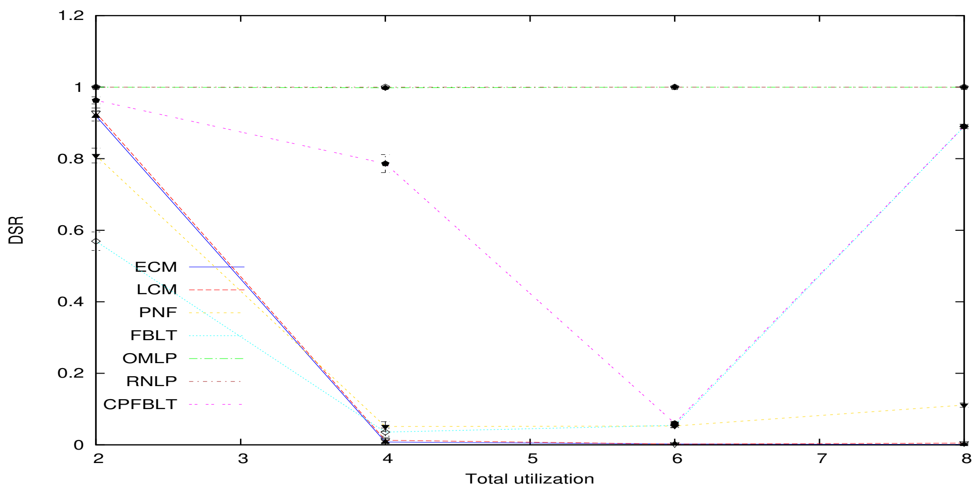


Figure B.241: DSR for Tasksets 241, 511, 781 and 1051

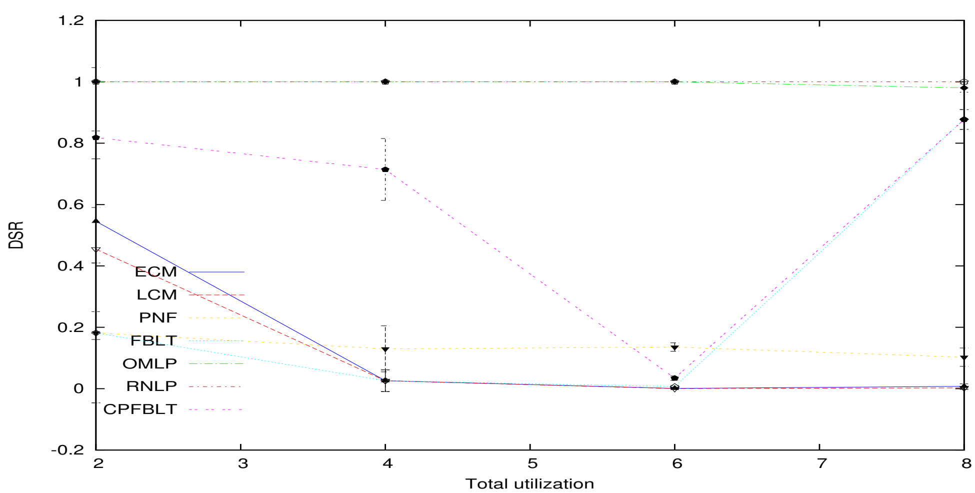


Figure B.242: DSR for Tasksets 242, 512, 782 and 1052

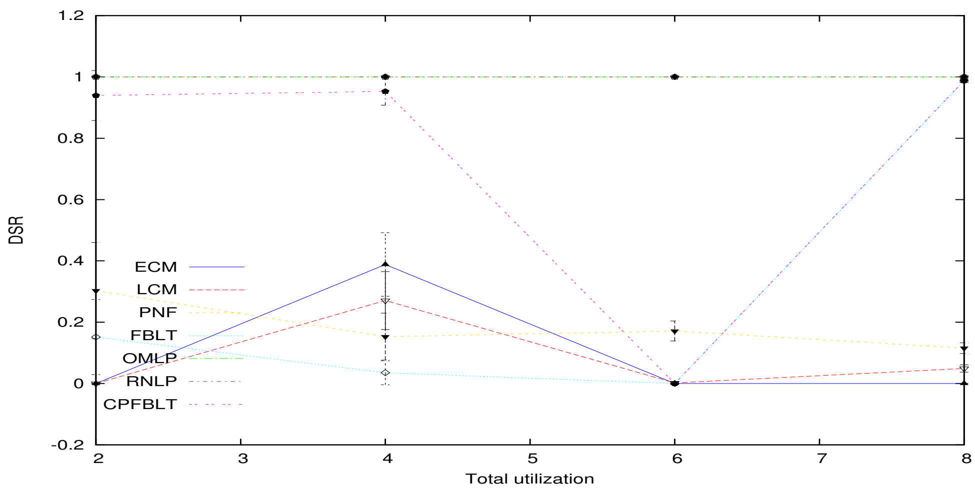


Figure B.243: DSR for Tasksets 243, 513, 783 and 1053

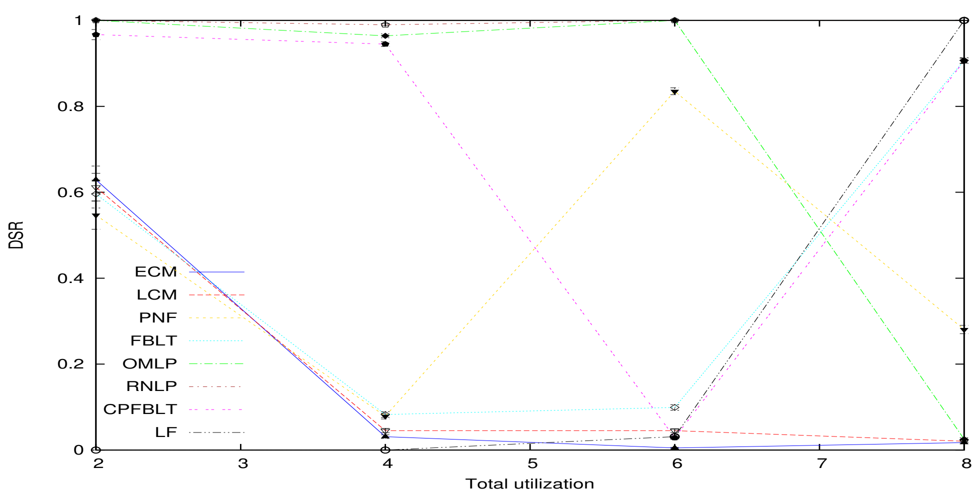


Figure B.244: DSR for Tasksets 244, 514, 784 and 1054

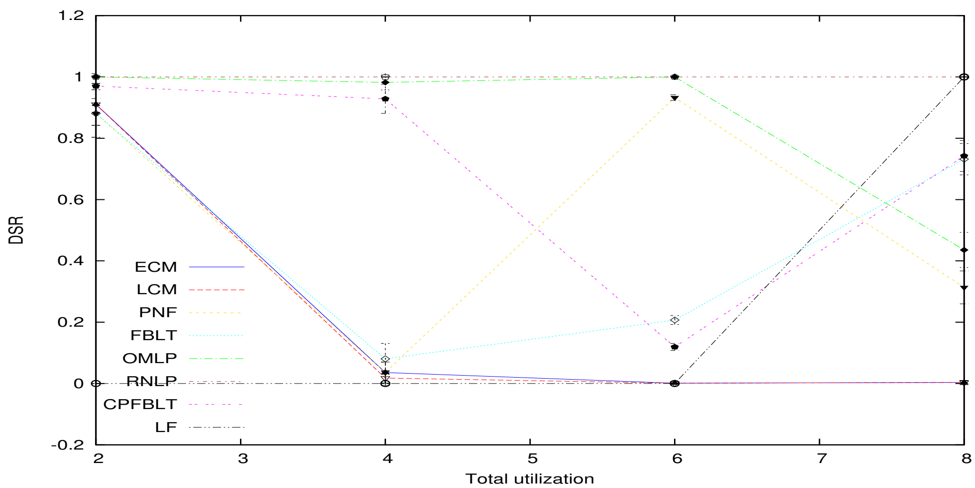


Figure B.245: DSR for Tasksets 245, 515, 785 and 1055

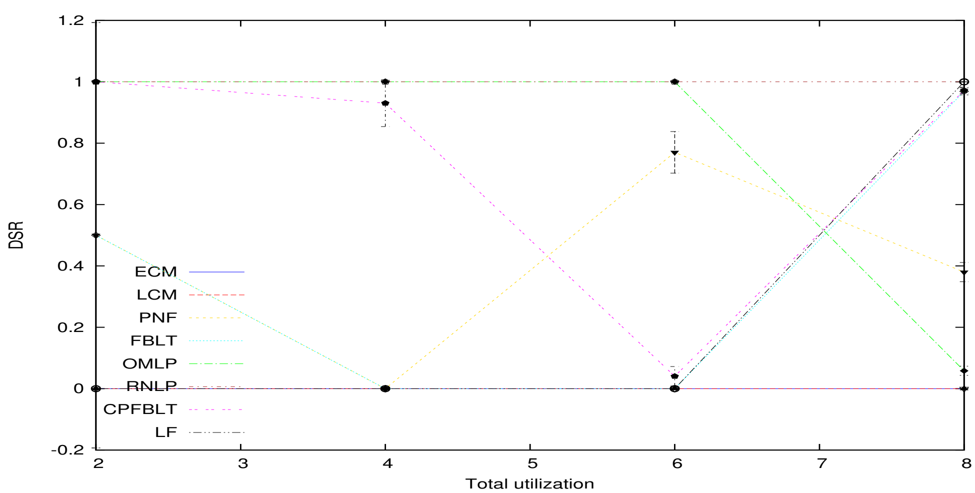


Figure B.246: DSR for Tasksets 246, 516, 786 and 1056

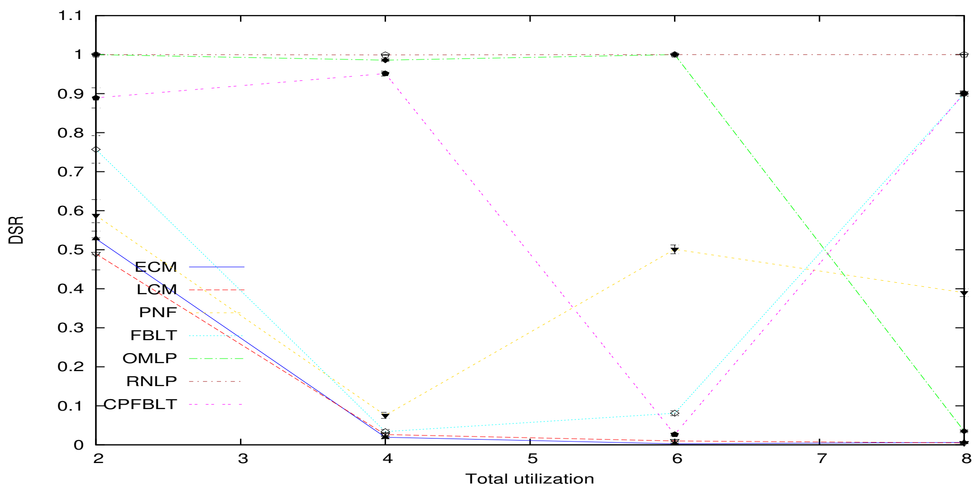


Figure B.247: DSR for Tasksets 247, 517, 787 and 1057

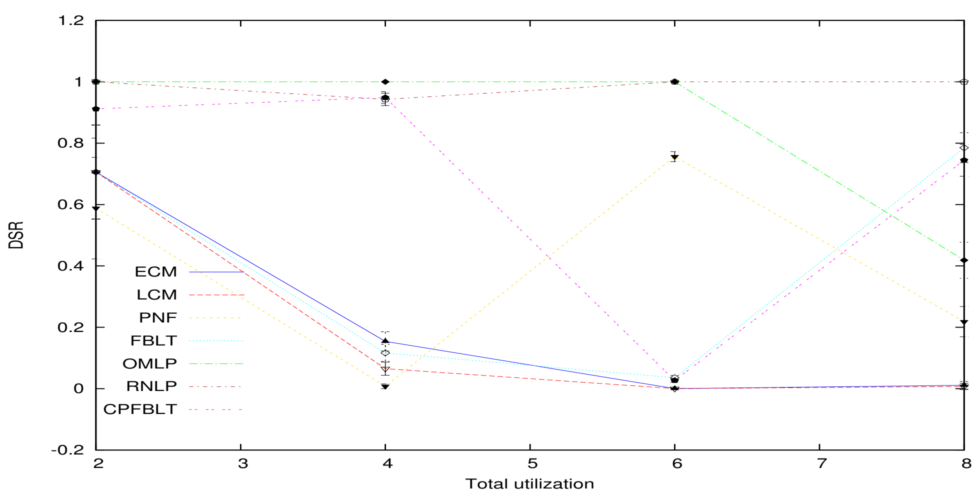


Figure B.248: DSR for Tasksets 248, 518, 788 and 1058

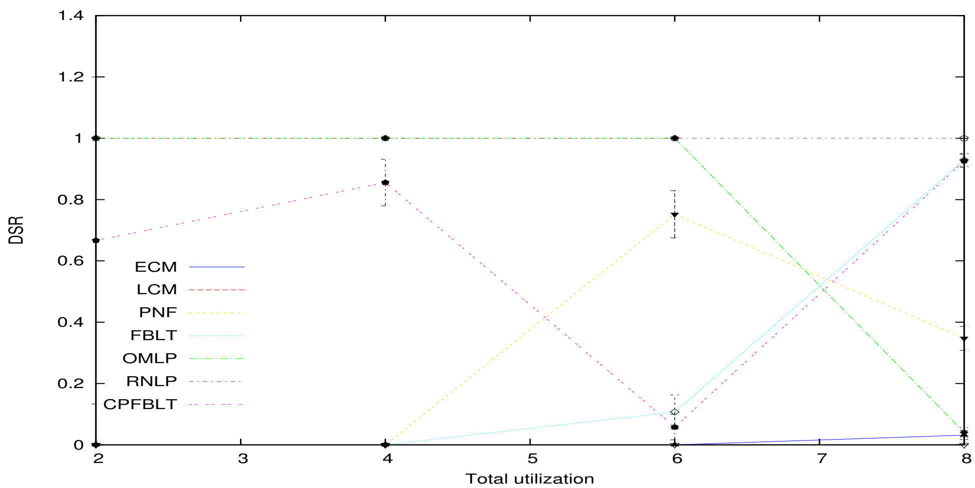


Figure B.249: DSR for Tasksets 249, 519, 789 and 1059

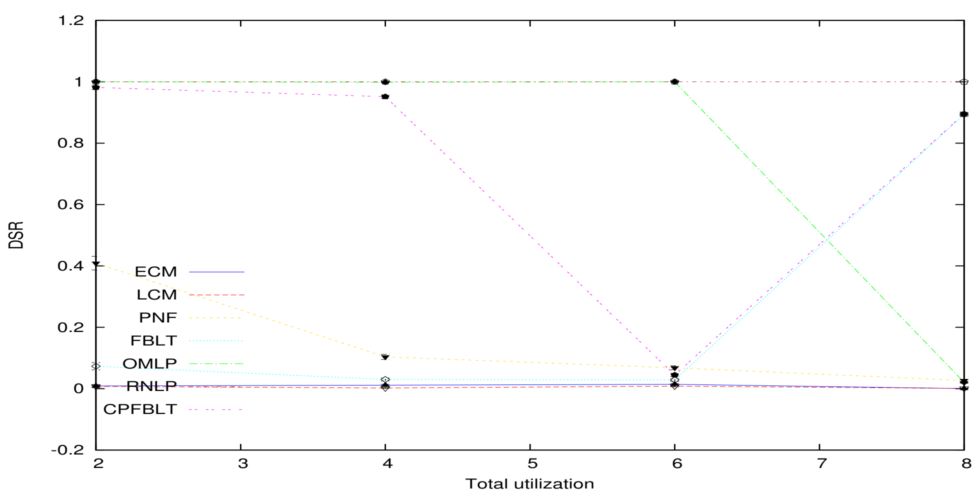


Figure B.250: DSR for Tasksets 250, 520, 790 and 1060

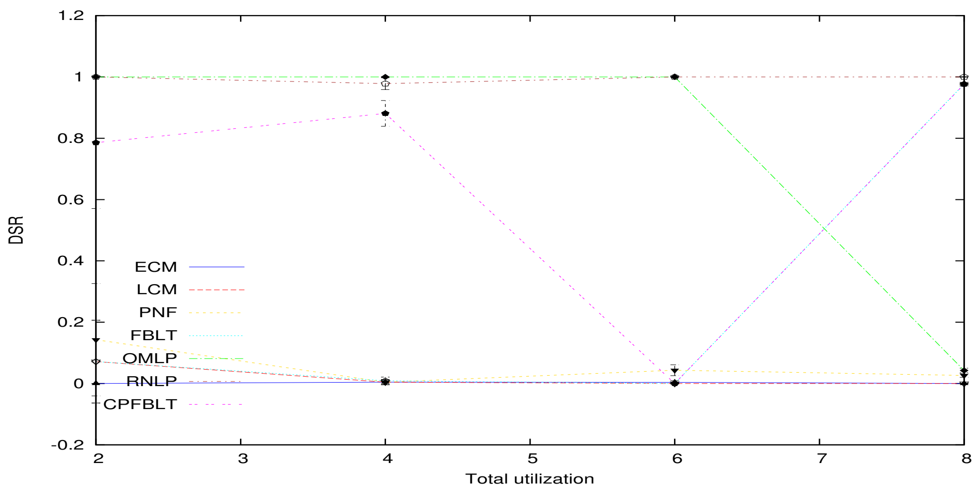


Figure B.251: DSR for Tasksets 251, 521, 791 and 1061

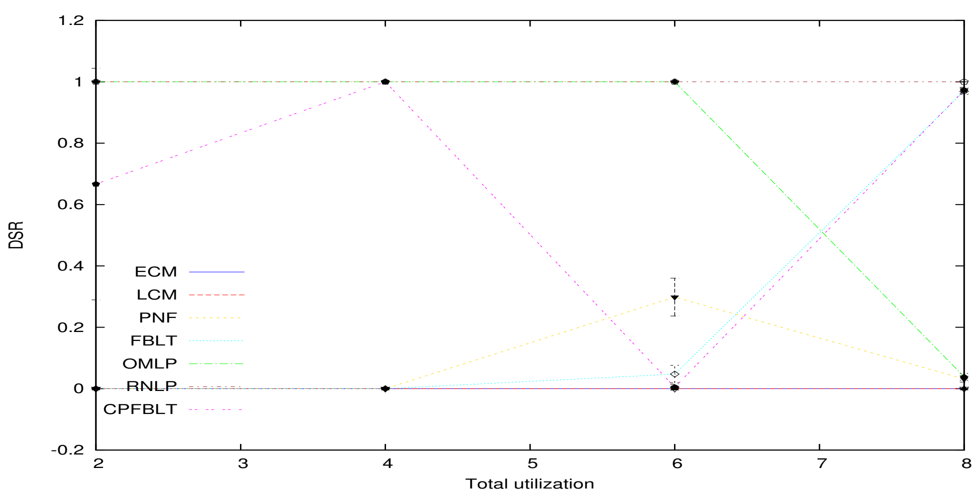


Figure B.252: DSR for Tasksets 252, 522, 792 and 1062

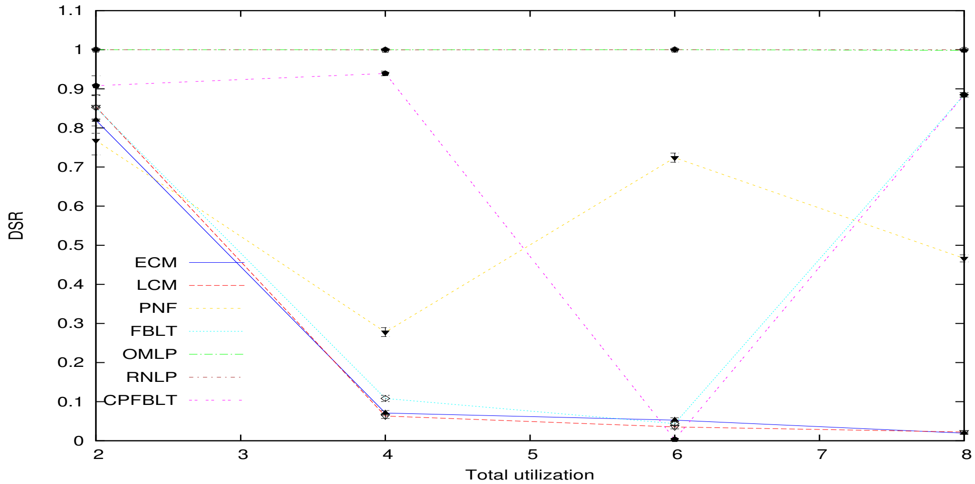


Figure B.253: DSR for Tasksets 253, 523, 793 and 1063

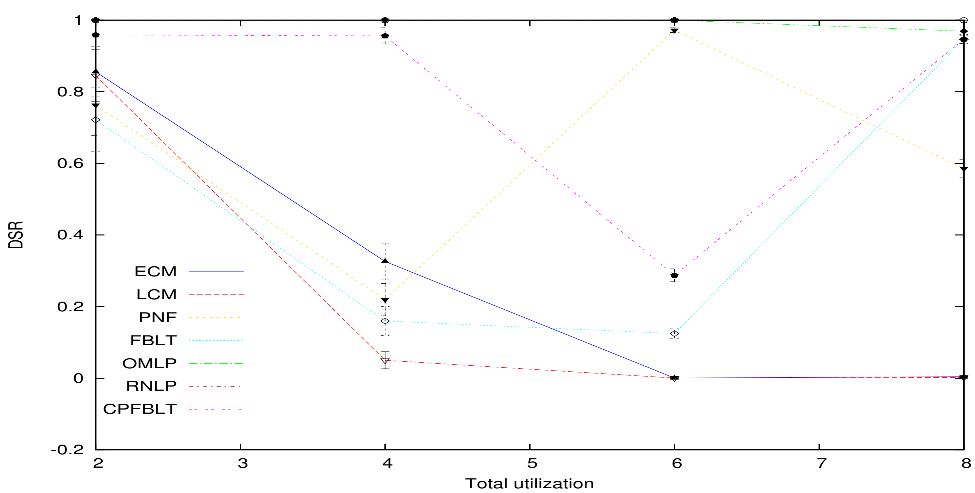


Figure B.254: DSR for Tasksets 254, 524, 794 and 1064

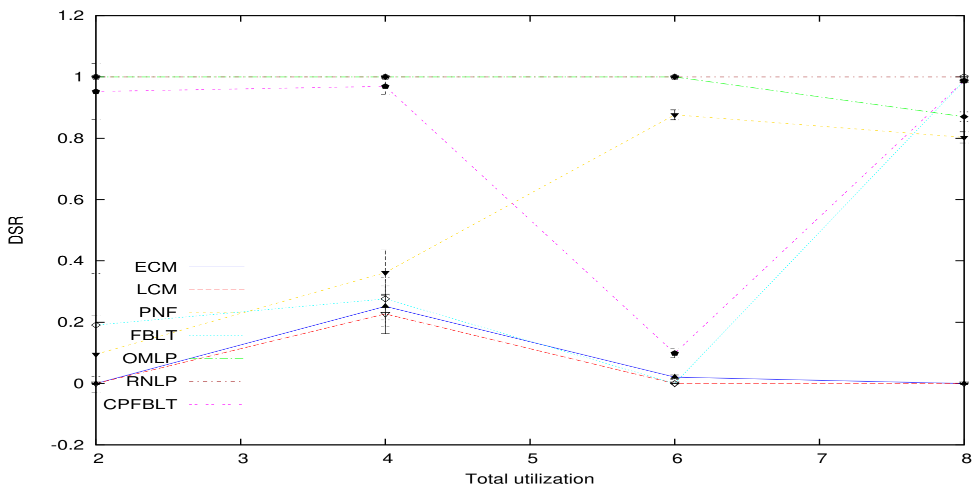


Figure B.255: DSR for Tasksets 255, 525, 795 and 1065

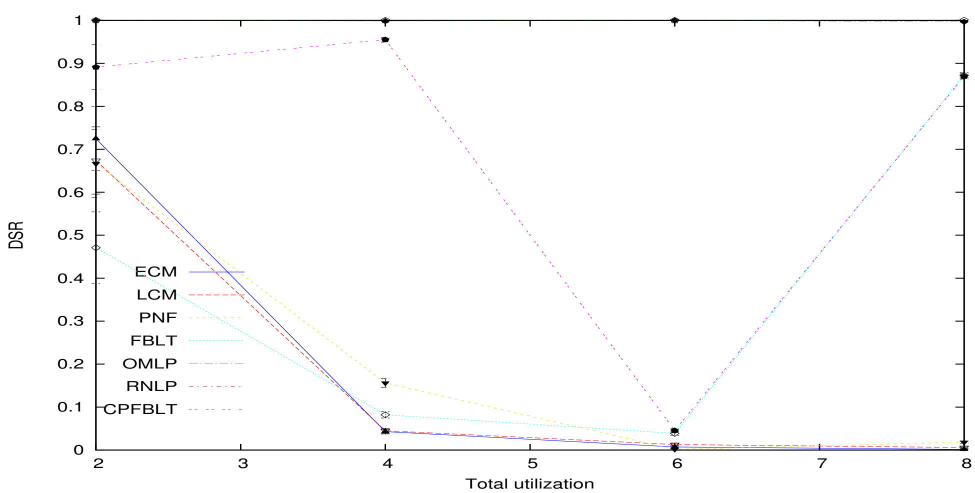


Figure B.256: DSR for Tasksets 256, 526, 796 and 1066

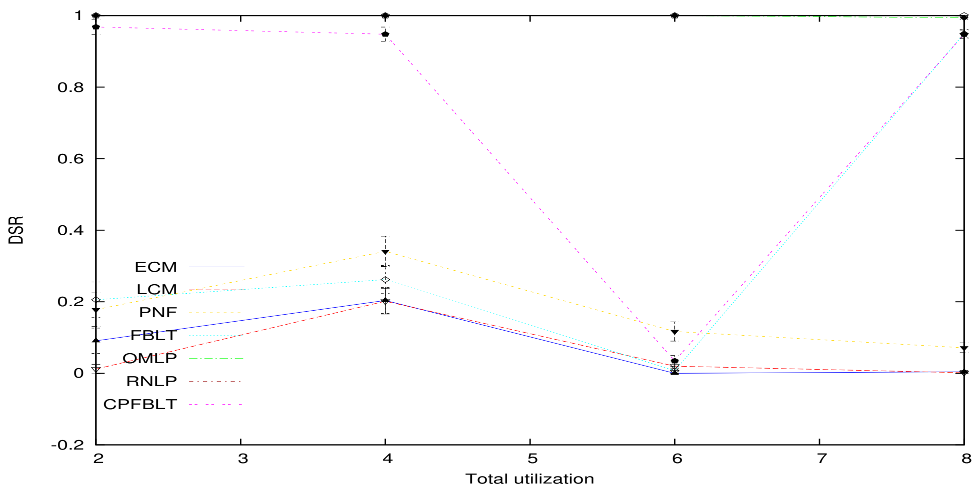


Figure B.257: DSR for Tasksets 257, 527, 797 and 1067

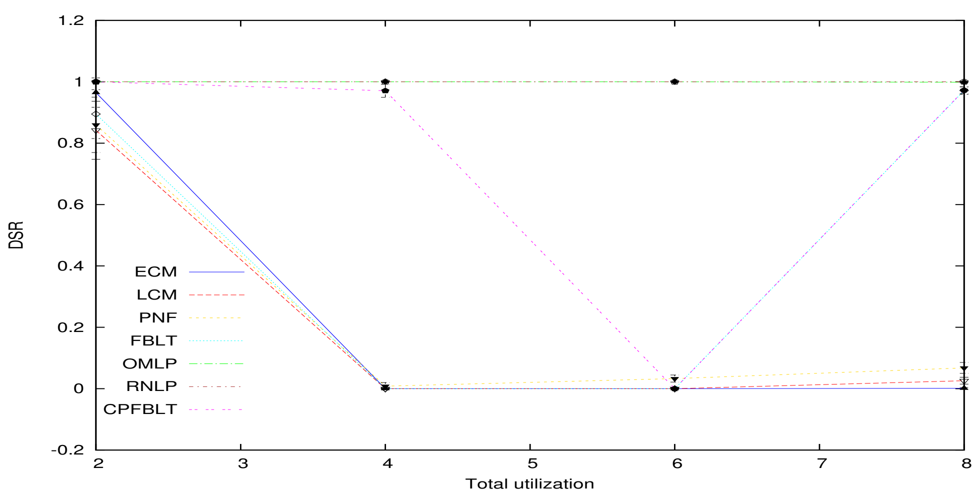


Figure B.258: DSR for Tasksets 258, 528, 798 and 1068

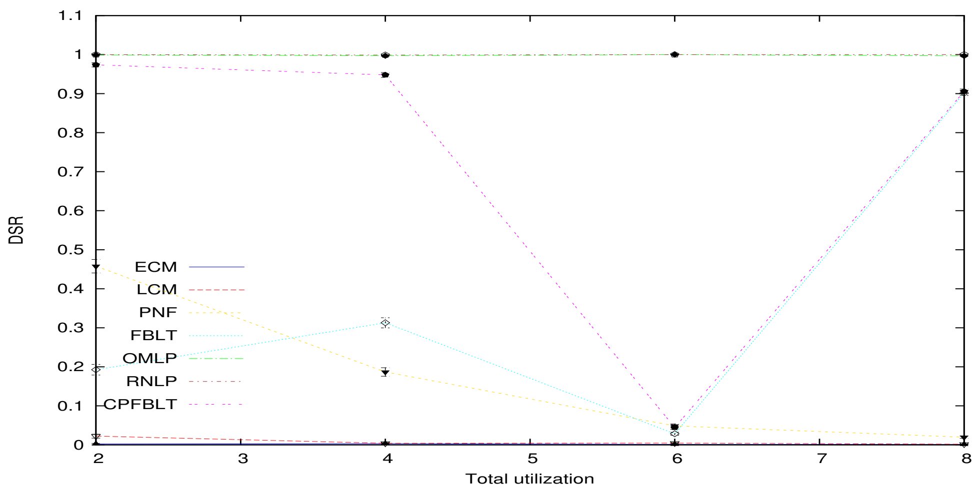


Figure B.259: DSR for Tasksets 259, 529, 799 and 1069

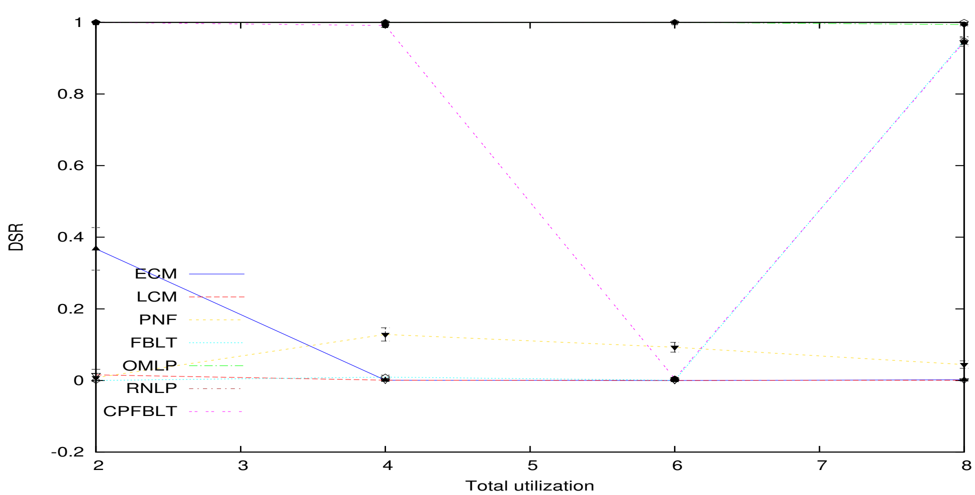


Figure B.260: DSR for Tasksets 260, 530, 800 and 1070

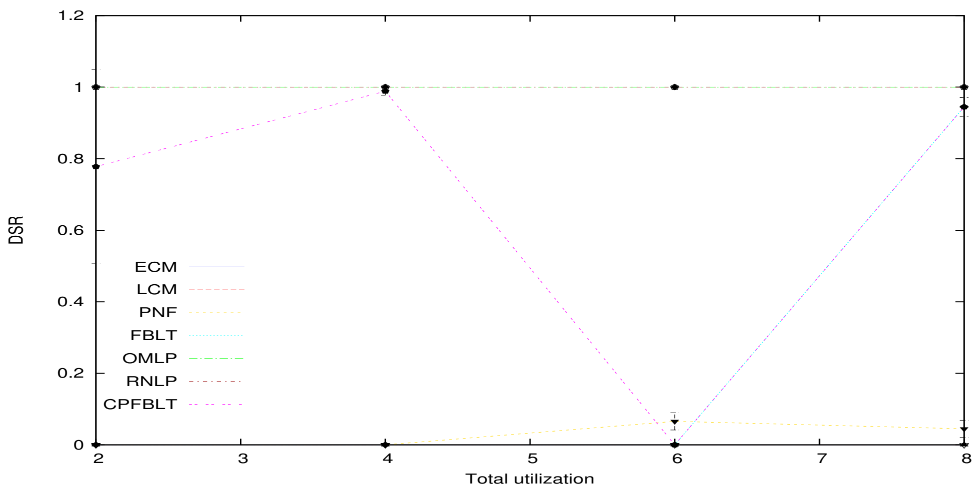


Figure B.261: DSR for Tasksets 261, 531, 801 and 1071

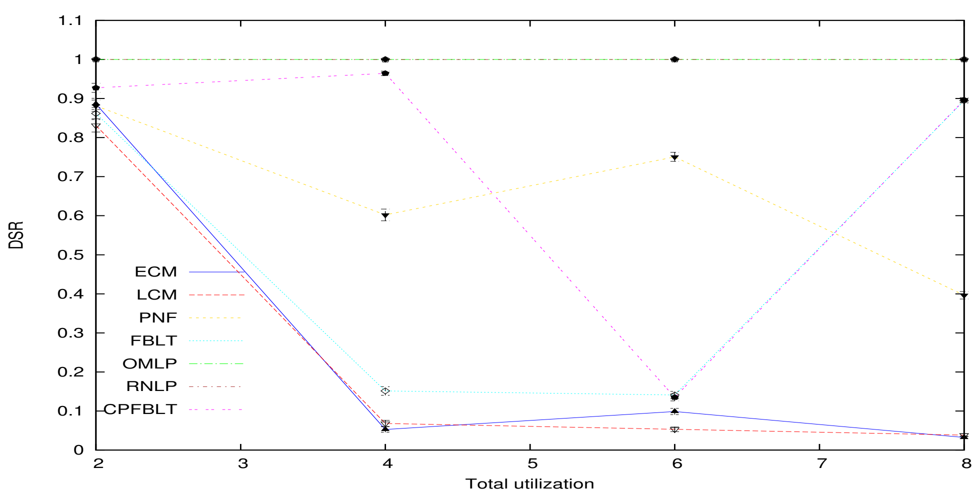


Figure B.262: DSR for Tasksets 262, 532, 802 and 1072

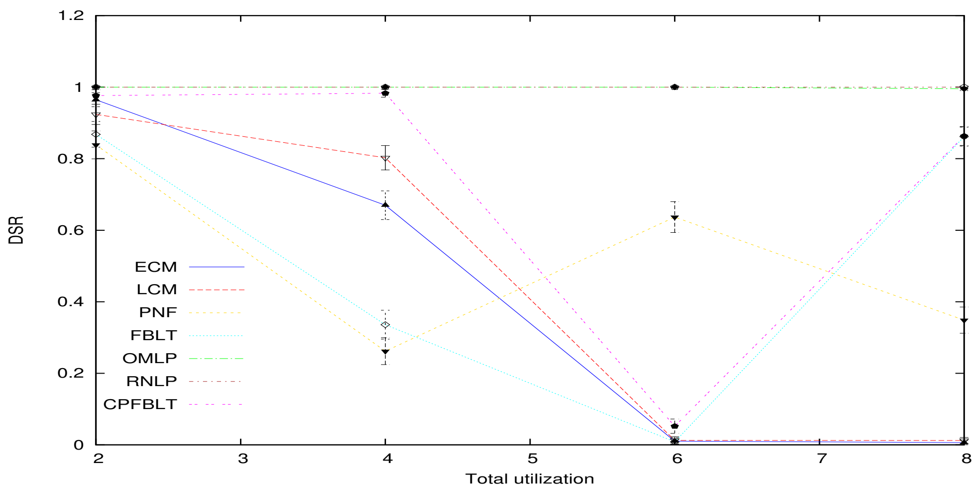


Figure B.263: DSR for Tasksets 263, 533, 803 and 1073

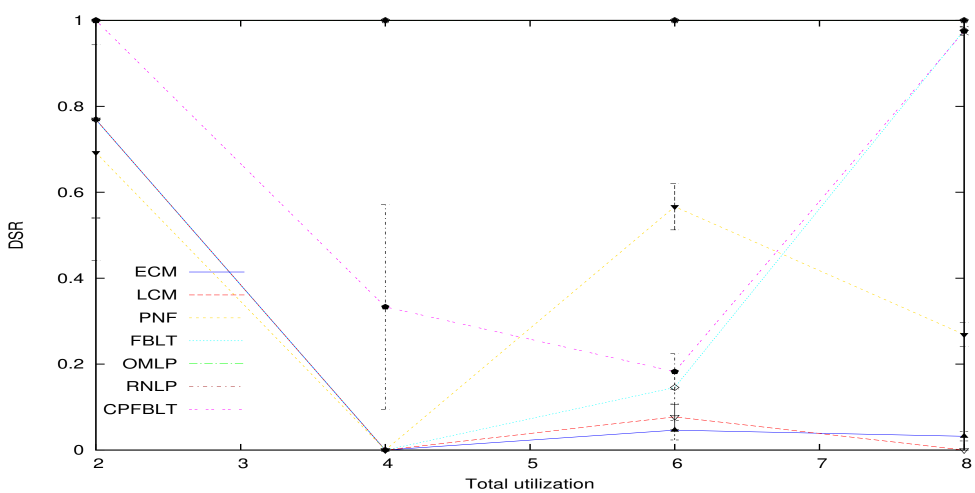


Figure B.264: DSR for Tasksets 264, 534, 804 and 1074

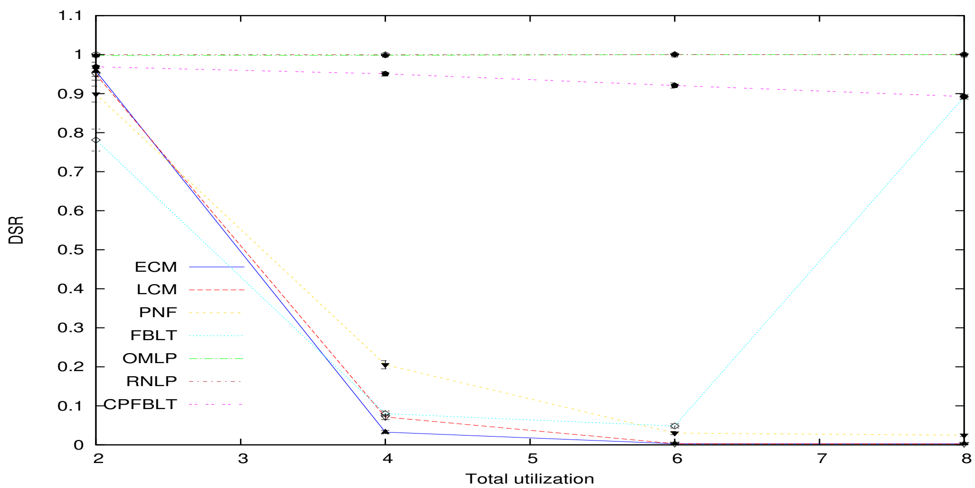


Figure B.265: DSR for Tasksets 265, 535, 805 and 1075

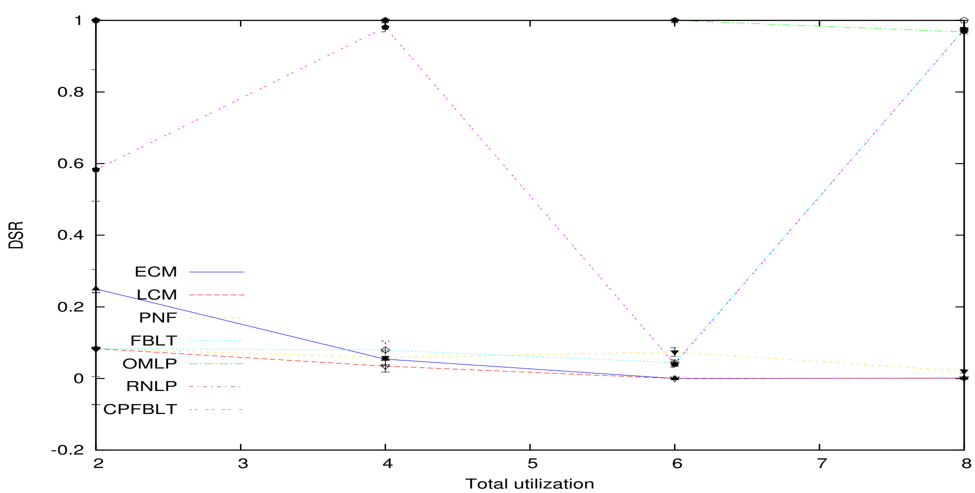


Figure B.266: DSR for Tasksets 266, 536, 806 and 1076

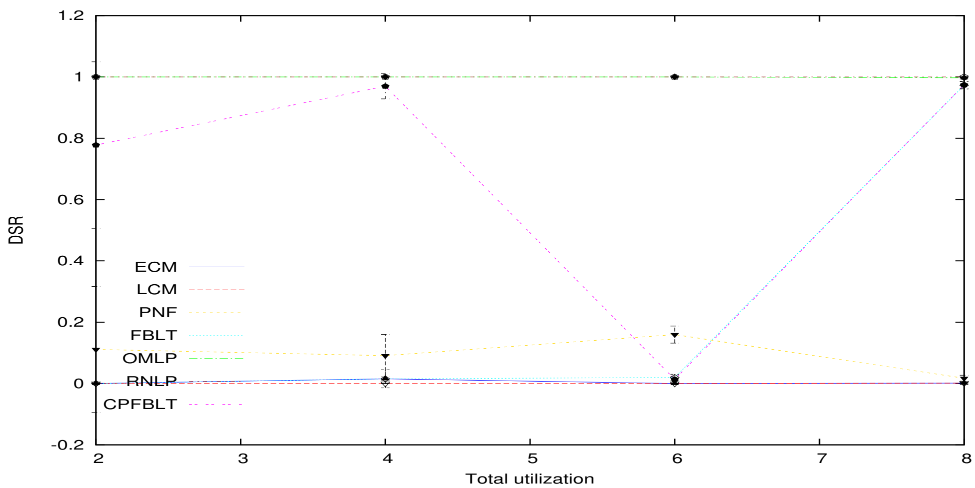


Figure B.267: DSR for Tasksets 267, 537, 807 and 1077

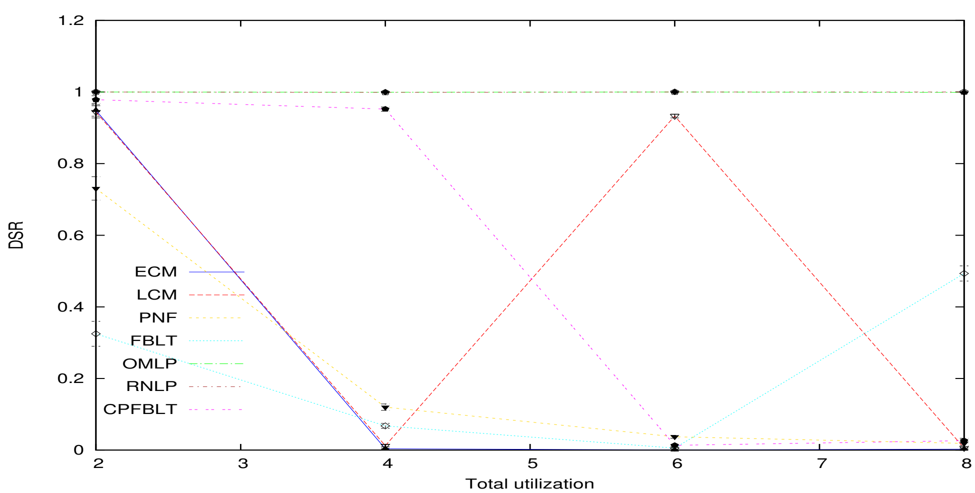


Figure B.268: DSR for Tasksets 268, 538, 808 and 1078

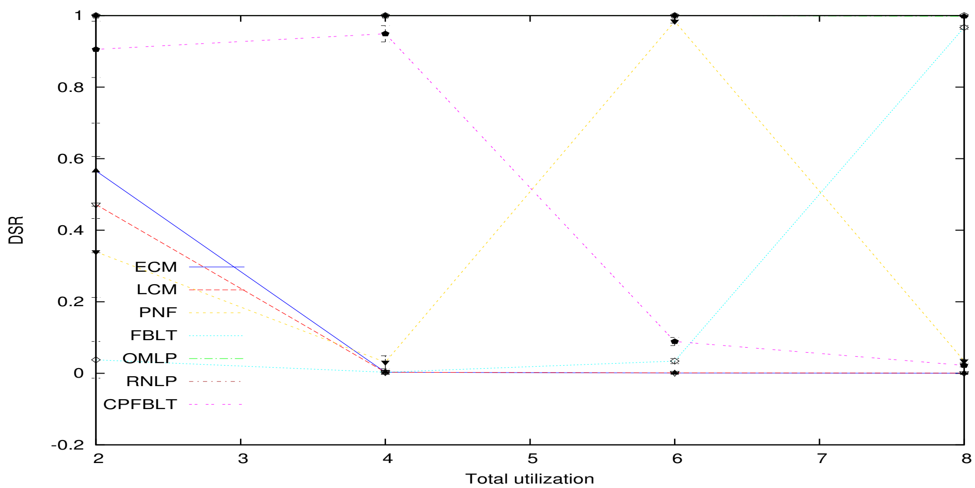


Figure B.269: DSR for Tasksets 269, 539, 809 and 1079

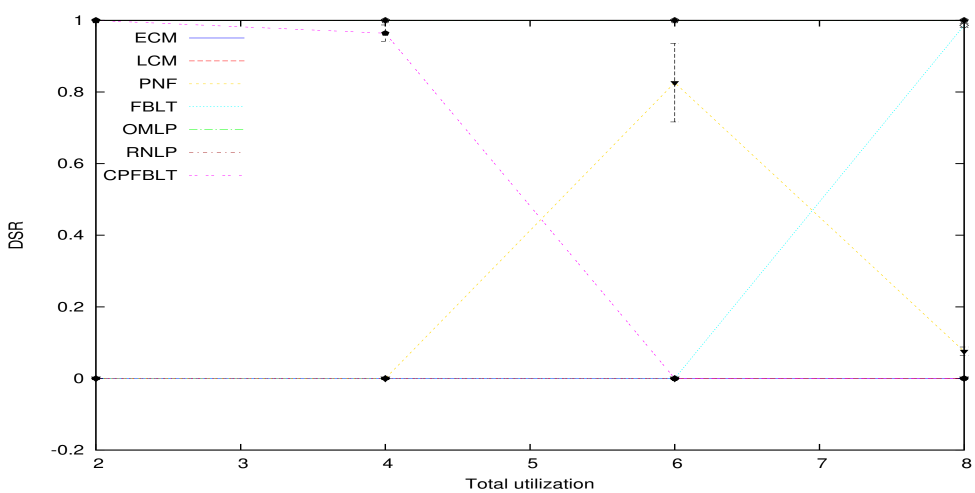


Figure B.270: DSR for Tasksets 270, 540, 810 and 1080

Appendix C

Complete Average Retry Cost Results

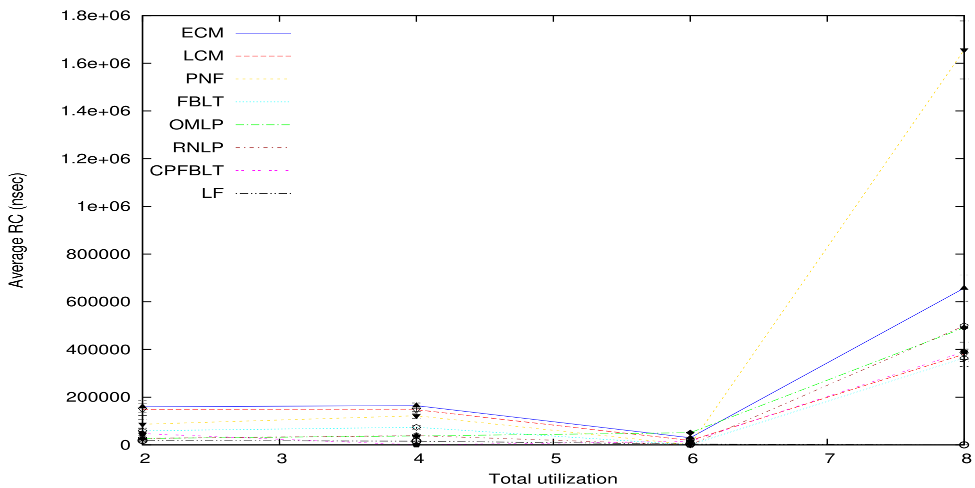


Figure C.1: Avg_RC for Tasksets 1, 271, 541 and 811

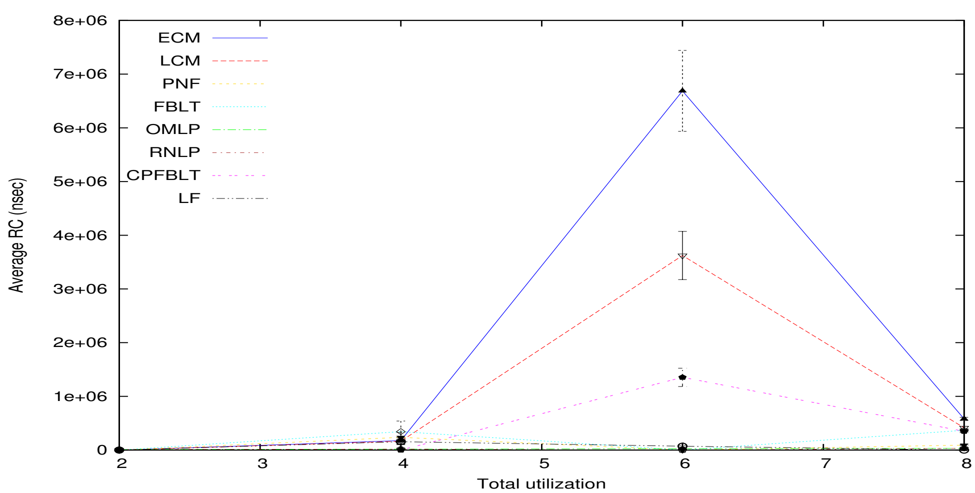


Figure C.2: Avg_RC for Tasksets 2, 272, 542 and 812

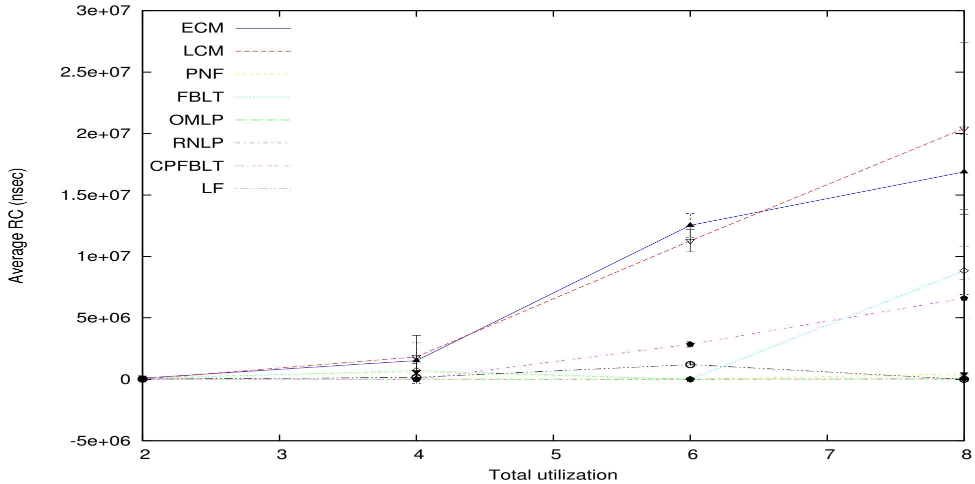


Figure C.3: Avg_RC for Tasksets 3, 273, 543 and 813

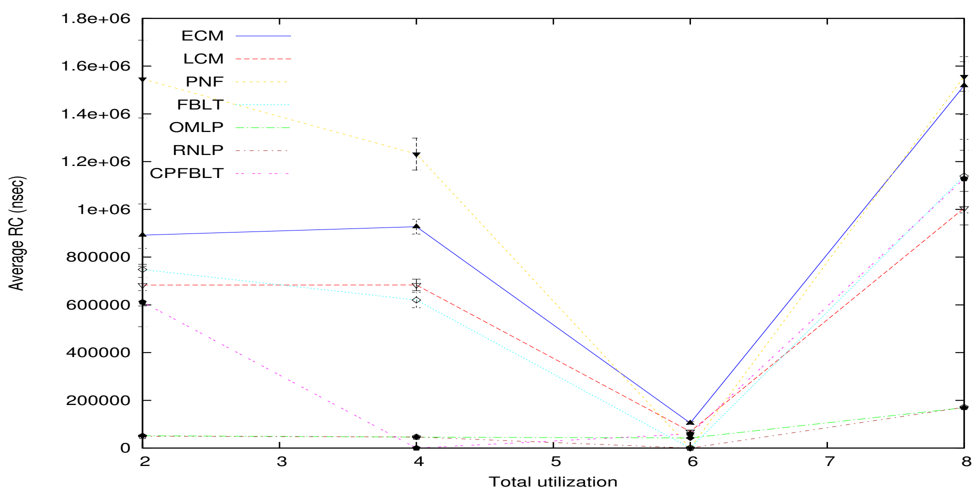


Figure C.4: Avg_RC for Tasksets 4, 274, 544 and 814

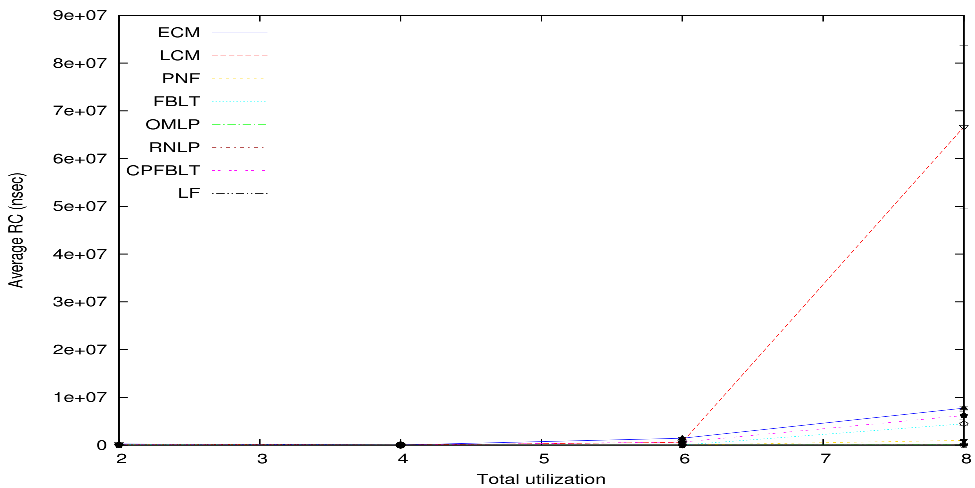


Figure C.5: Avg_RC for Tasksets 5, 275, 545 and 815

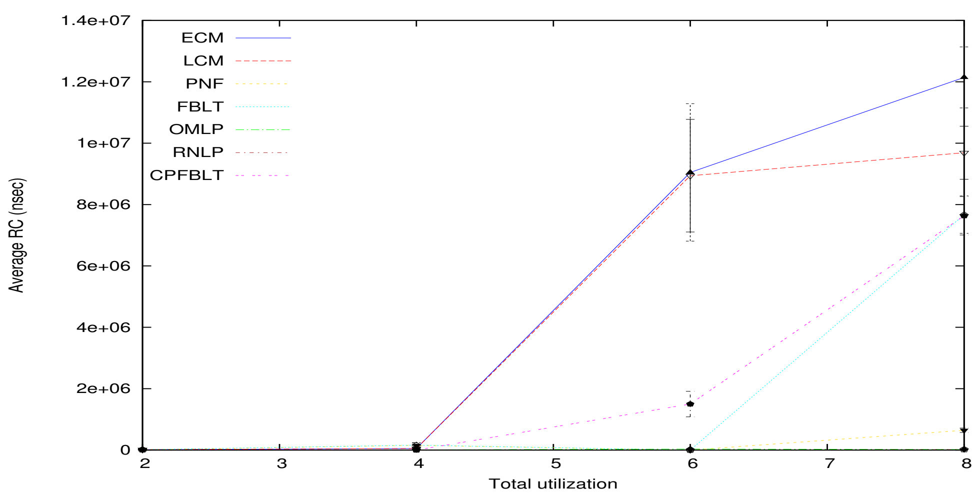


Figure C.6: Avg_RC for Tasksets 6, 276, 546 and 816

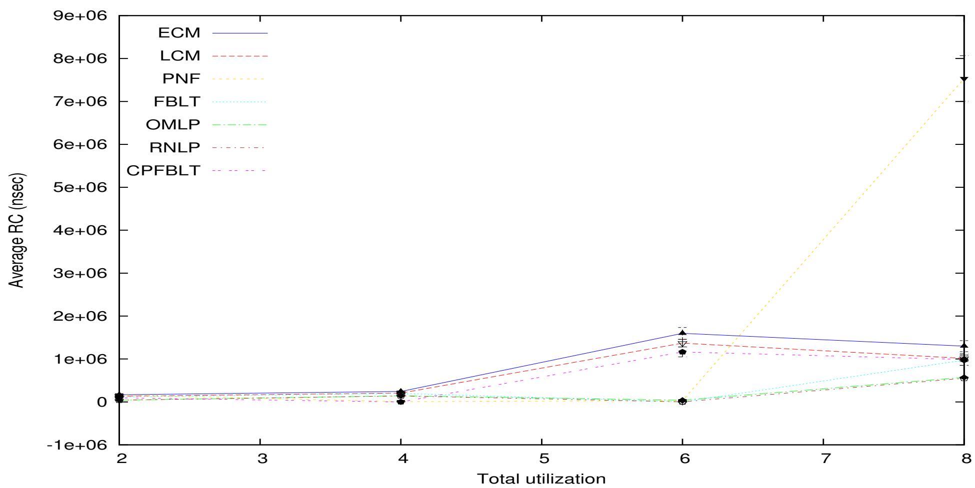


Figure C.7: Avg_RC for Tasksets 7, 277, 547 and 817

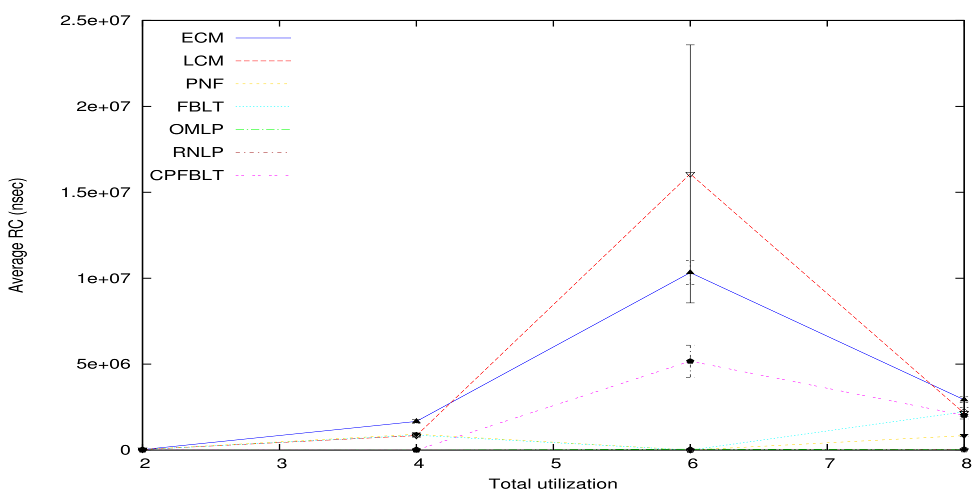


Figure C.8: Avg_RC for Tasksets 8, 278, 548 and 818

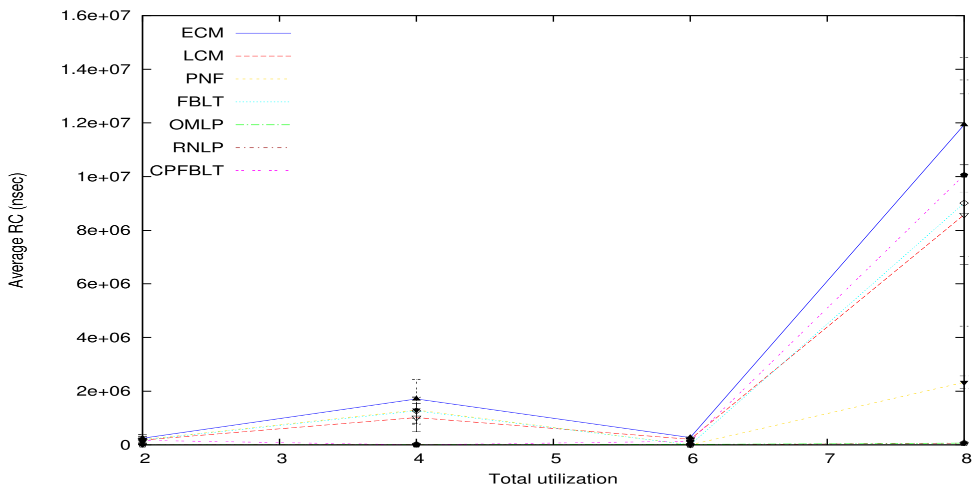


Figure C.9: Avg_RC for Tasksets 9, 279, 549 and 819

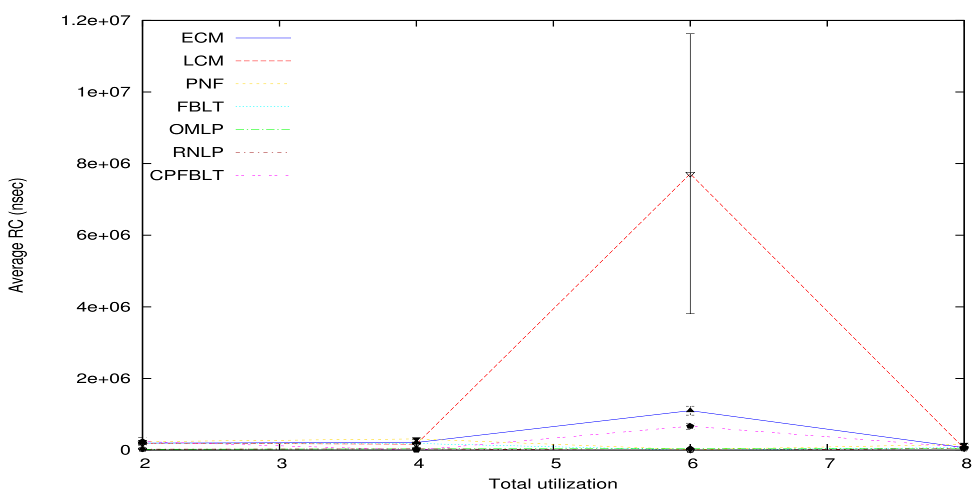


Figure C.10: Avg_RC for Tasksets 10, 280, 550 and 820

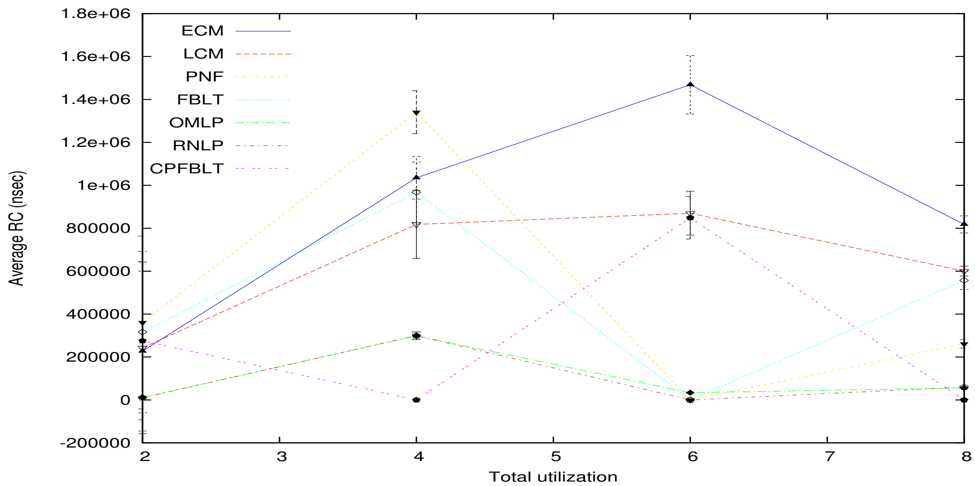


Figure C.11: Avg_RC for Tasksets 11, 281, 551 and 821

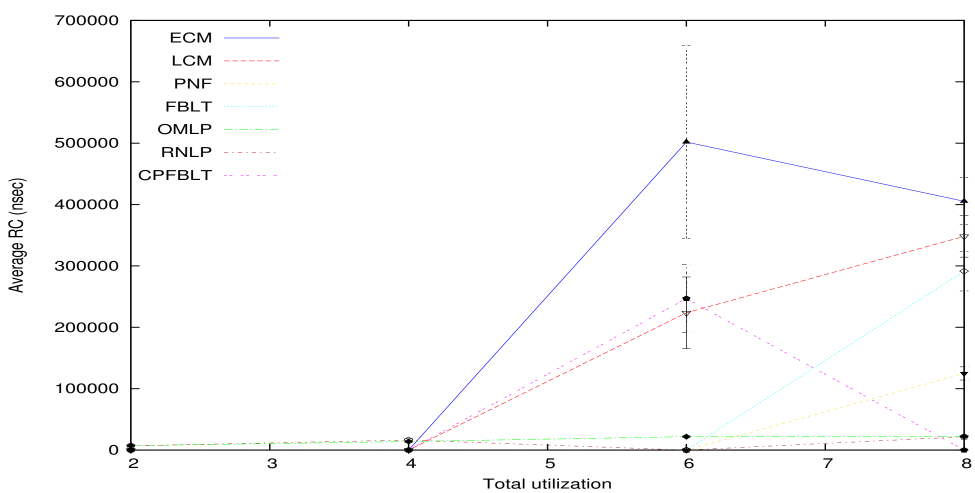


Figure C.12: Avg_RC for Tasksets 12, 282, 552 and 822

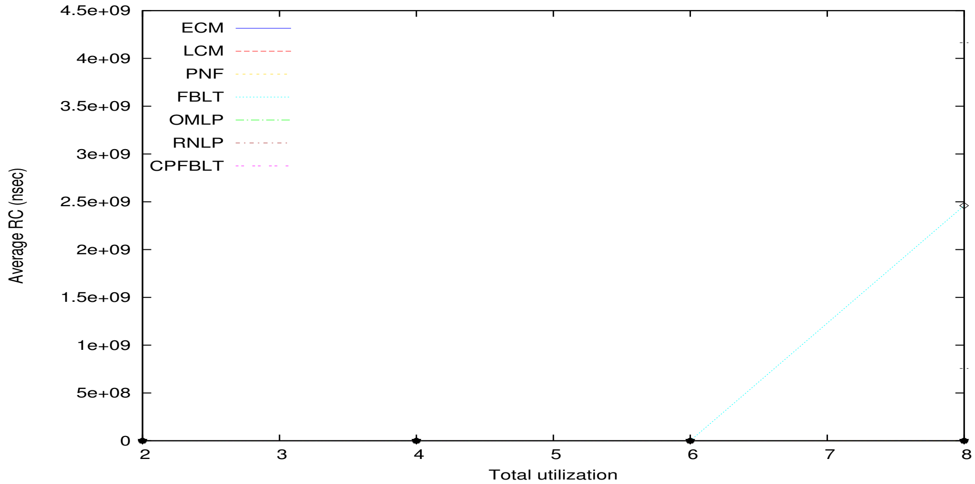


Figure C.13: Avg_RC for Tasksets 13, 283, 553 and 823

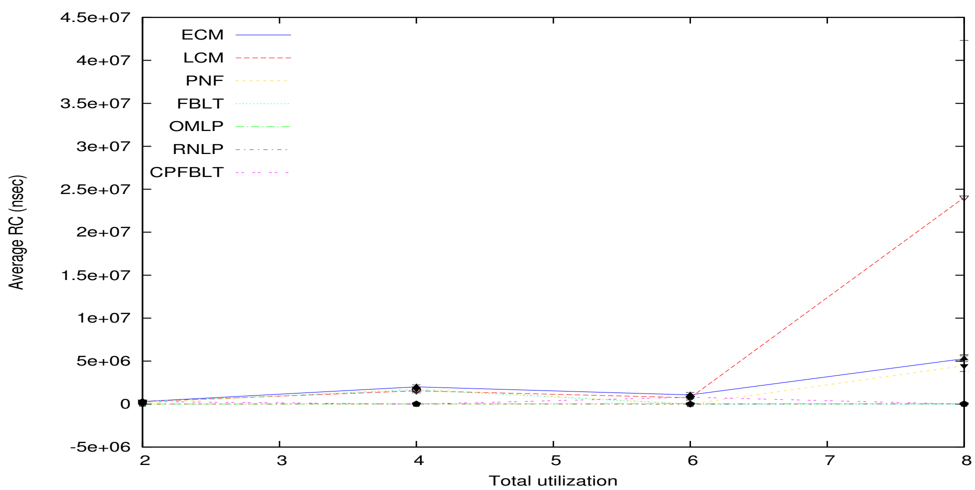


Figure C.14: Avg_RC for Tasksets 14, 284, 554 and 824

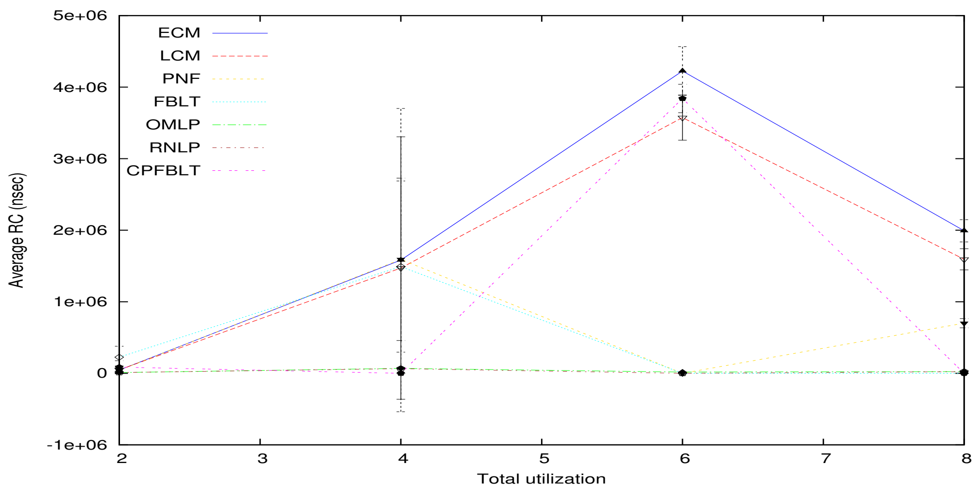


Figure C.15: Avg_RC for Tasksets 15, 285, 555 and 825

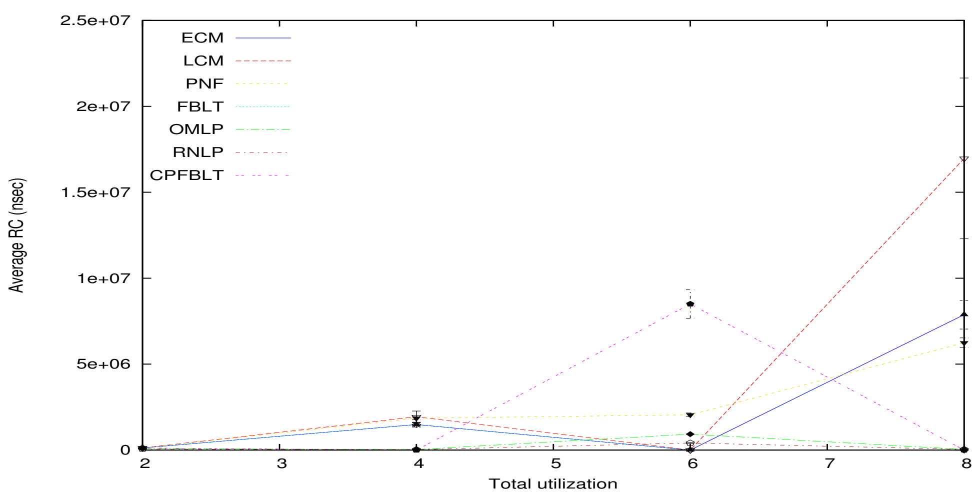


Figure C.16: Avg_RC for Tasksets 16, 286, 556 and 826

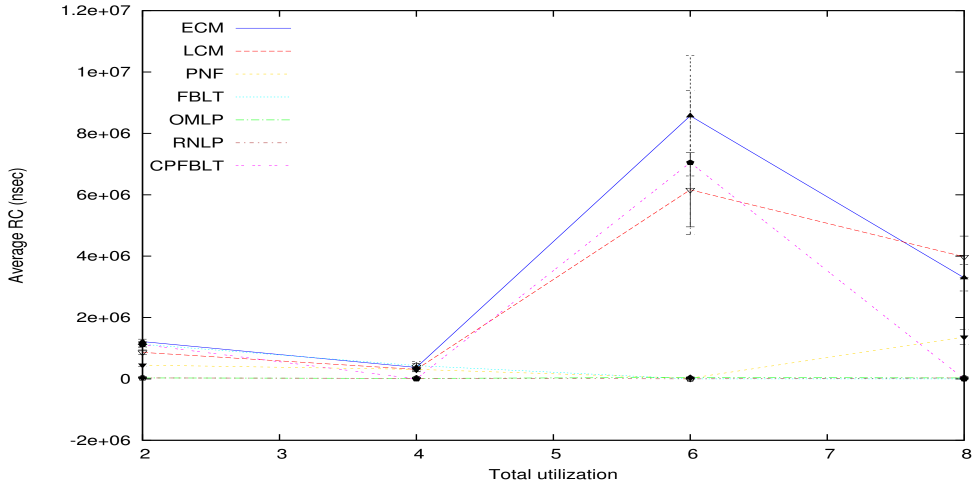


Figure C.17: Avg_RC for Tasksets 17, 287, 557 and 827

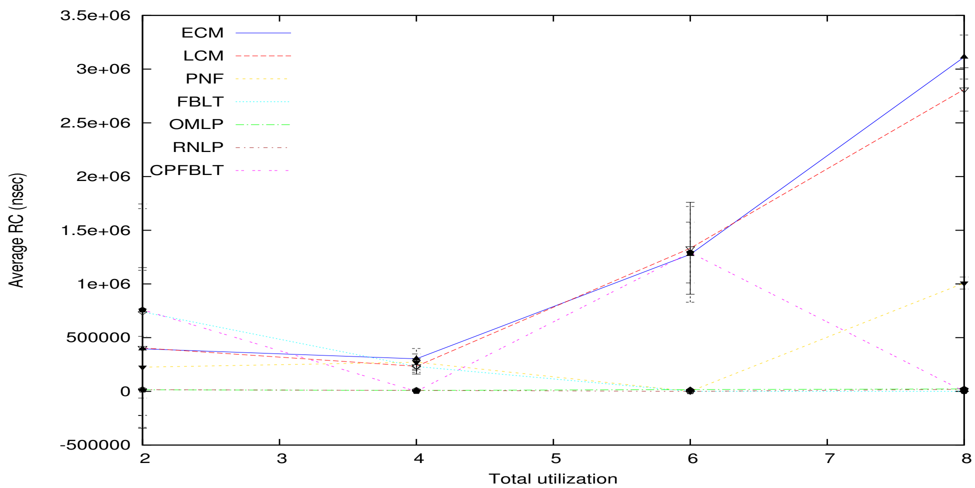


Figure C.18: Avg_RC for Tasksets 18, 288, 558 and 828

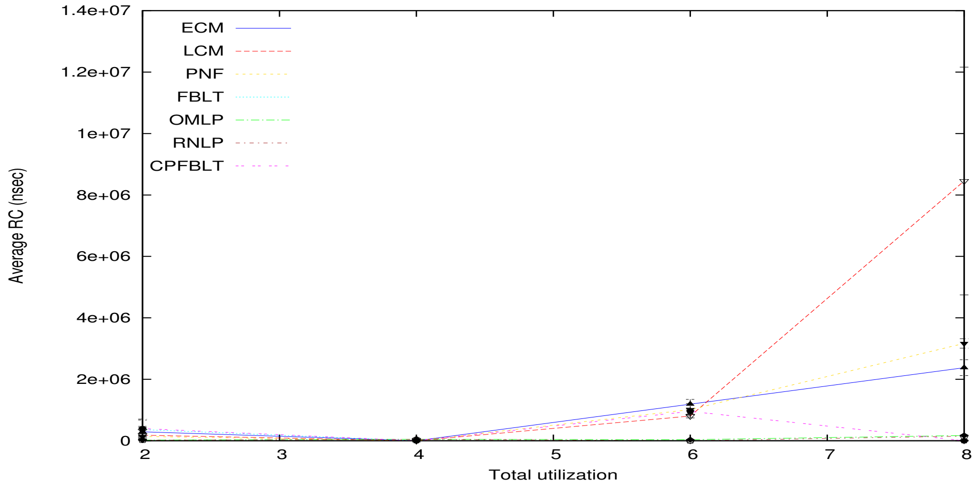


Figure C.19: Avg_RC for Tasksets 19, 289, 559 and 829

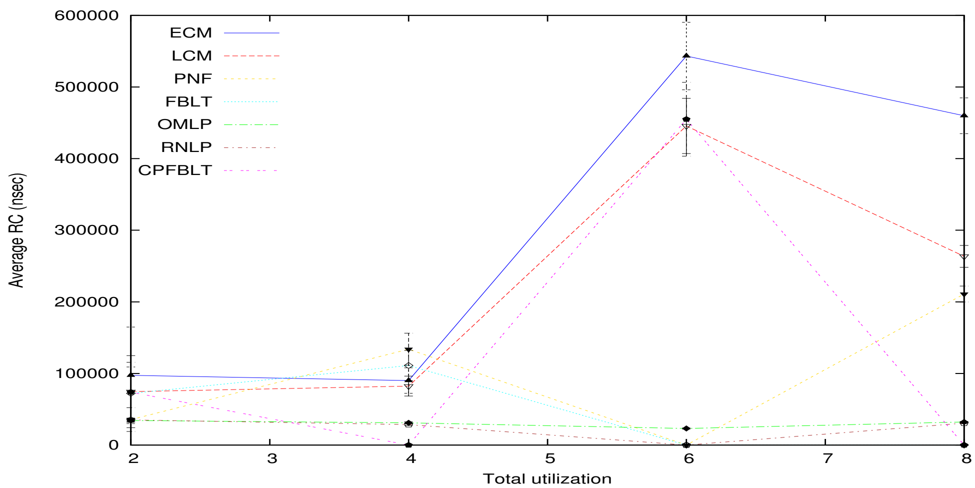


Figure C.20: Avg_RC for Tasksets 20, 290, 560 and 830

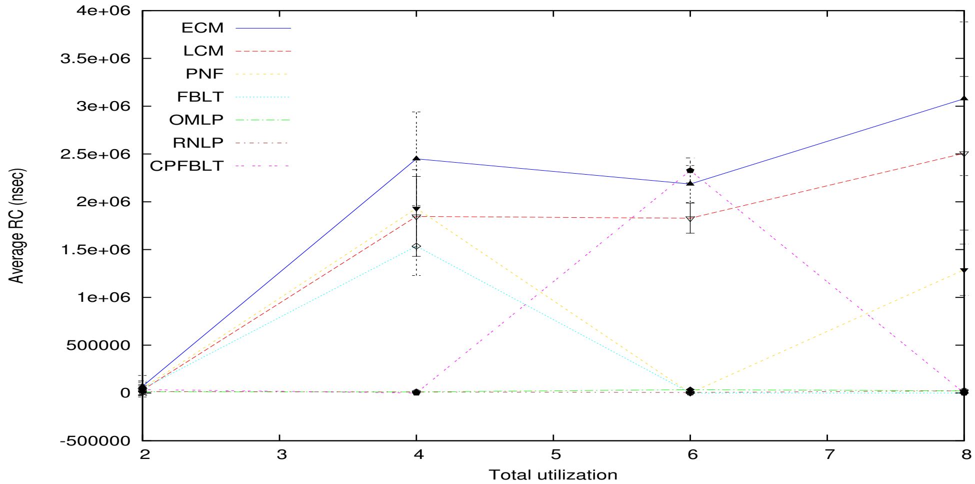


Figure C.21: Avg_RC for Tasksets 21, 291, 561 and 831

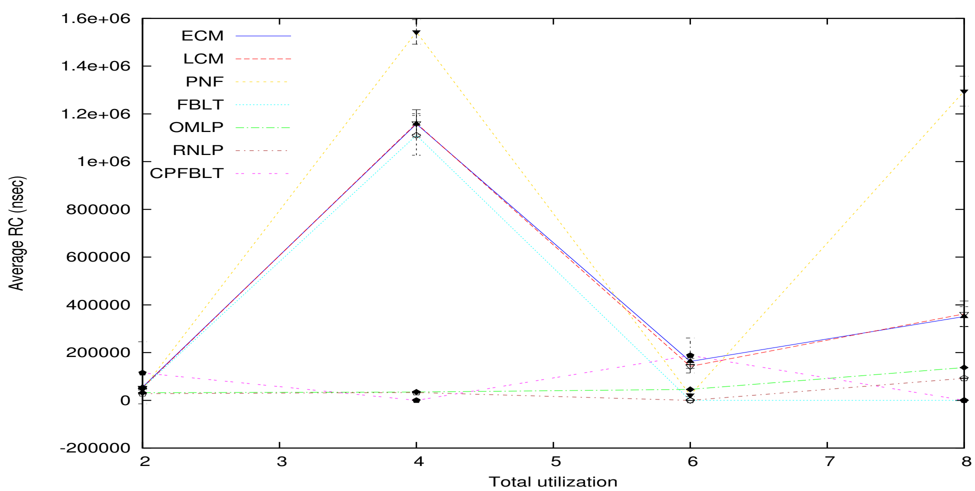


Figure C.22: Avg_RC for Tasksets 22, 292, 562 and 832

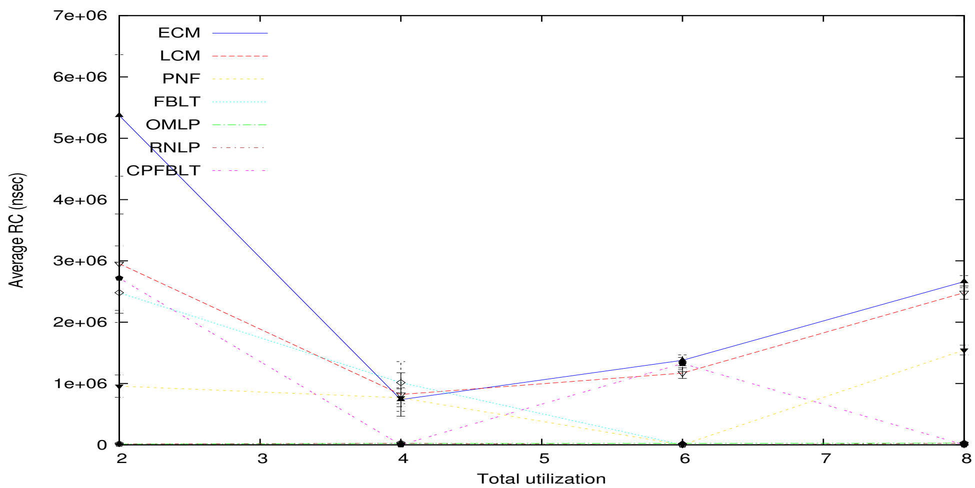


Figure C.23: Avg_RC for Tasksets 23, 293, 563 and 833

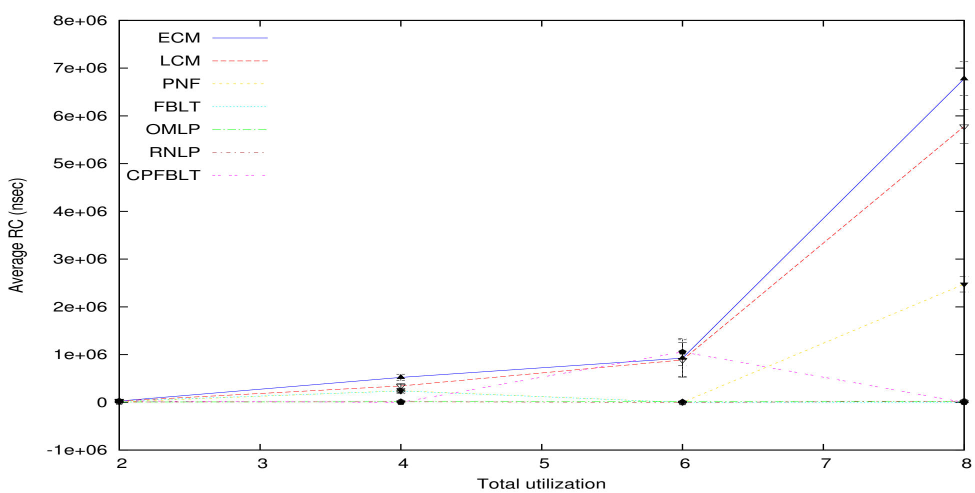


Figure C.24: Avg_RC for Tasksets 24, 294, 564 and 834

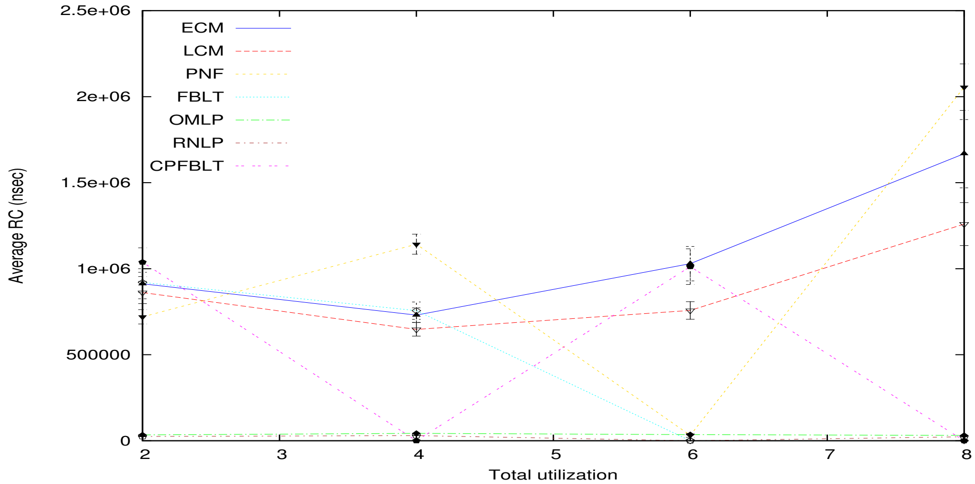


Figure C.25: Avg_RC for Tasksets 25, 295, 565 and 835

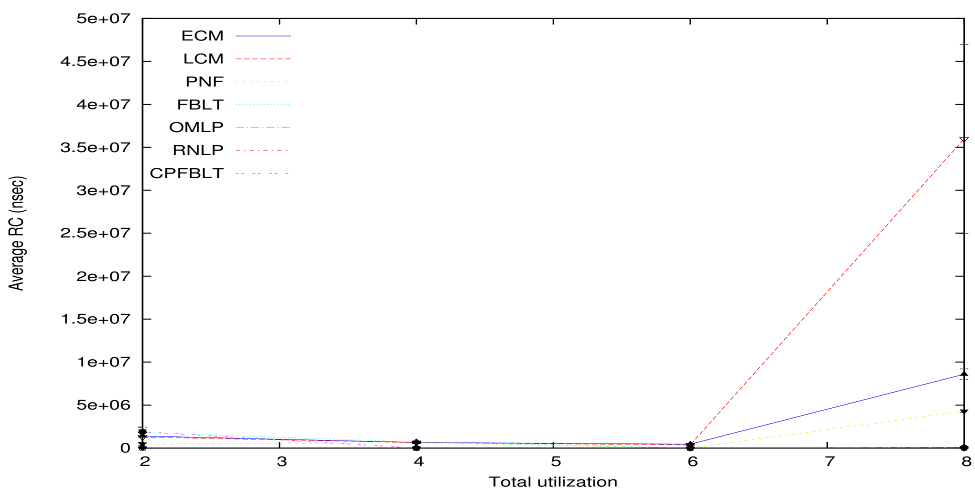


Figure C.26: Avg_RC for Tasksets 26, 296, 566 and 836

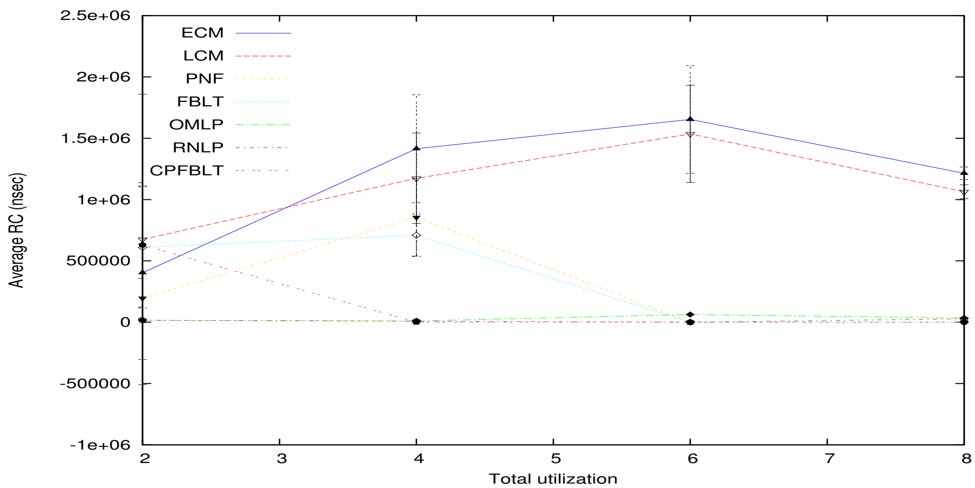


Figure C.27: Avg_RC for Tasksets 27, 297, 567 and 837

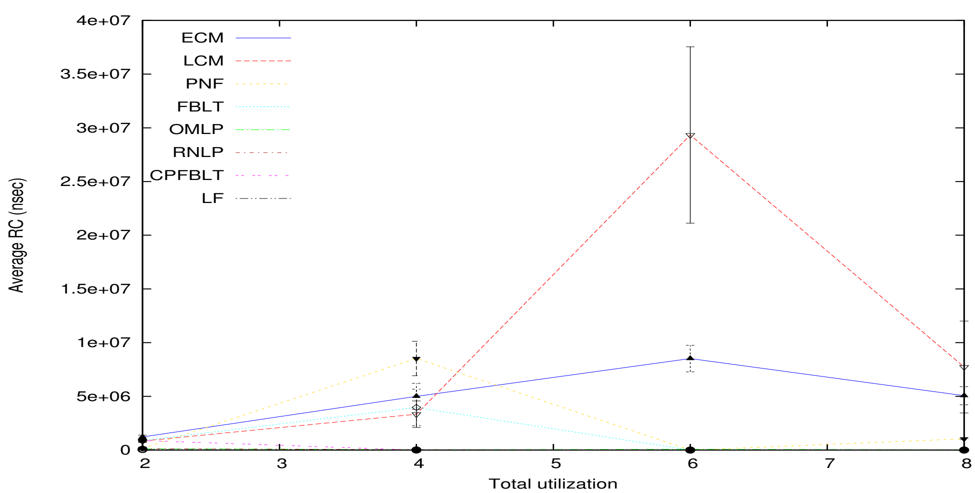


Figure C.28: Avg_RC for Tasksets 28, 298, 568 and 838

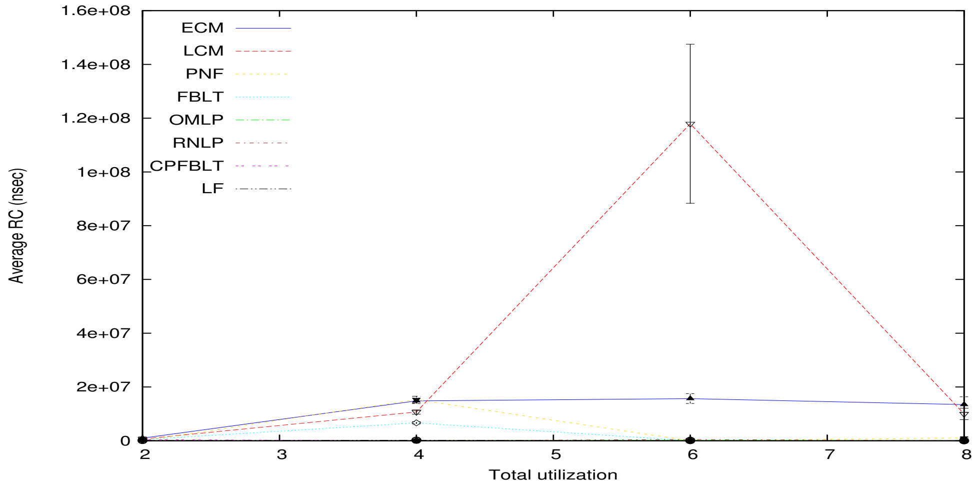


Figure C.29: Avg_RC for Tasksets 29, 299, 569 and 839

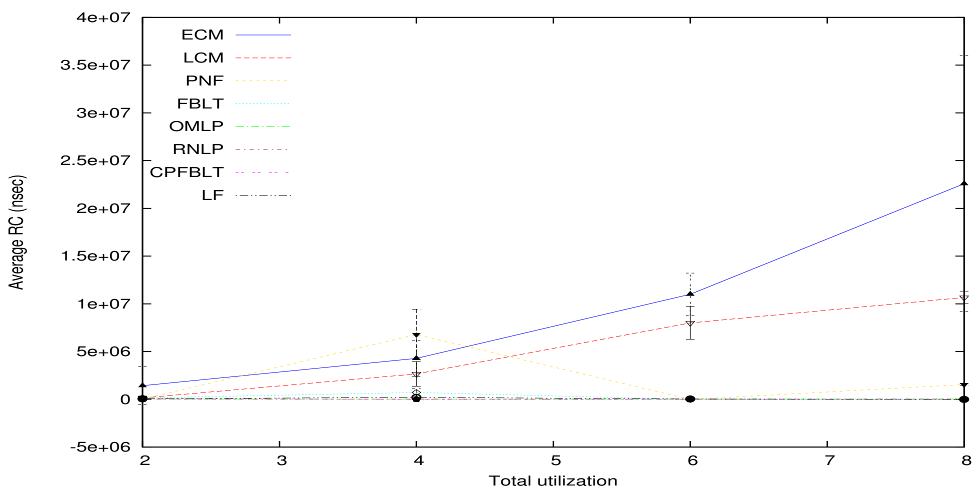


Figure C.30: Avg_RC for Tasksets 30, 300, 570 and 840

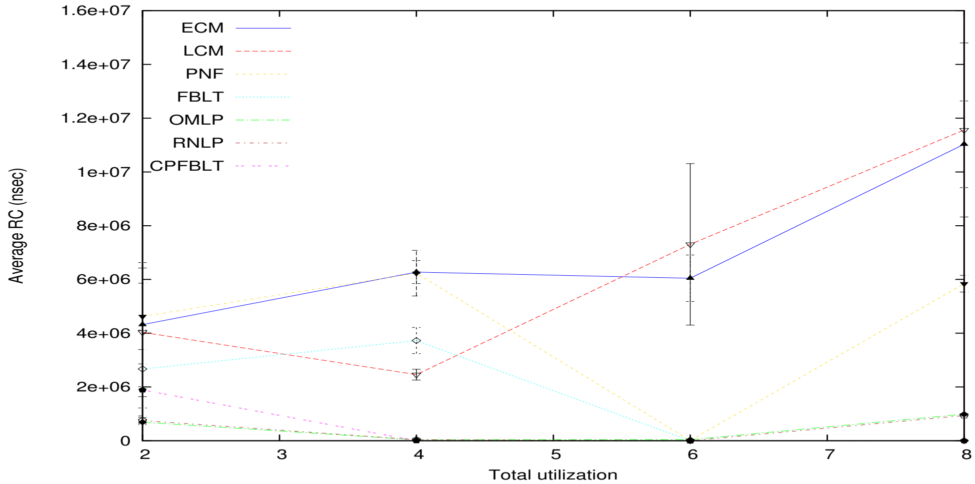


Figure C.31: Avg_RC for Tasksets 31, 301, 571 and 841

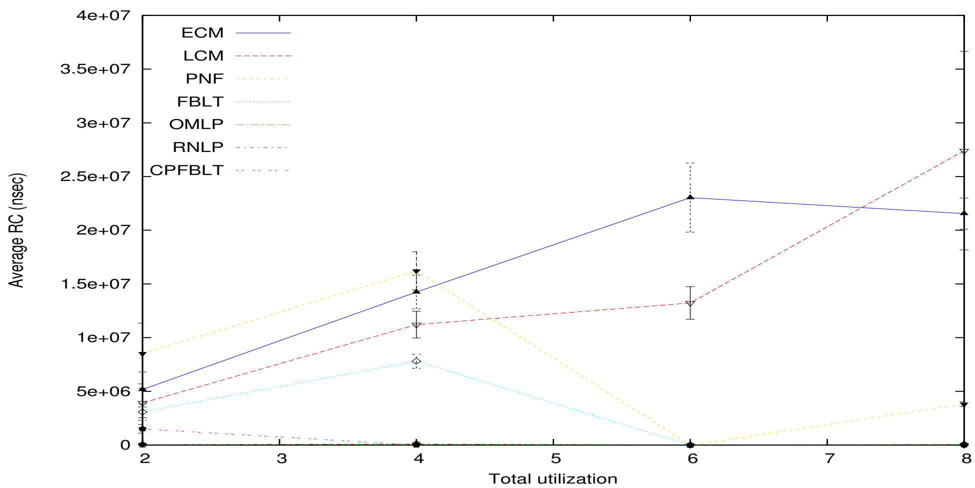


Figure C.32: Avg_RC for Tasksets 32, 302, 572 and 842

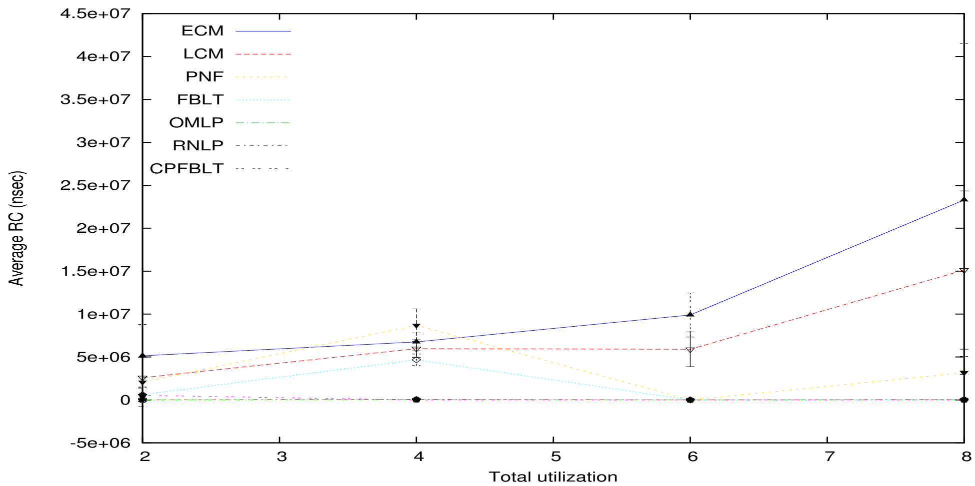


Figure C.33: Avg_RC for Tasksets 33, 303, 573 and 843

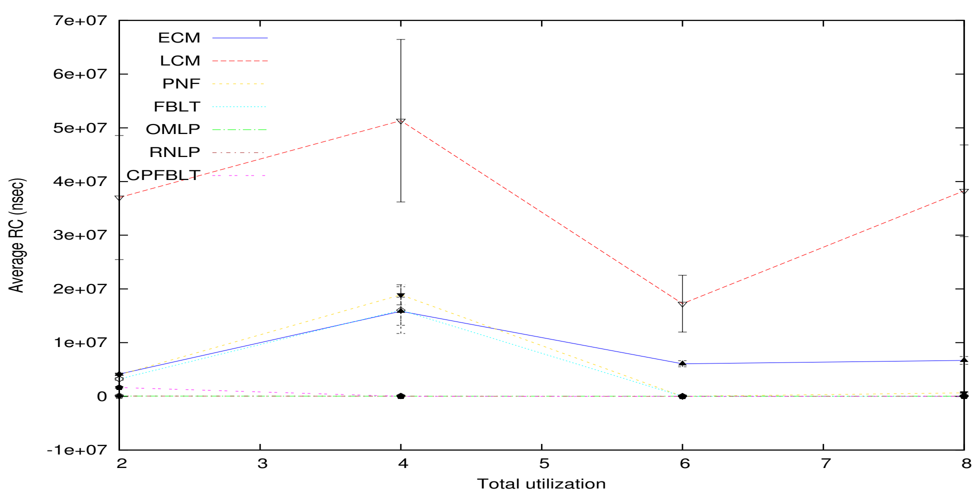


Figure C.34: Avg_RC for Tasksets 34, 304, 574 and 844

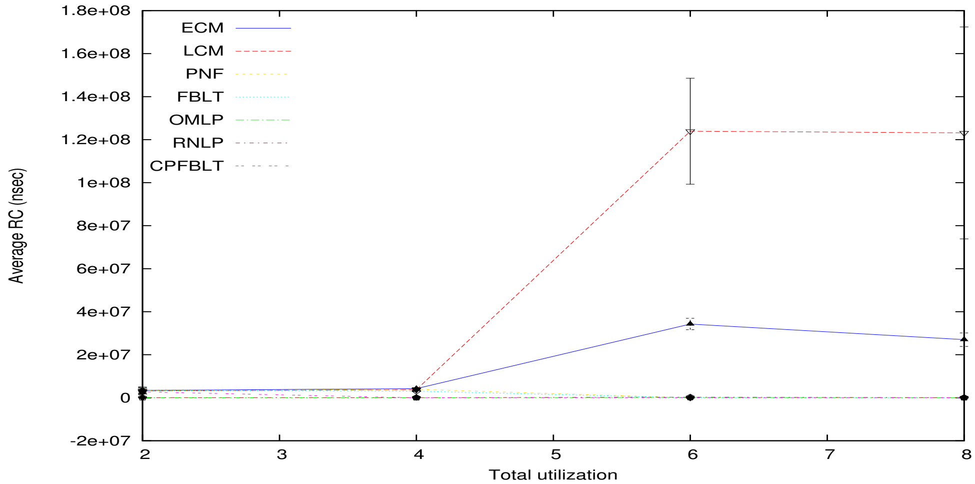


Figure C.35: Avg_RC for Tasksets 35, 305, 575 and 845

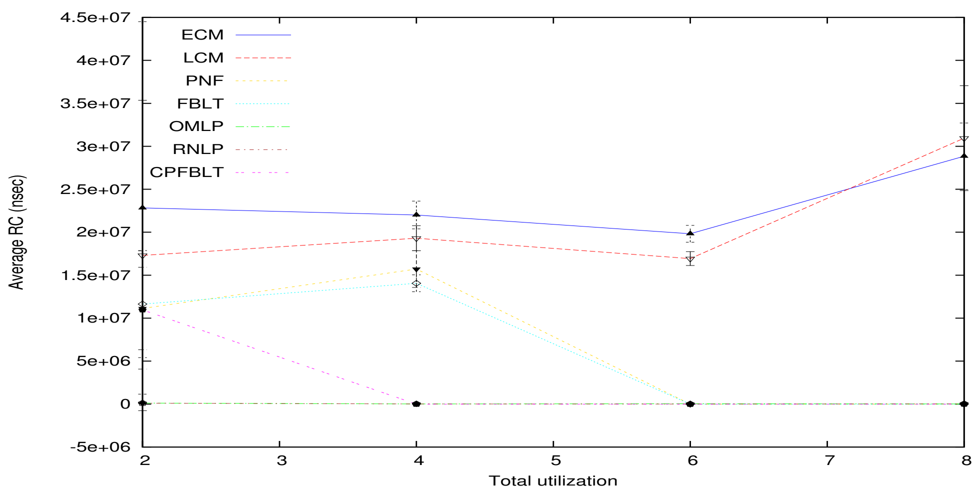


Figure C.36: Avg_RC for Tasksets 36, 306, 576 and 846

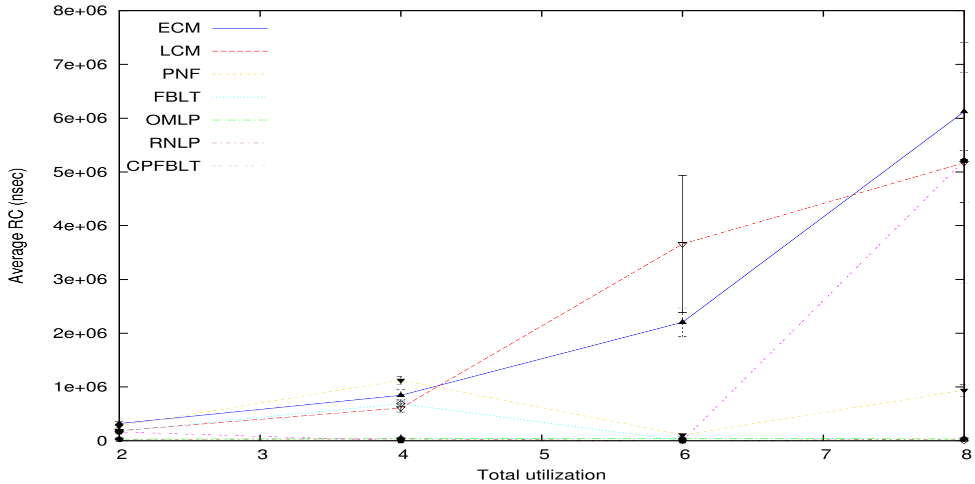


Figure C.37: Avg_RC for Tasksets 37, 307, 577 and 847

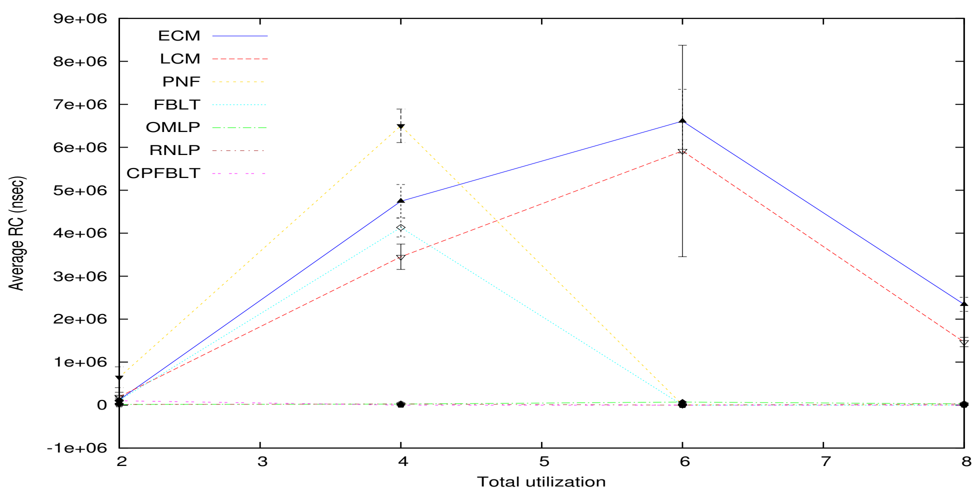


Figure C.38: Avg_RC for Tasksets 38, 308, 578 and 848

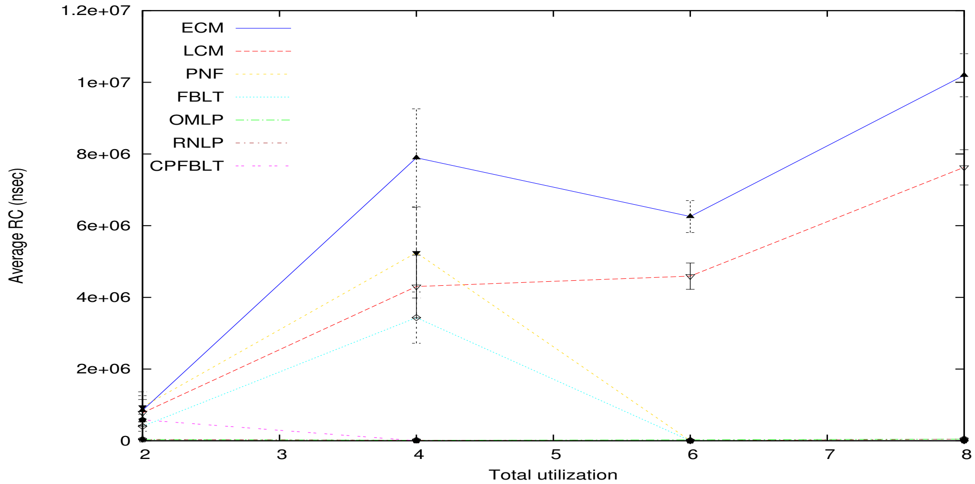


Figure C.39: Avg_RC for Tasksets 39, 309, 579 and 849

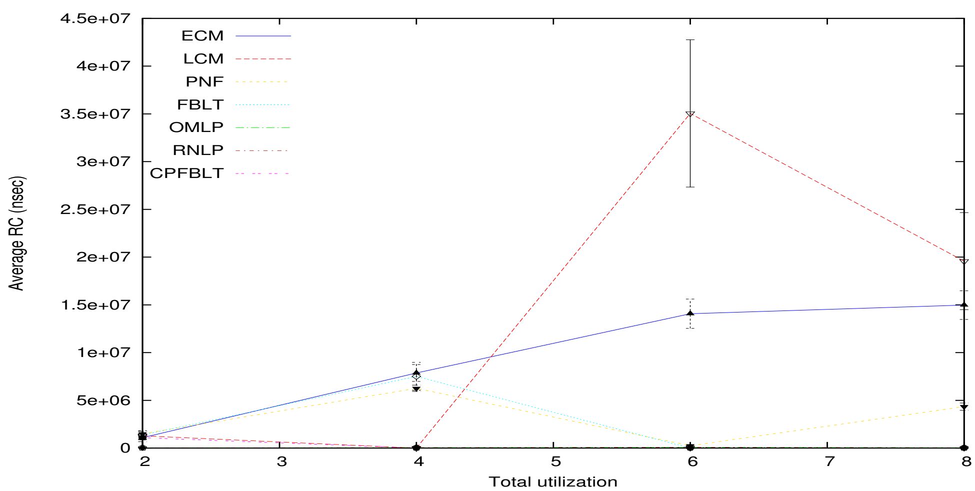


Figure C.40: Avg_RC for Tasksets 40, 310, 580 and 850

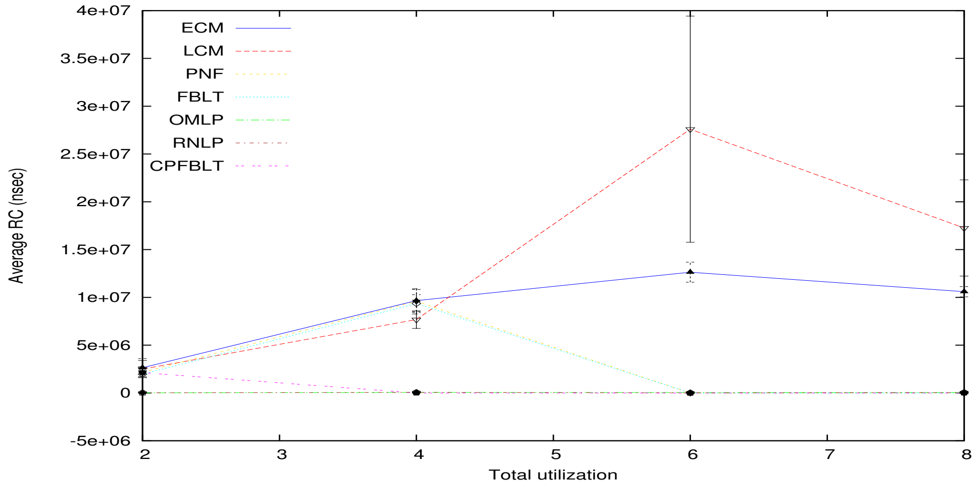


Figure C.41: Avg_RC for Tasksets 41, 311, 581 and 851

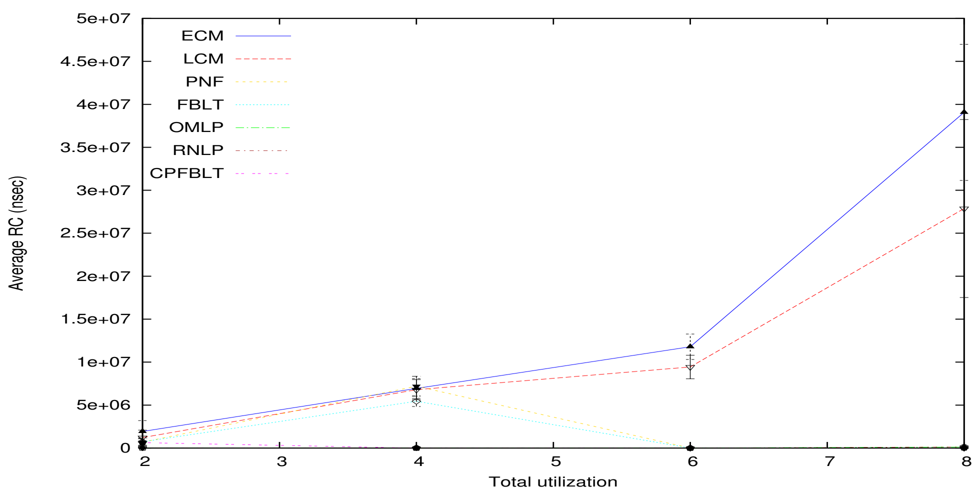


Figure C.42: Avg_RC for Tasksets 42, 312, 582 and 852

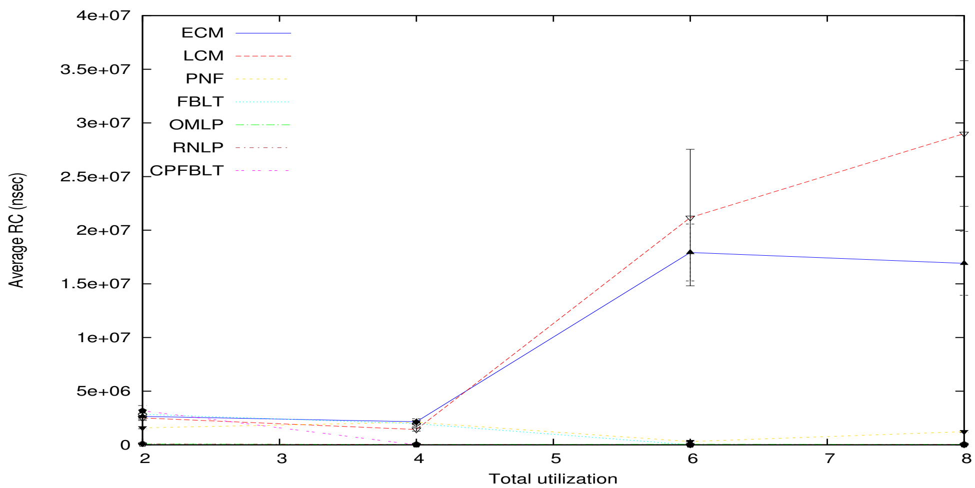


Figure C.43: Avg_RC for Tasksets 43, 313, 583 and 853

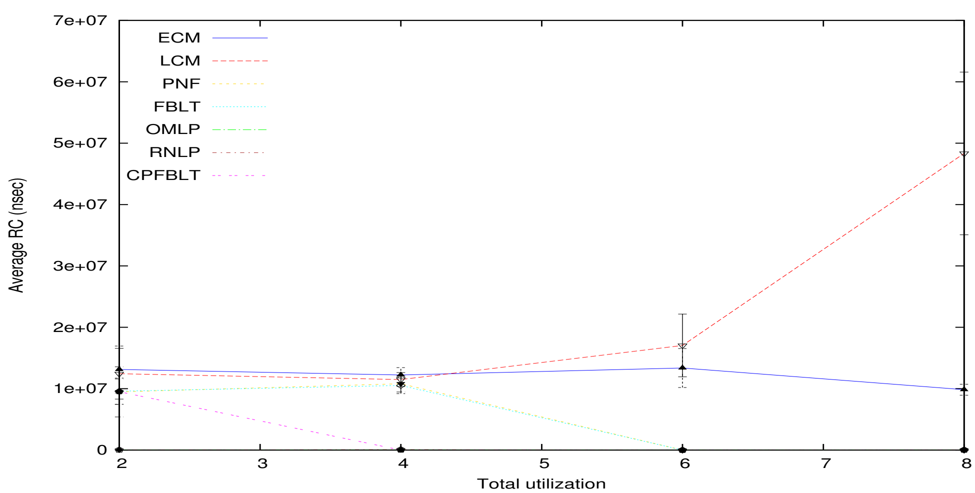


Figure C.44: Avg_RC for Tasksets 44, 314, 584 and 854

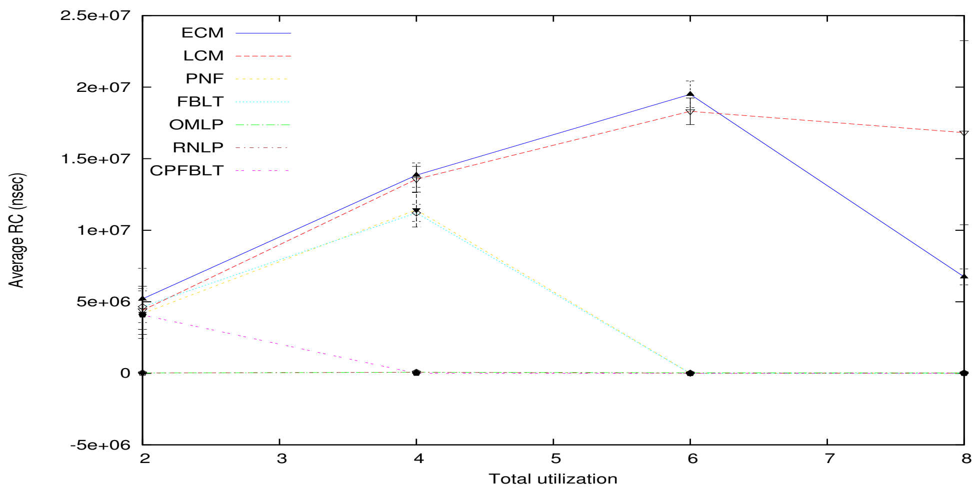


Figure C.45: Avg_RC for Tasksets 45, 315, 585 and 855

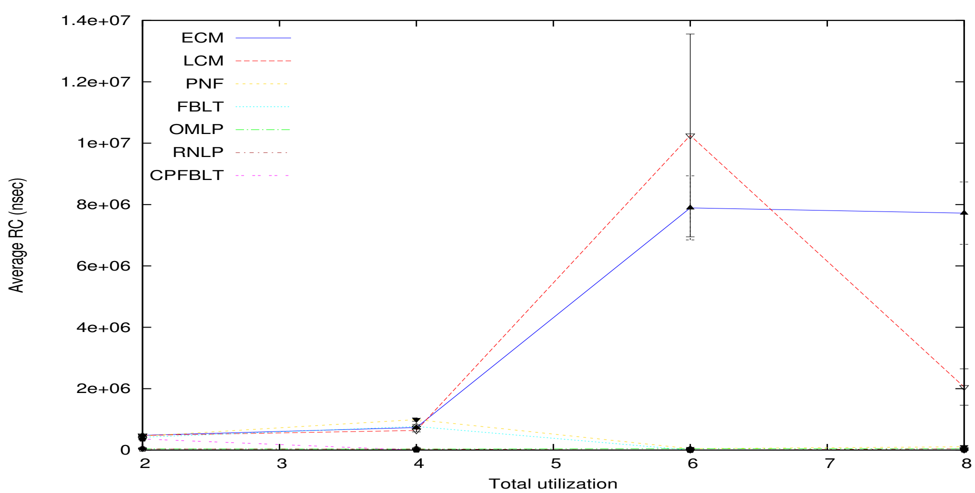


Figure C.46: Avg_RC for Tasksets 46, 316, 586 and 856

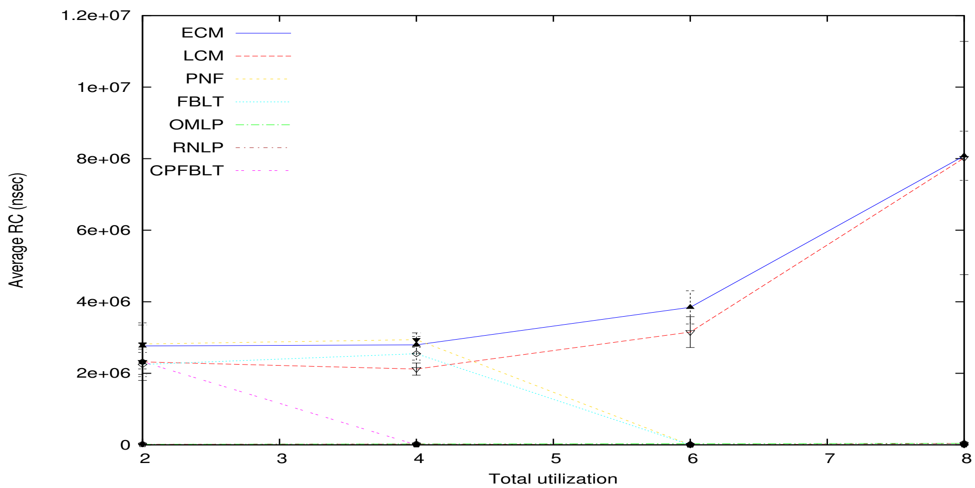


Figure C.47: Avg_RC for Tasksets 47, 317, 587 and 857

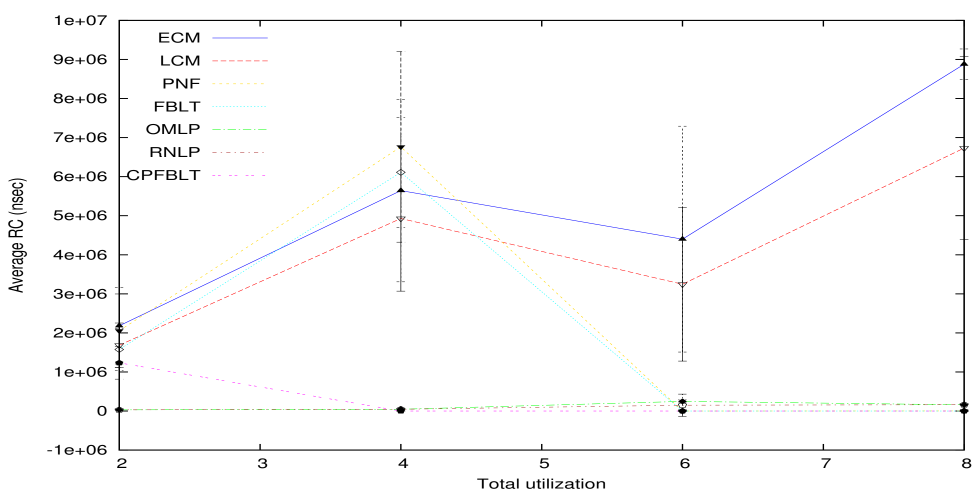


Figure C.48: Avg_RC for Tasksets 48, 318, 588 and 858

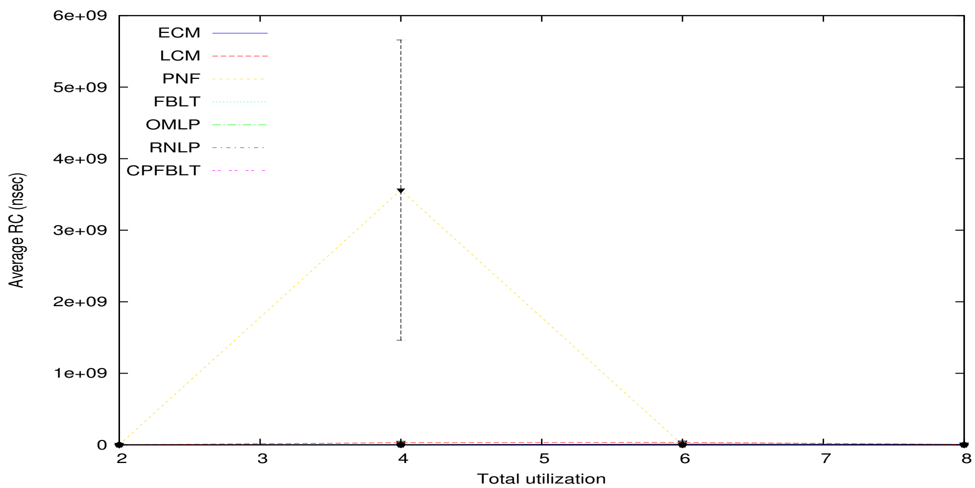


Figure C.49: Avg_RC for Tasksets 49, 319, 589 and 859

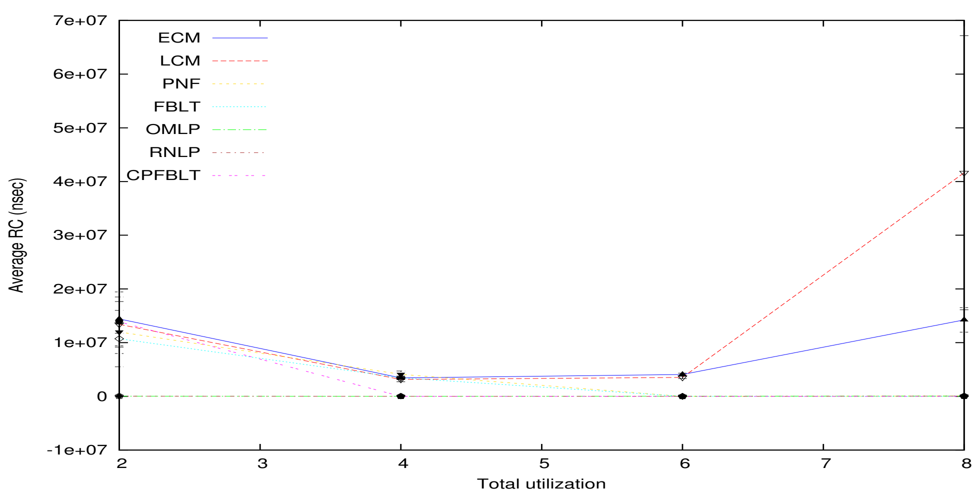


Figure C.50: Avg_RC for Tasksets 50, 320, 590 and 860

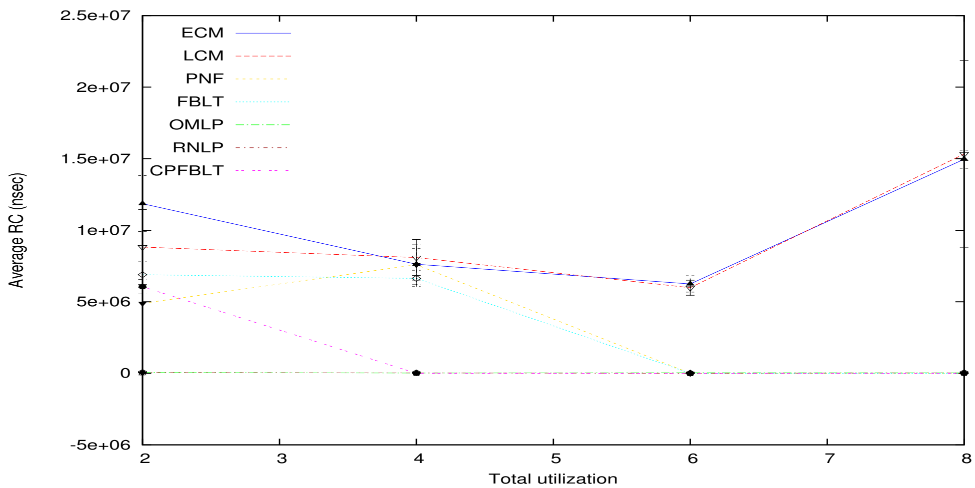


Figure C.51: Avg_RC for Tasksets 51, 321, 591 and 861

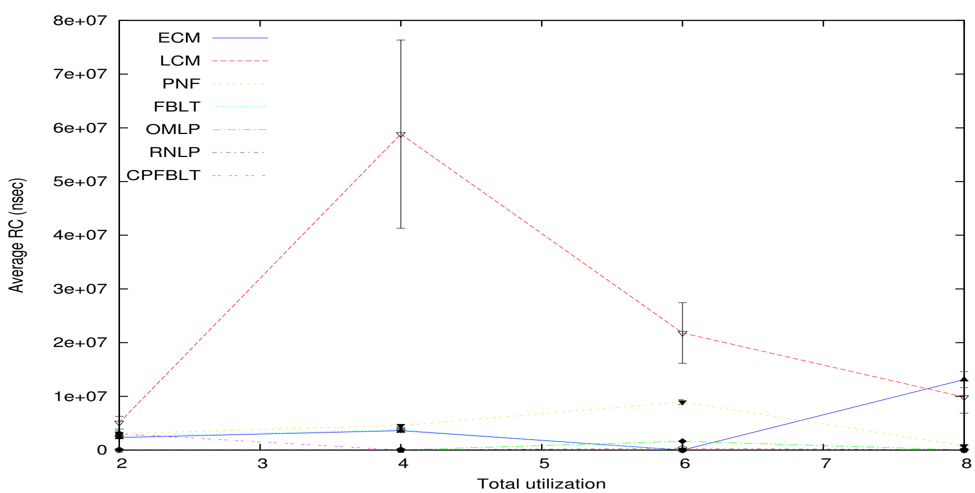


Figure C.52: Avg_RC for Tasksets 52, 322, 592 and 862

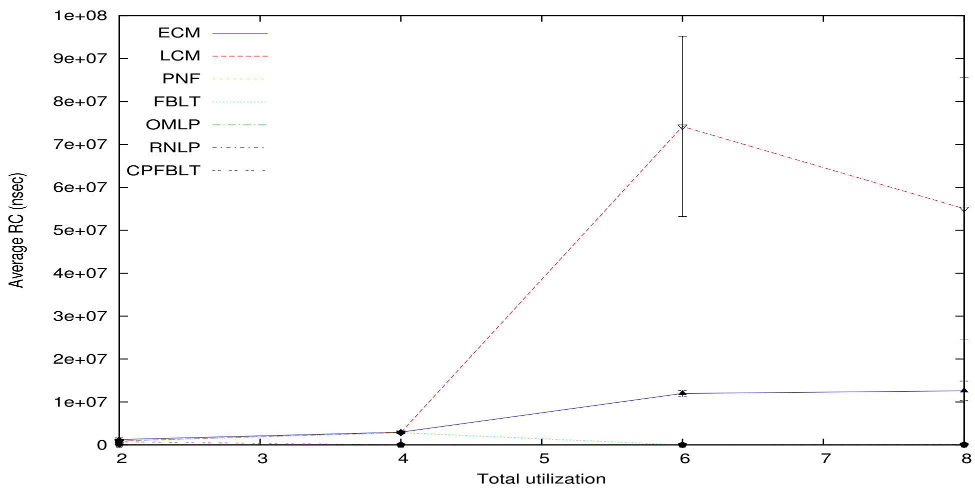


Figure C.53: Avg_RC for Tasksets 53, 323, 593 and 863

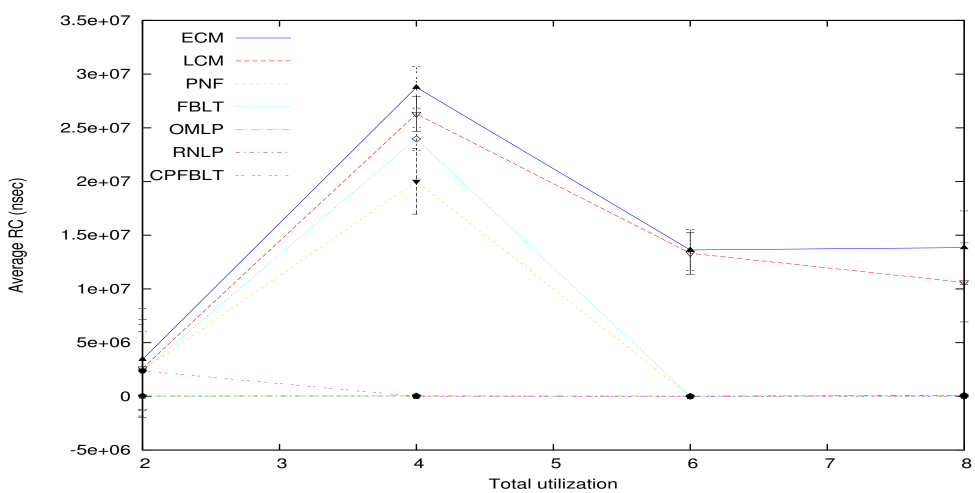


Figure C.54: Avg_RC for Tasksets 54, 324, 594 and 864

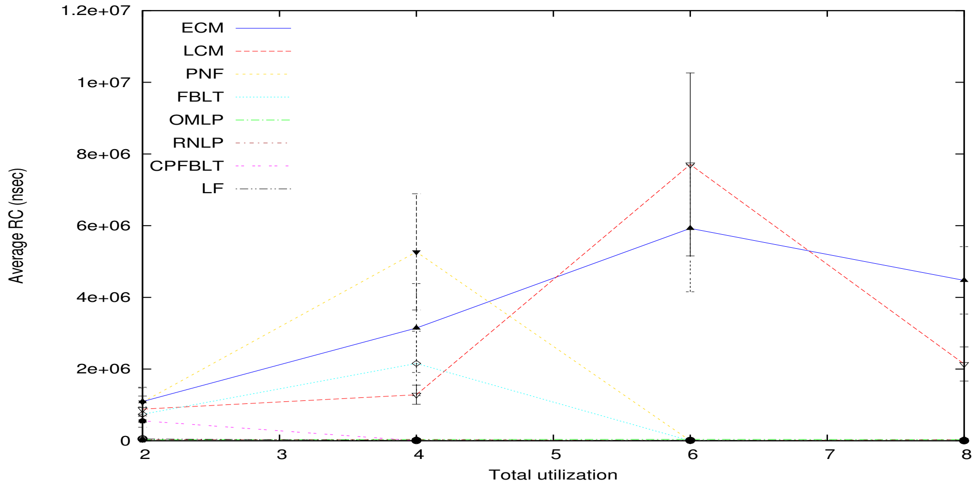


Figure C.55: Avg_RC for Tasksets 55, 325, 595 and 865

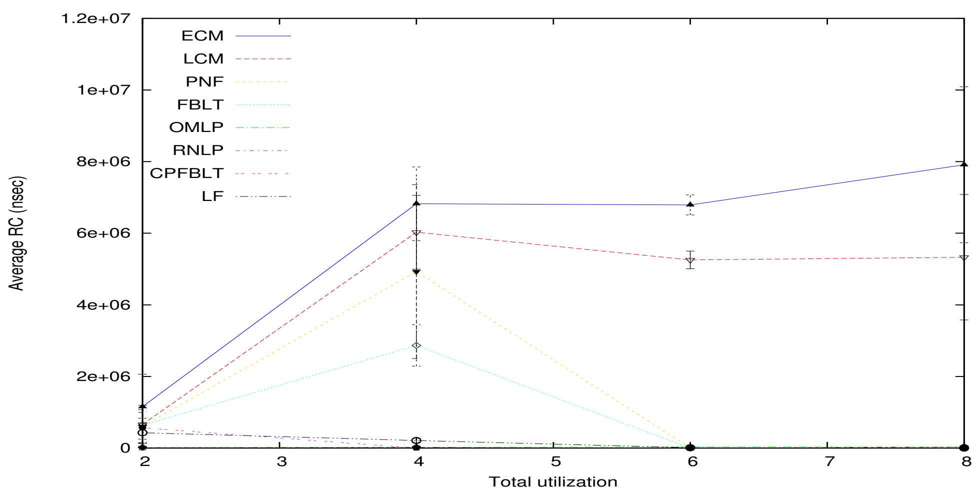


Figure C.56: Avg_RC for Tasksets 56, 326, 596 and 866

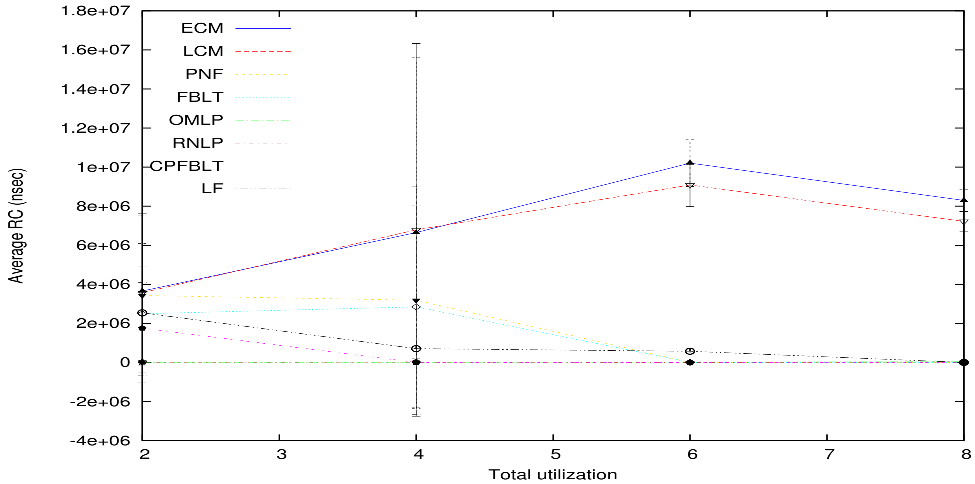


Figure C.57: Avg_RC for Tasksets 57, 327, 597 and 867

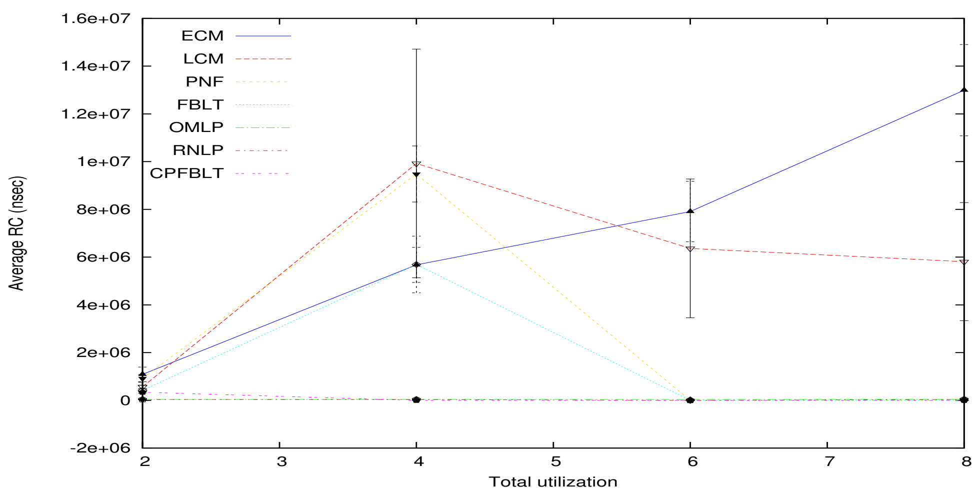


Figure C.58: Avg_RC for Tasksets 58, 328, 598 and 868

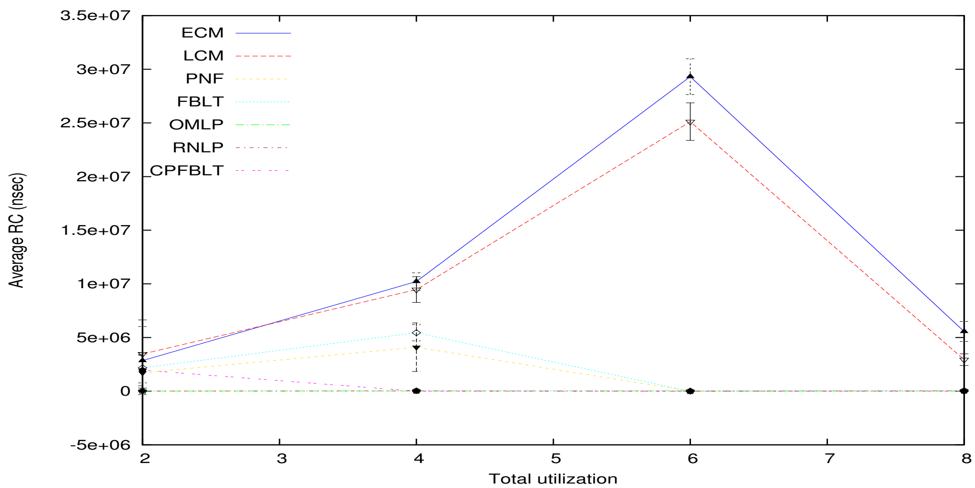


Figure C.59: Avg_RC for Tasksets 59, 329, 599 and 869

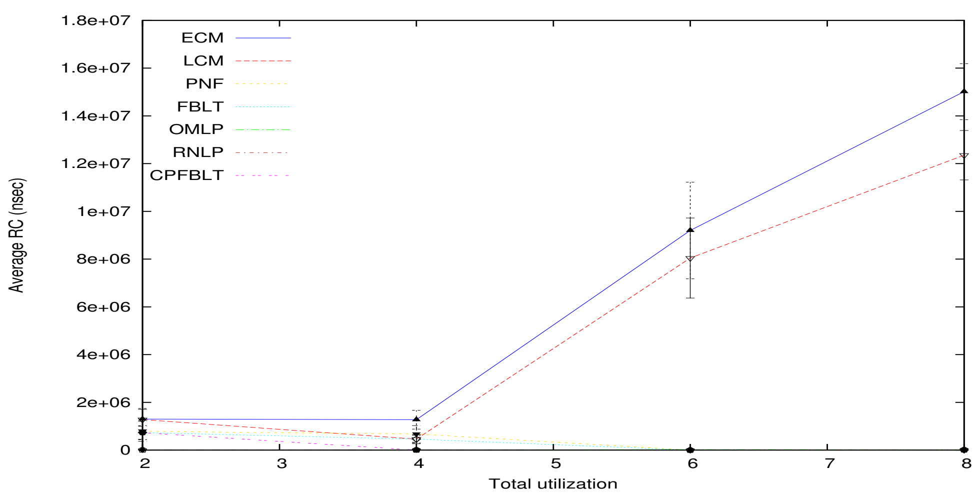


Figure C.60: Avg_RC for Tasksets 60, 330, 600 and 870

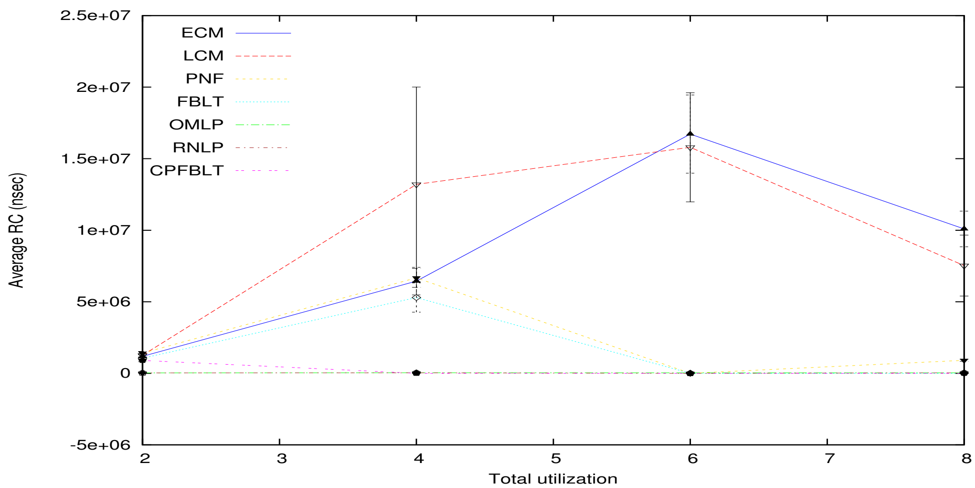


Figure C.61: Avg_RC for Tasksets 61, 331, 601 and 871

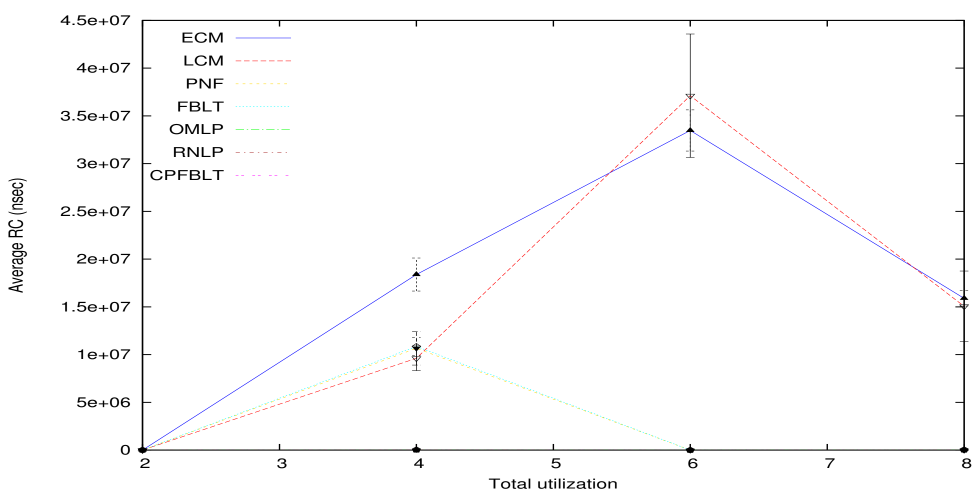


Figure C.62: Avg_RC for Tasksets 62, 332, 602 and 872

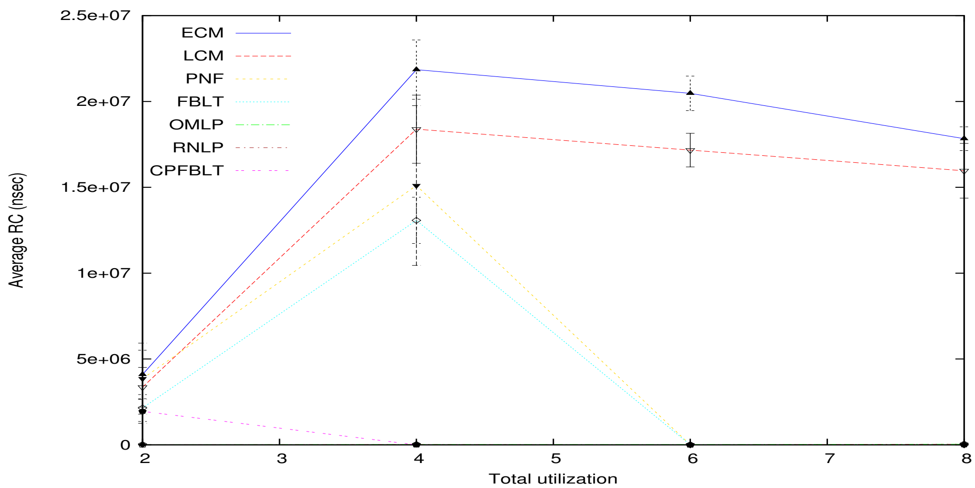


Figure C.63: Avg_RC for Tasksets 63, 333, 603 and 873

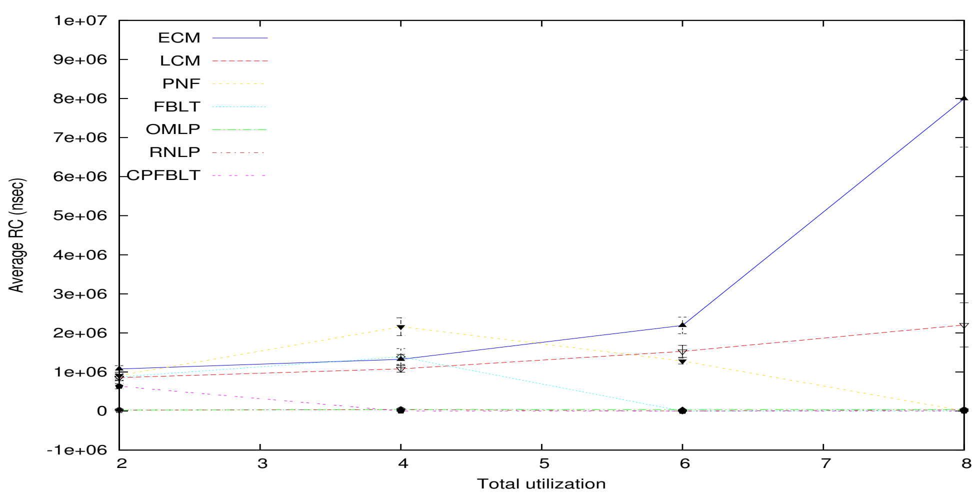


Figure C.64: Avg_RC for Tasksets 64, 334, 604 and 874

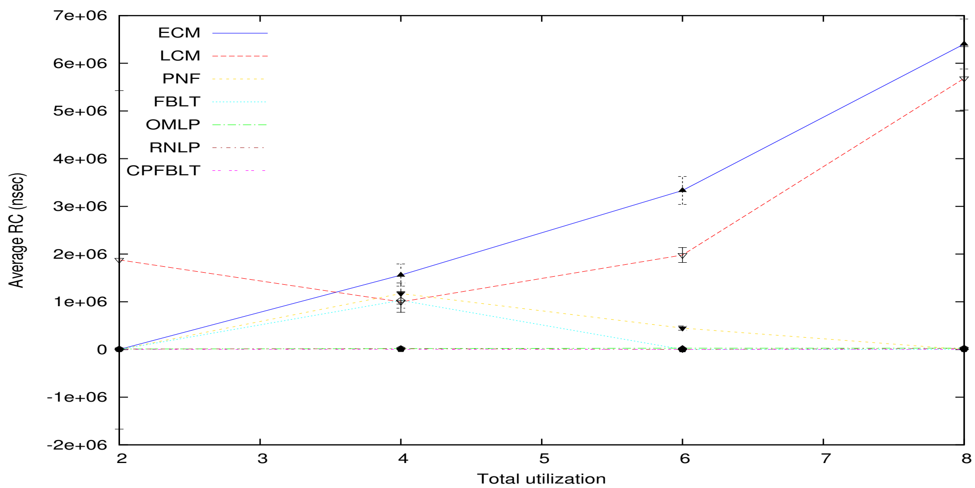


Figure C.65: Avg_RC for Tasksets 65, 335, 605 and 875

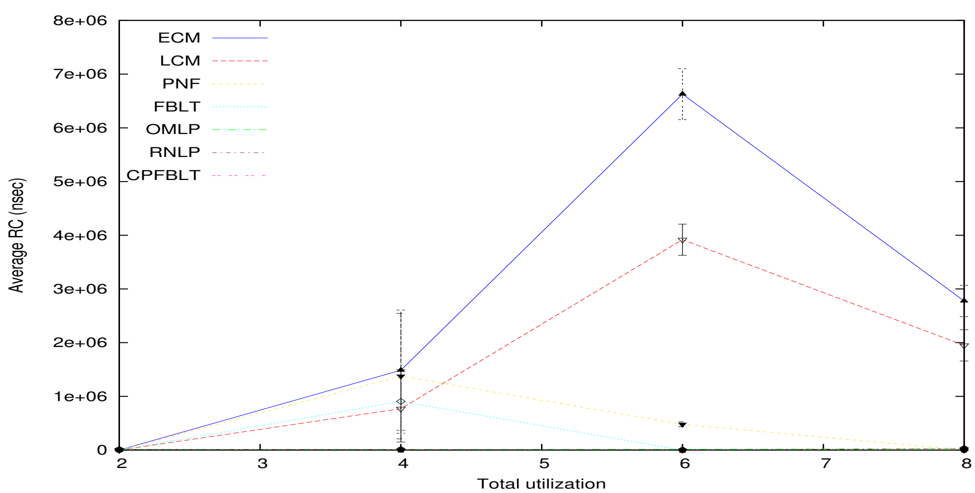


Figure C.66: Avg_RC for Tasksets 66, 336, 606 and 876

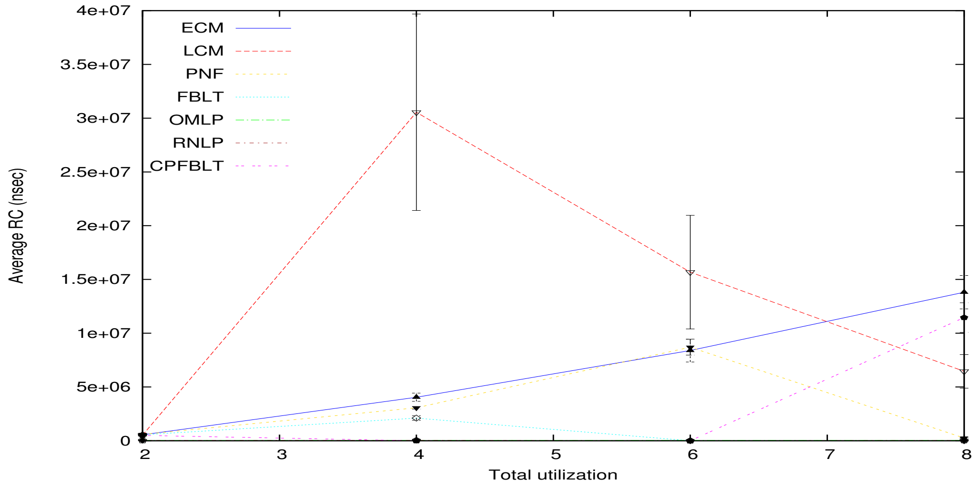


Figure C.67: Avg_RC for Tasksets 67, 337, 607 and 877

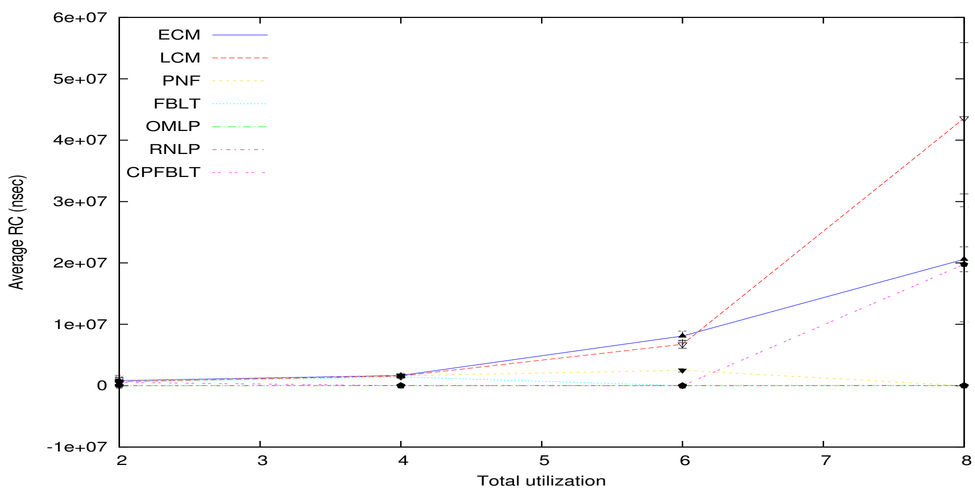


Figure C.68: Avg_RC for Tasksets 68, 338, 608 and 878

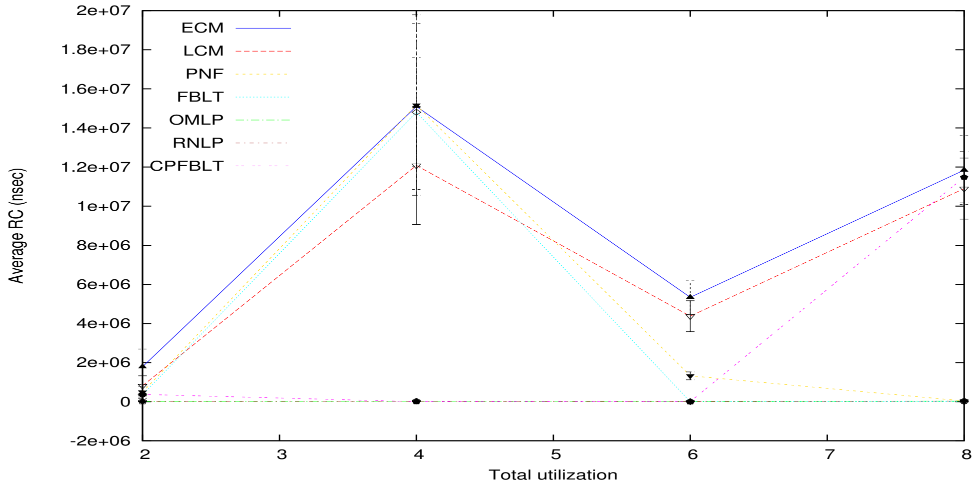


Figure C.69: Avg_RC for Tasksets 69, 339, 609 and 879

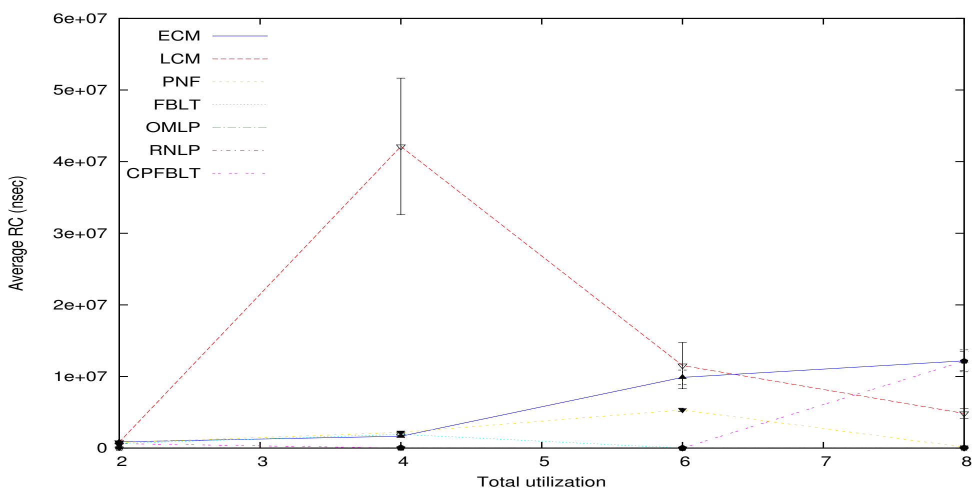


Figure C.70: Avg_RC for Tasksets 70, 340, 610 and 880

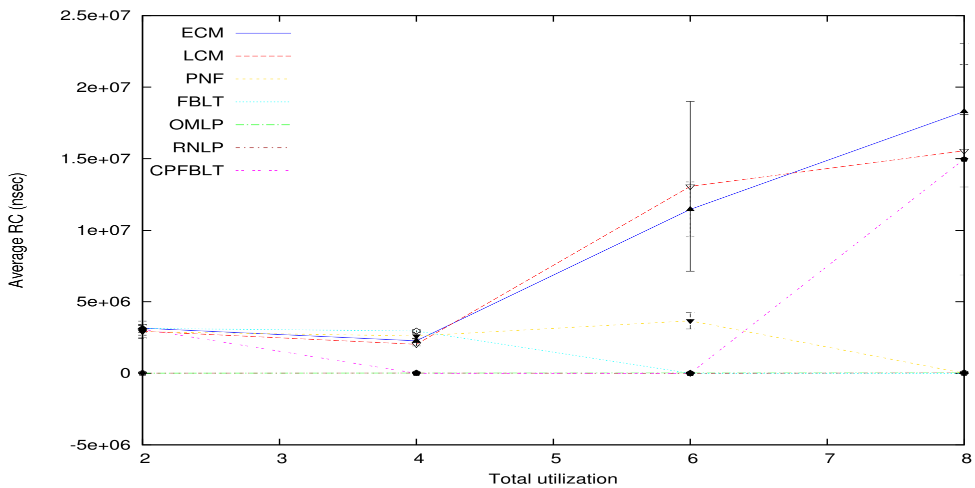


Figure C.71: Avg_RC for Tasksets 71, 341, 611 and 881

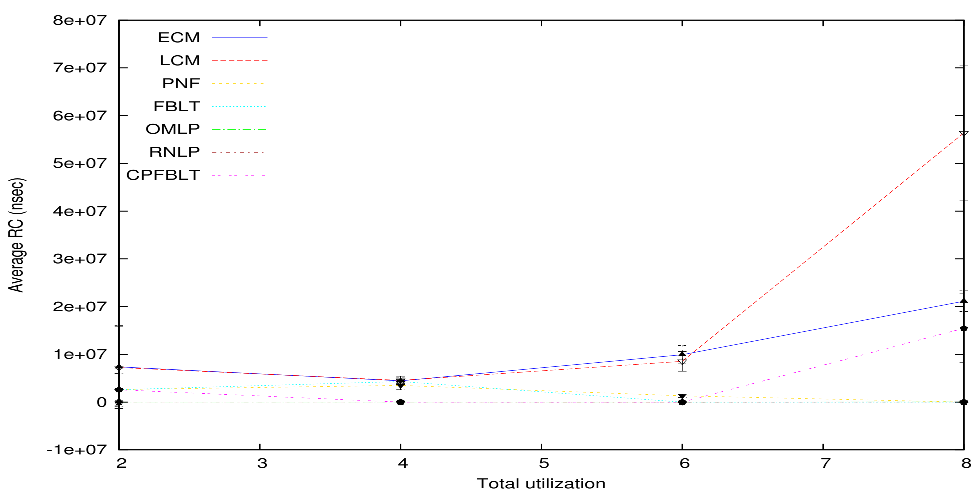


Figure C.72: Avg_RC for Tasksets 72, 342, 612 and 882

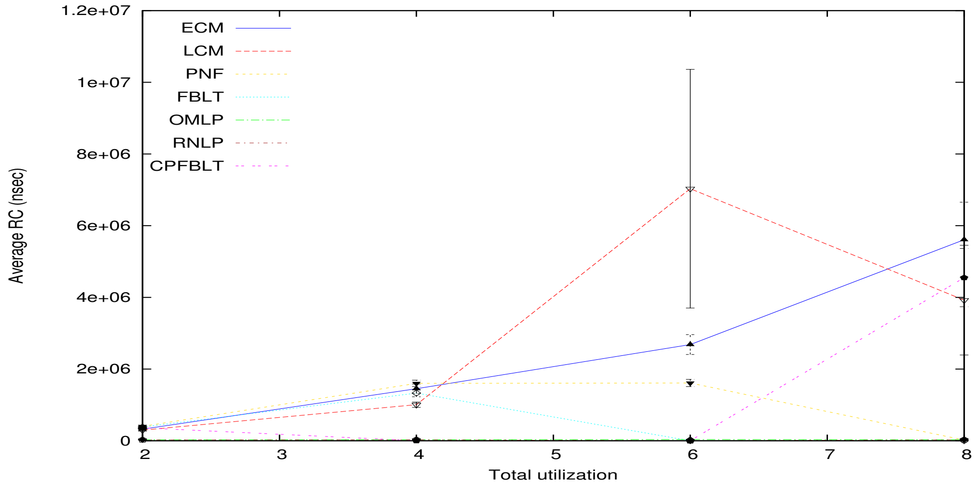


Figure C.73: Avg_RC for Tasksets 73, 343, 613 and 883

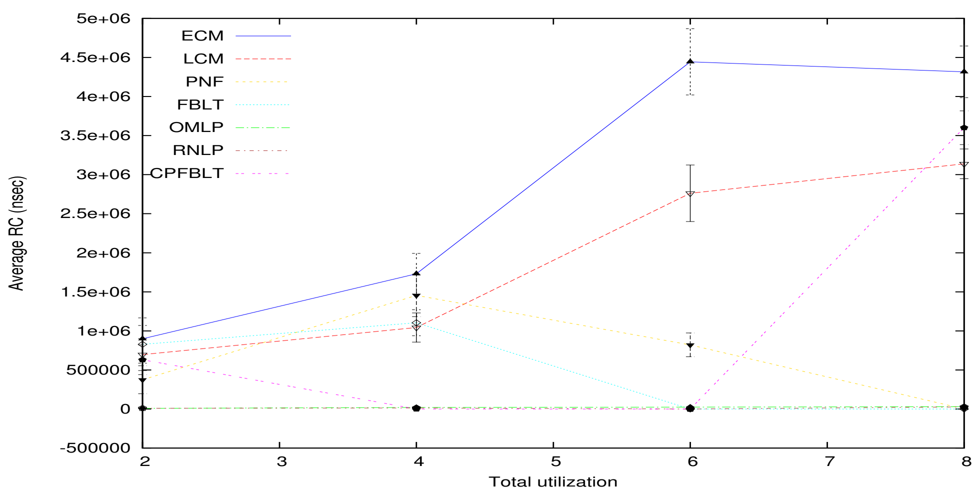


Figure C.74: Avg_RC for Tasksets 74, 344, 614 and 884

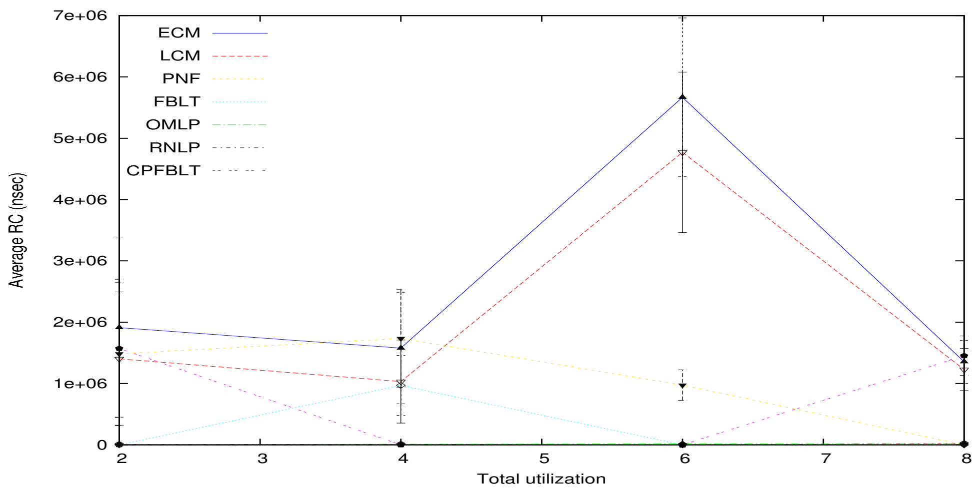


Figure C.75: Avg_RC for Tasksets 75, 345, 615 and 885

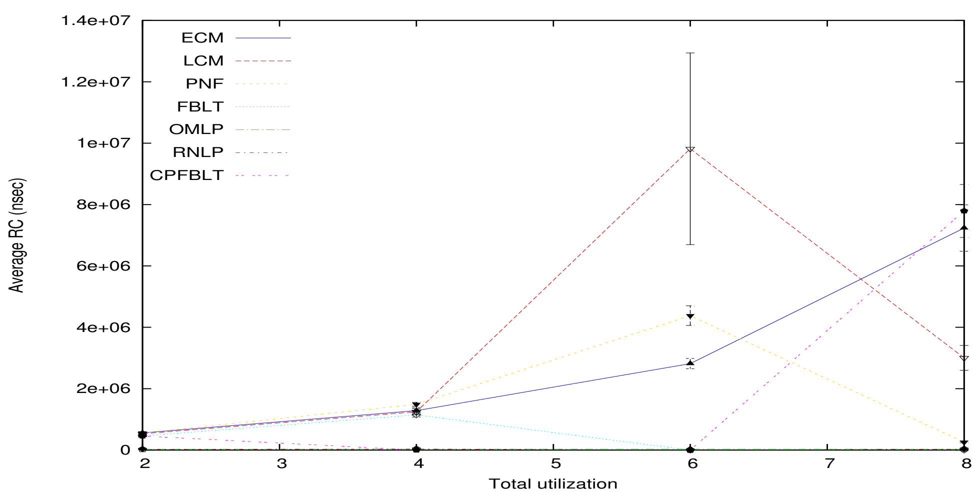


Figure C.76: Avg_RC for Tasksets 76, 346, 616 and 886

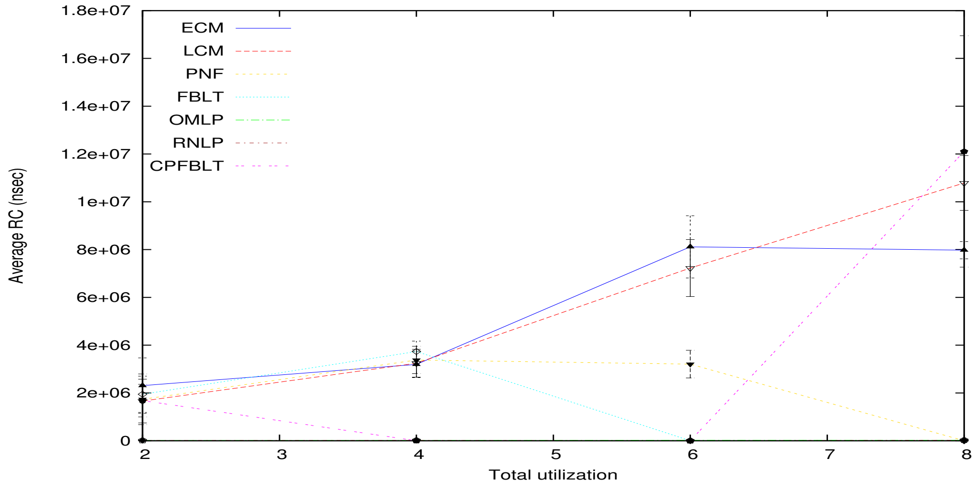


Figure C.77: Avg_RC for Tasksets 77, 347, 617 and 887

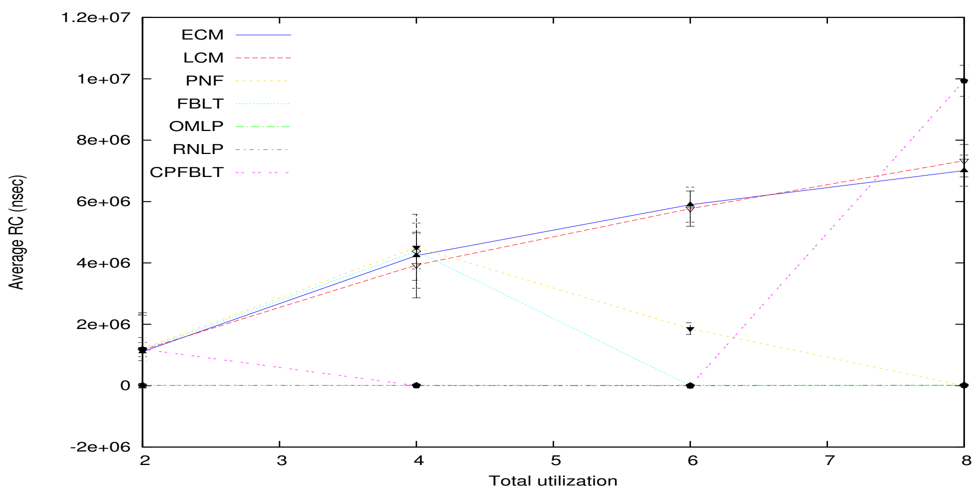


Figure C.78: Avg_RC for Tasksets 78, 348, 618 and 888

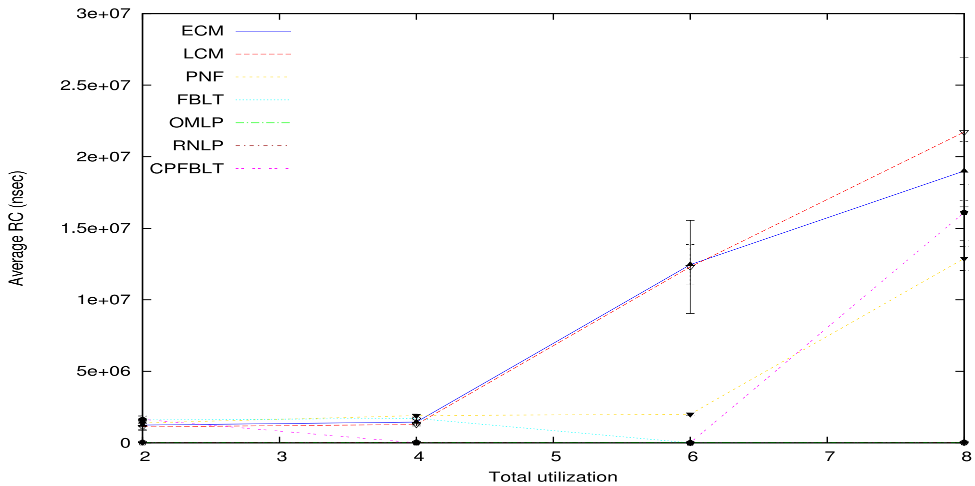


Figure C.79: Avg_RC for Tasksets 79, 349, 619 and 889

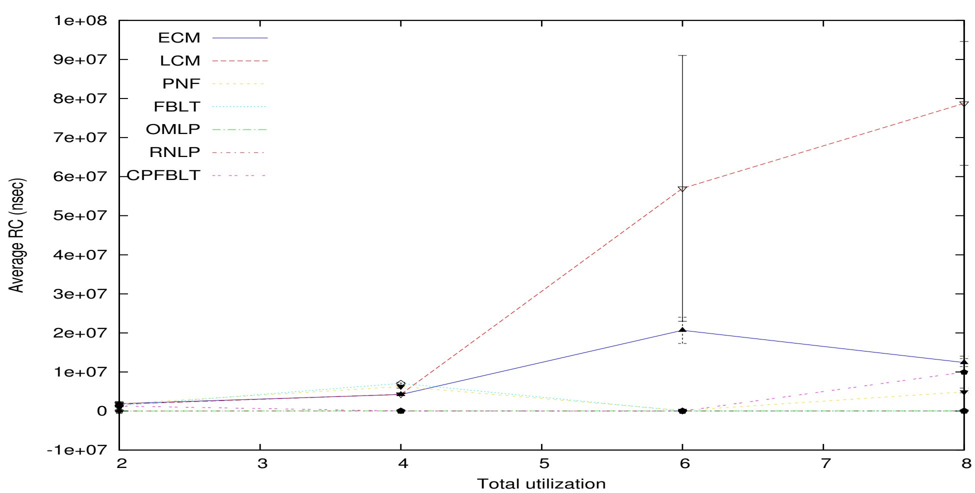


Figure C.80: Avg_RC for Tasksets 80, 350, 620 and 890

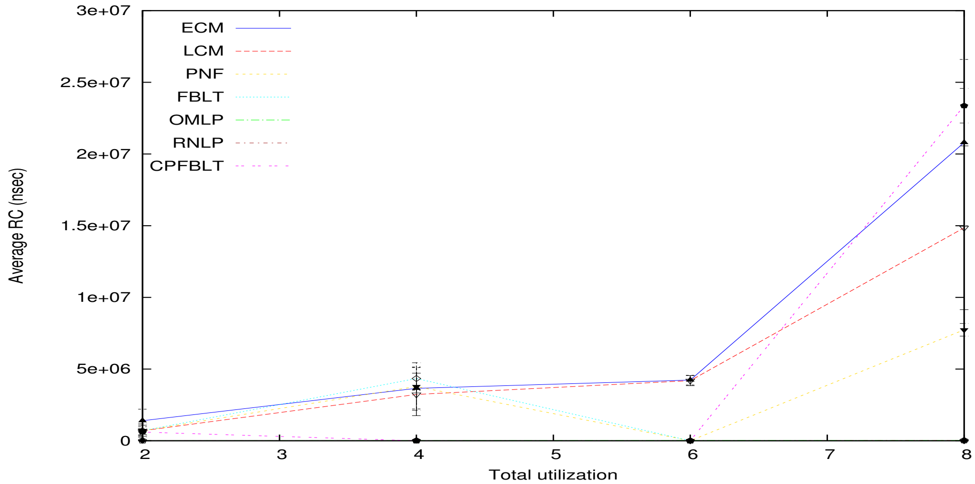


Figure C.81: Avg_RC for Tasksets 81, 351, 621 and 891

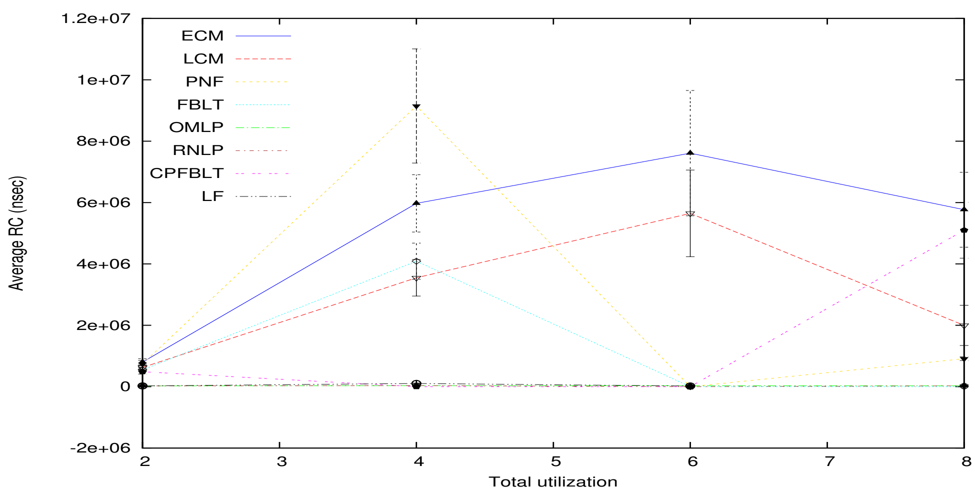


Figure C.82: Avg_RC for Tasksets 82, 352, 622 and 892

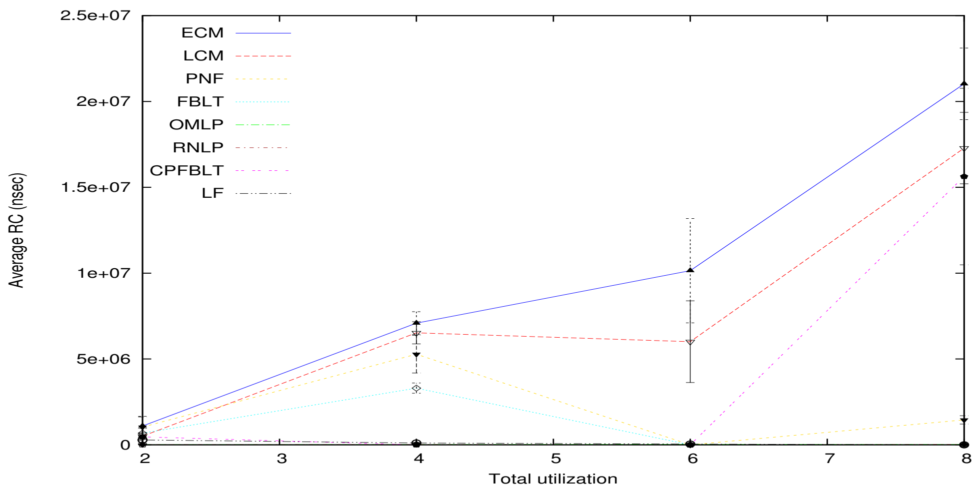


Figure C.83: Avg_RC for Tasksets 83, 353, 623 and 893

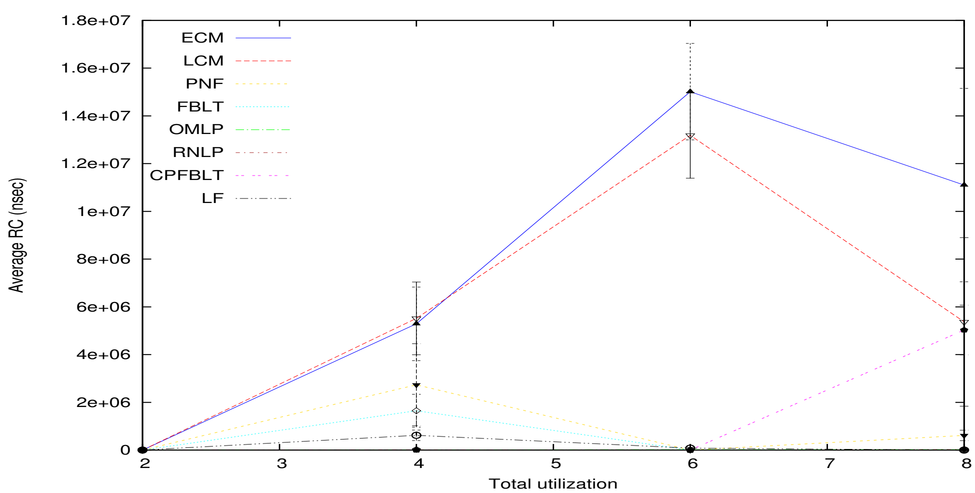


Figure C.84: Avg_RC for Tasksets 84, 354, 624 and 894

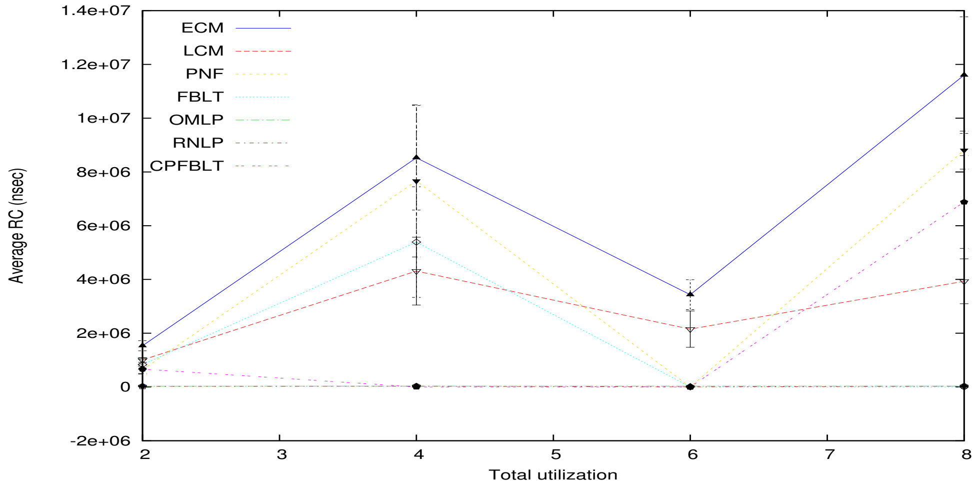


Figure C.85: Avg_RC for Tasksets 85, 355, 625 and 895

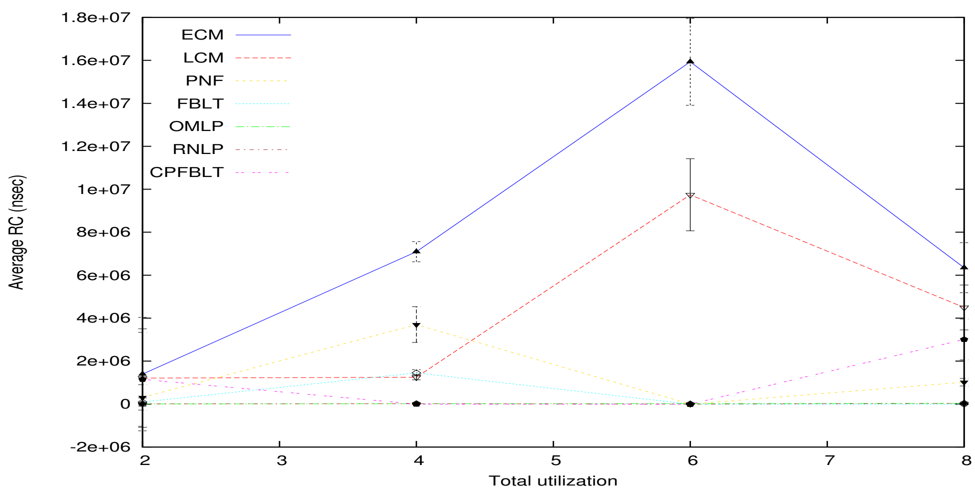


Figure C.86: Avg_RC for Tasksets 86, 356, 626 and 896

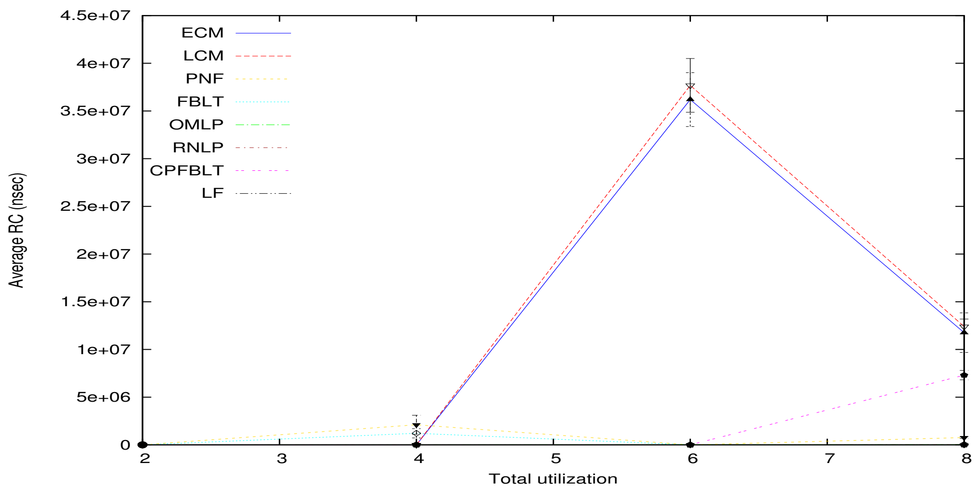


Figure C.87: Avg_RC for Tasksets 87, 357, 627 and 897

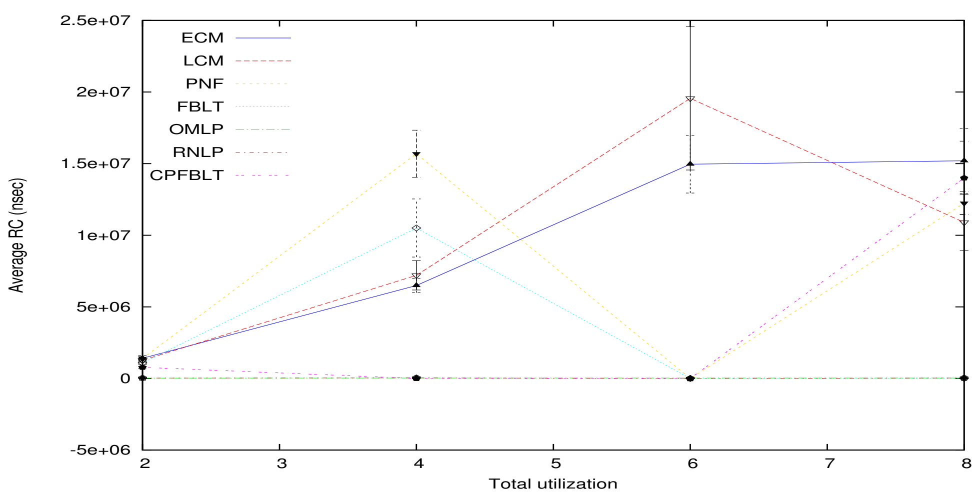


Figure C.88: Avg_RC for Tasksets 88, 358, 628 and 898

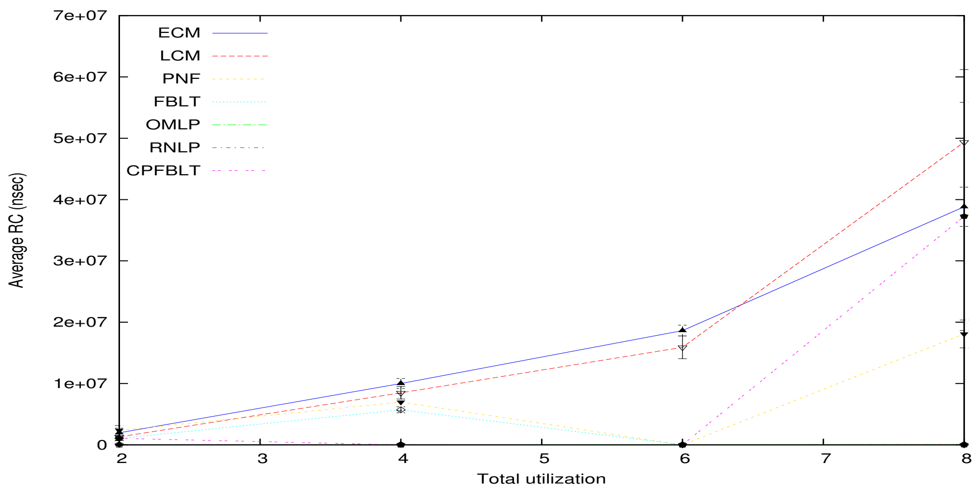


Figure C.89: Avg_RC for Tasksets 89, 359, 629 and 899

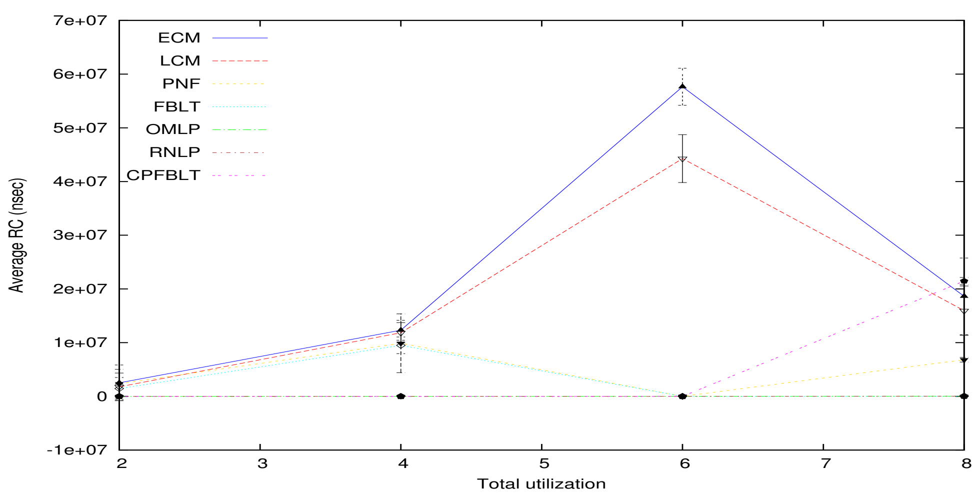


Figure C.90: Avg_RC for Tasksets 90, 360, 630 and 900

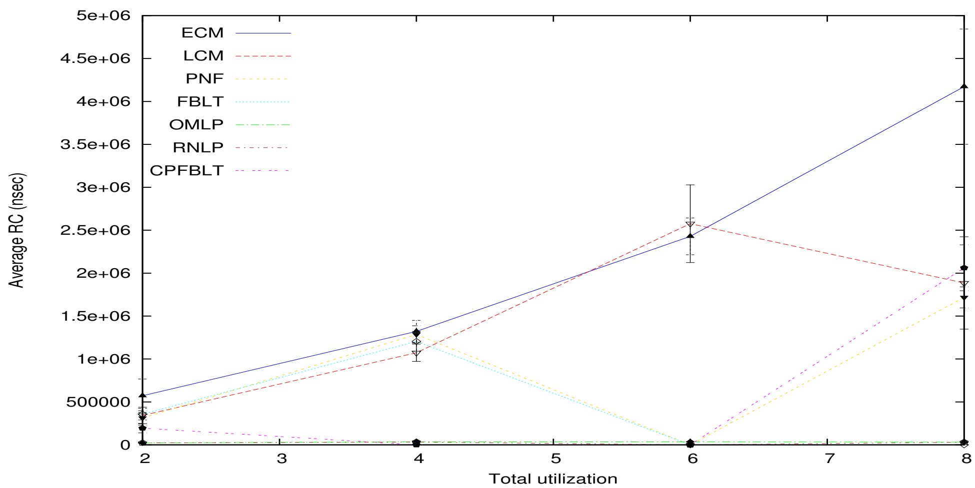


Figure C.91: Avg_RC for Tasksets 91, 361, 631 and 901

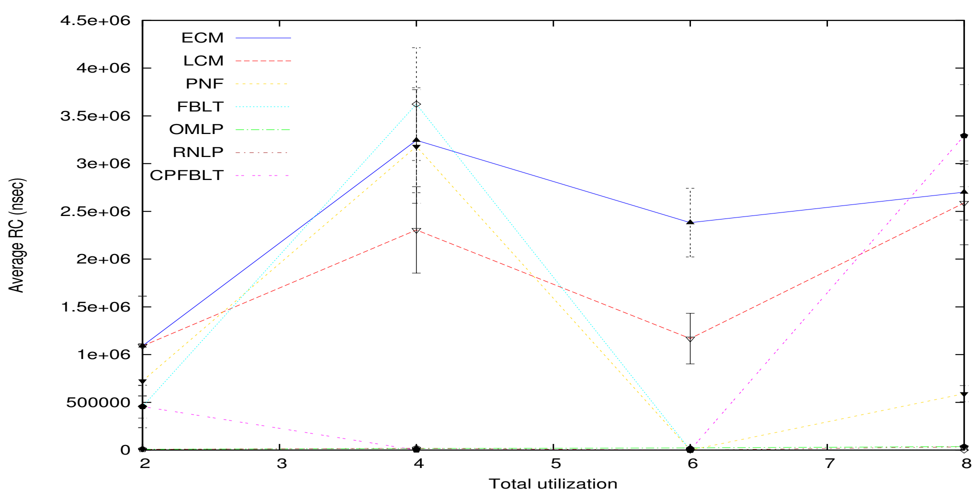


Figure C.92: Avg_RC for Tasksets 92, 362, 632 and 902

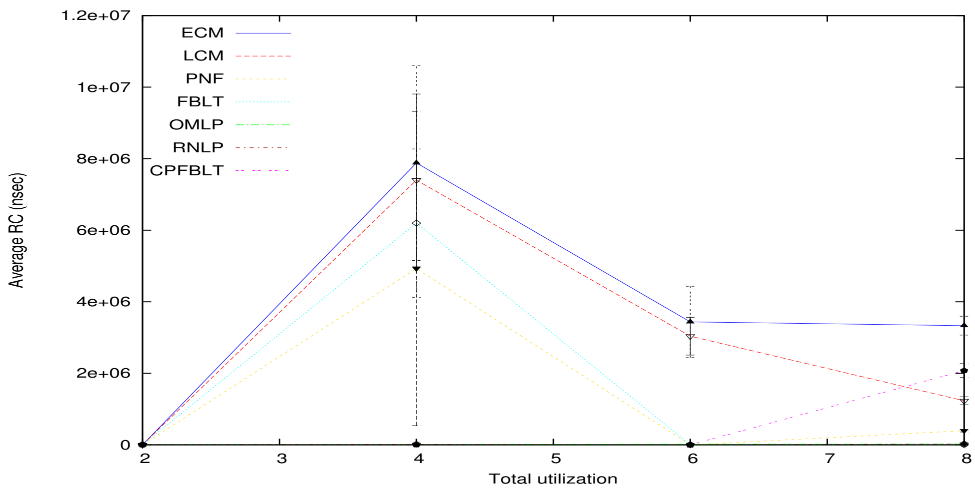


Figure C.93: Avg_RC for Tasksets 93, 363, 633 and 903

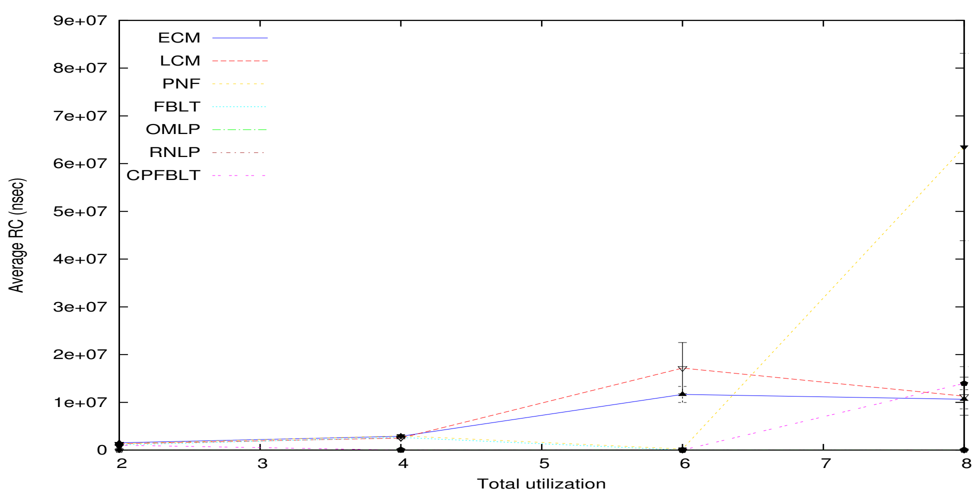


Figure C.94: Avg_RC for Tasksets 94, 364, 634 and 904

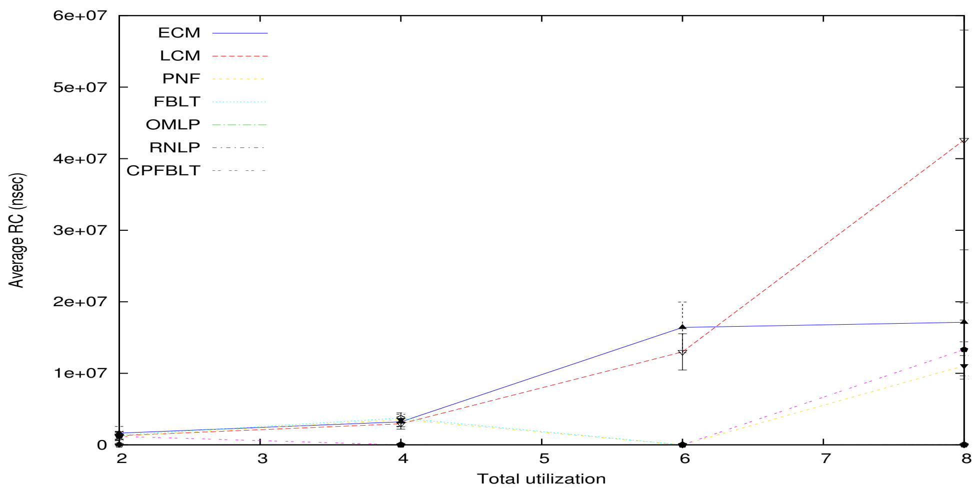


Figure C.95: Avg_RC for Tasksets 95, 365, 635 and 905

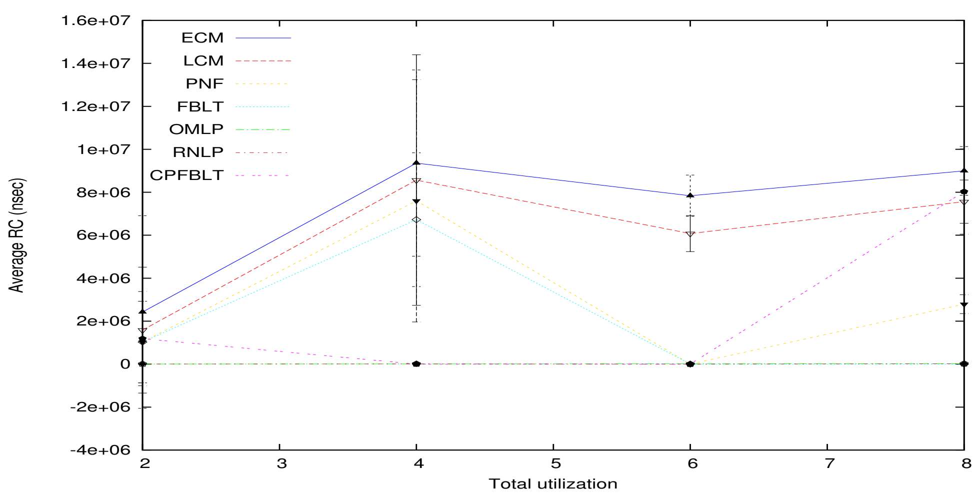


Figure C.96: Avg_RC for Tasksets 96, 366, 636 and 906

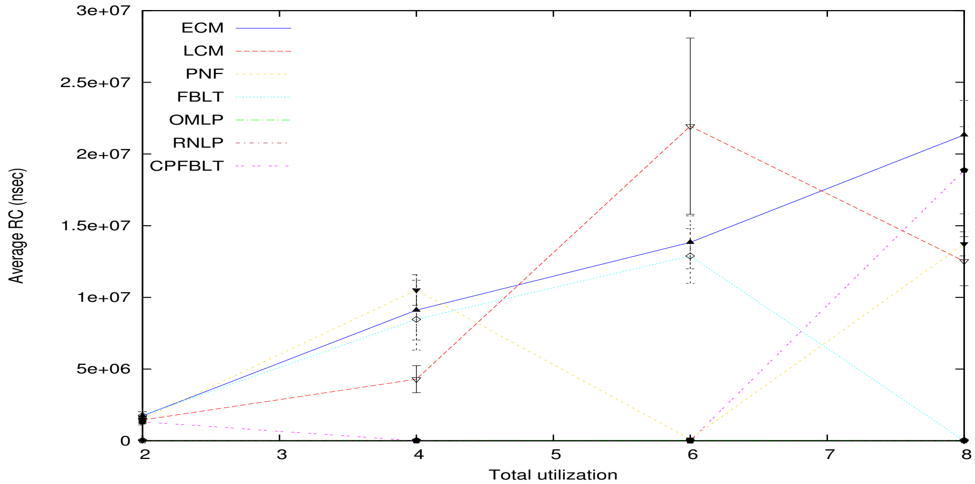


Figure C.97: Avg_RC for Tasksets 97, 367, 637 and 907

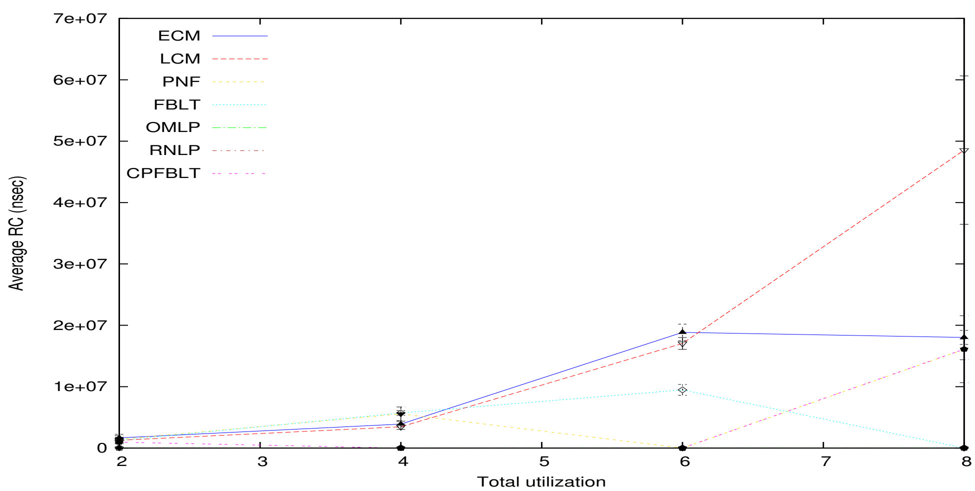


Figure C.98: Avg_RC for Tasksets 98, 368, 638 and 908

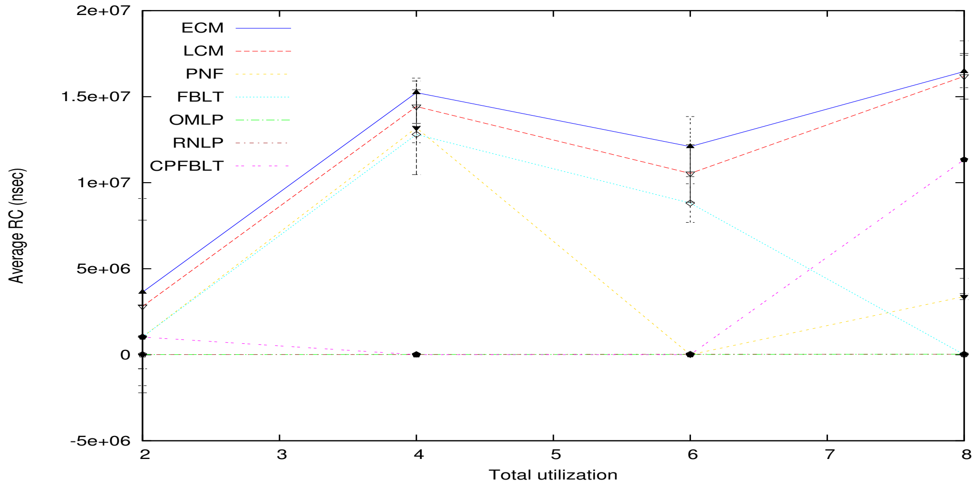


Figure C.99: Avg_RC for Tasksets 99, 369, 639 and 909

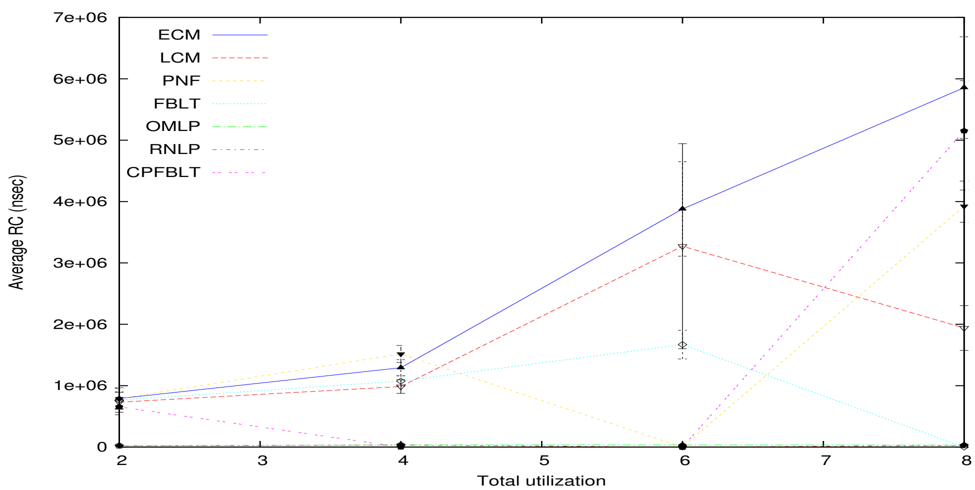


Figure C.100: Avg_RC for Tasksets 100, 370, 640 and 910

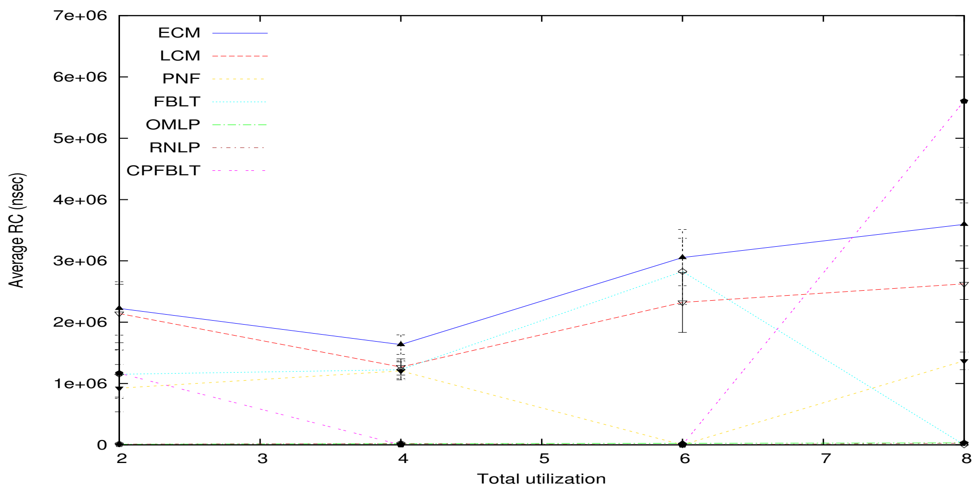


Figure C.101: Avg_RC for Tasksets 101, 371, 641 and 911

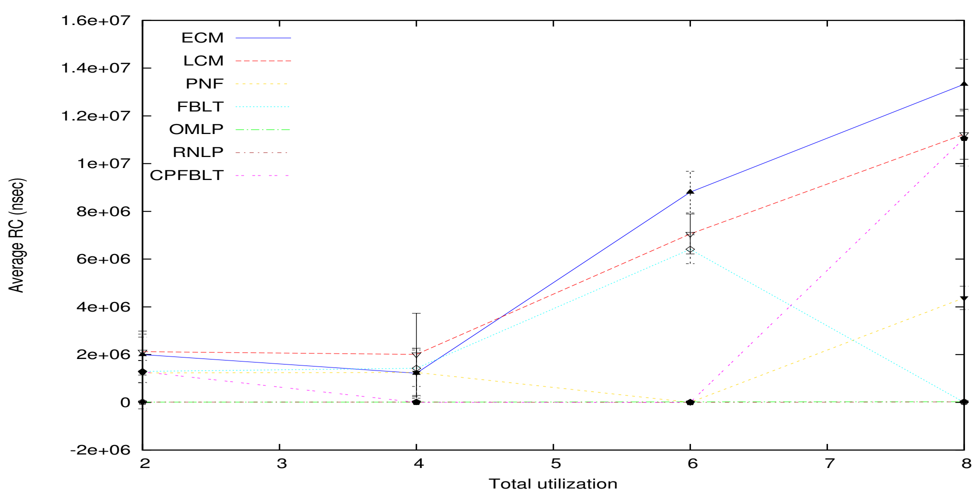


Figure C.102: Avg_RC for Tasksets 102, 372, 642 and 912

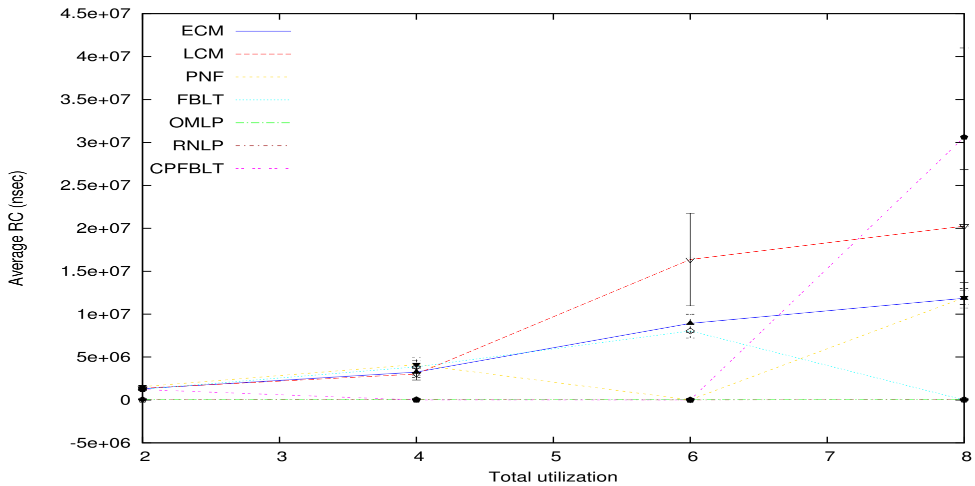


Figure C.103: Avg_RC for Tasksets 103, 373, 643 and 913

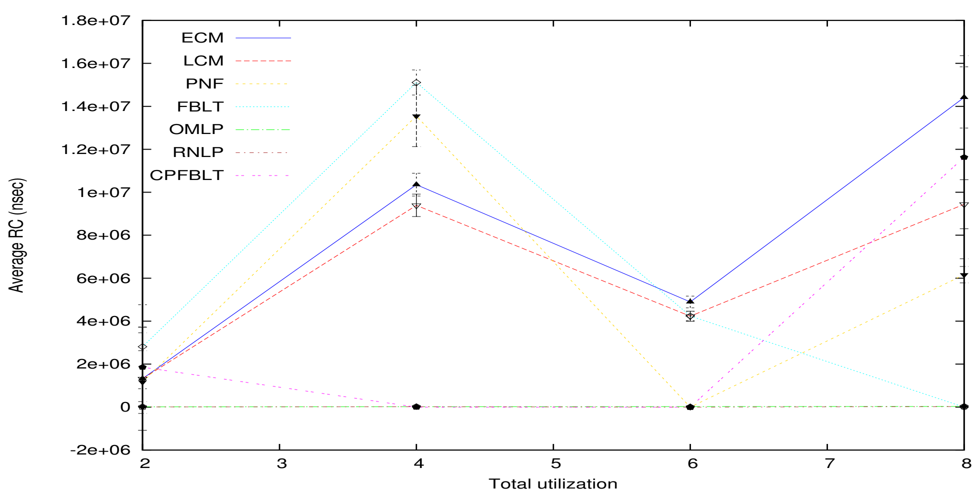


Figure C.104: Avg_RC for Tasksets 104, 374, 644 and 914

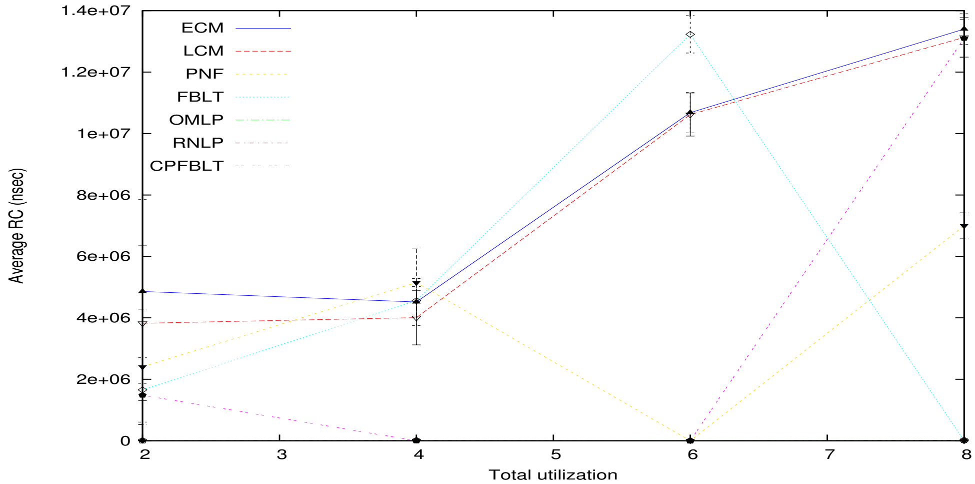


Figure C.105: Avg_RC for Tasksets 105, 375, 645 and 915

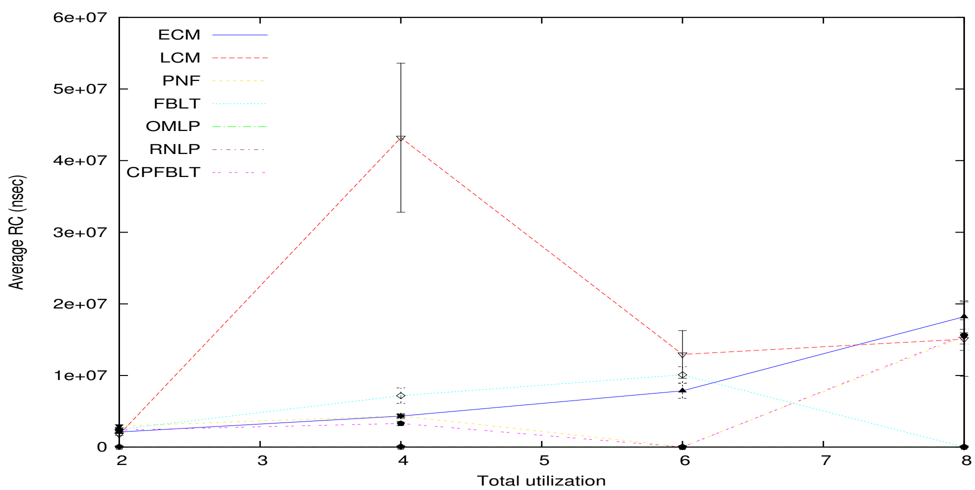


Figure C.106: Avg_RC for Tasksets 106, 376, 646 and 916

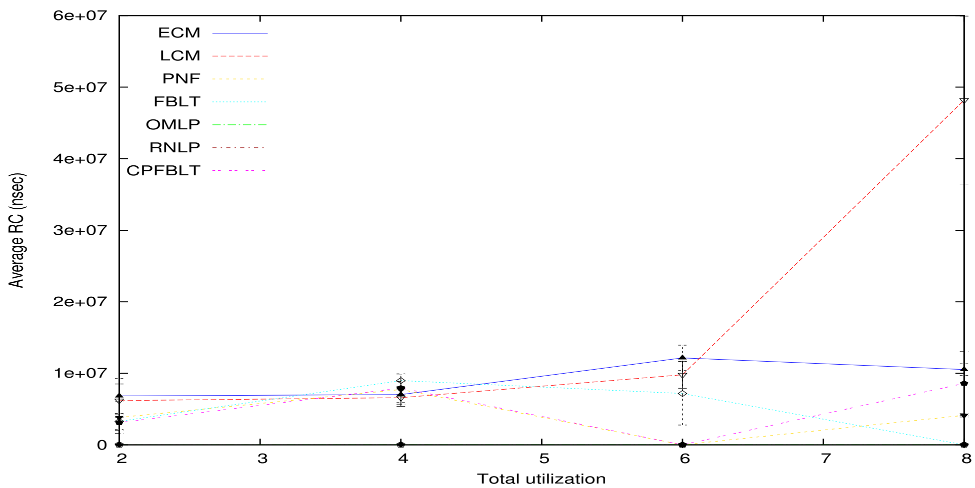


Figure C.107: Avg_RC for Tasksets 107, 377, 647 and 917

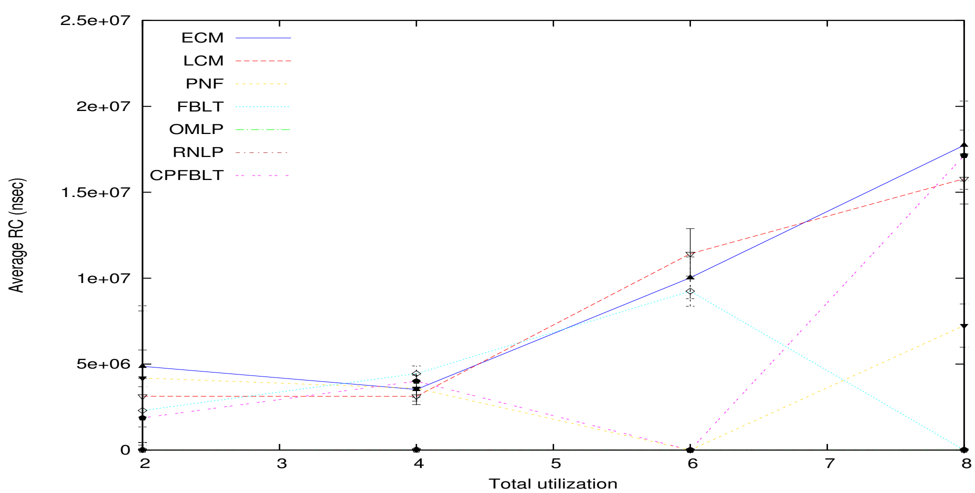


Figure C.108: Avg_RC for Tasksets 108, 378, 648 and 918

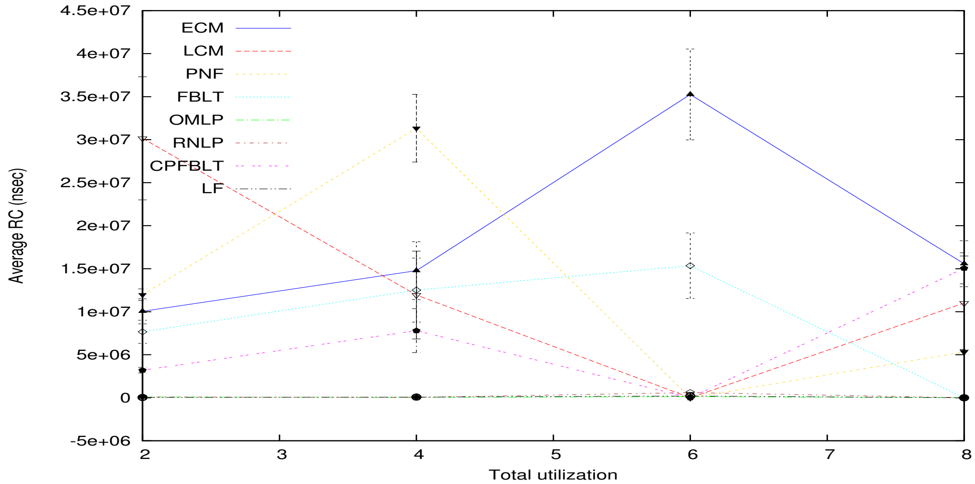


Figure C.109: Avg_RC for Tasksets 109, 379, 649 and 919

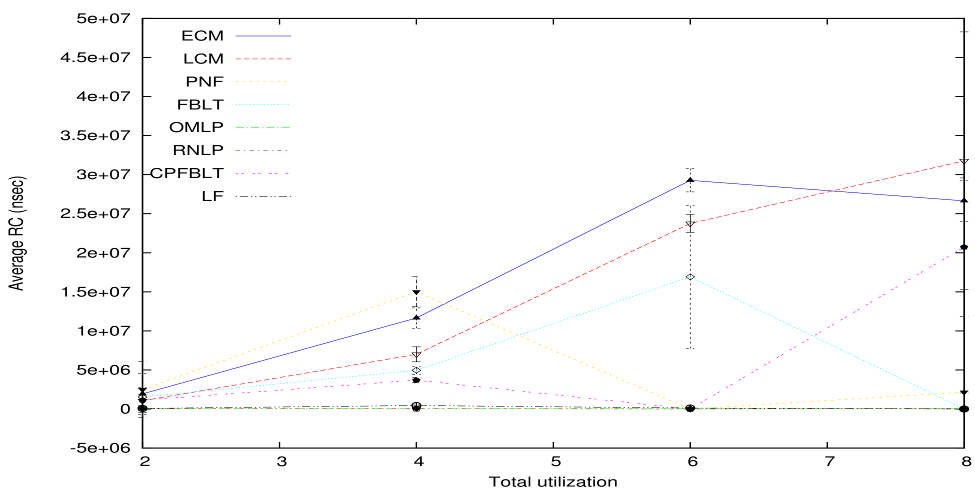


Figure C.110: Avg_RC for Tasksets 110, 380, 650 and 920

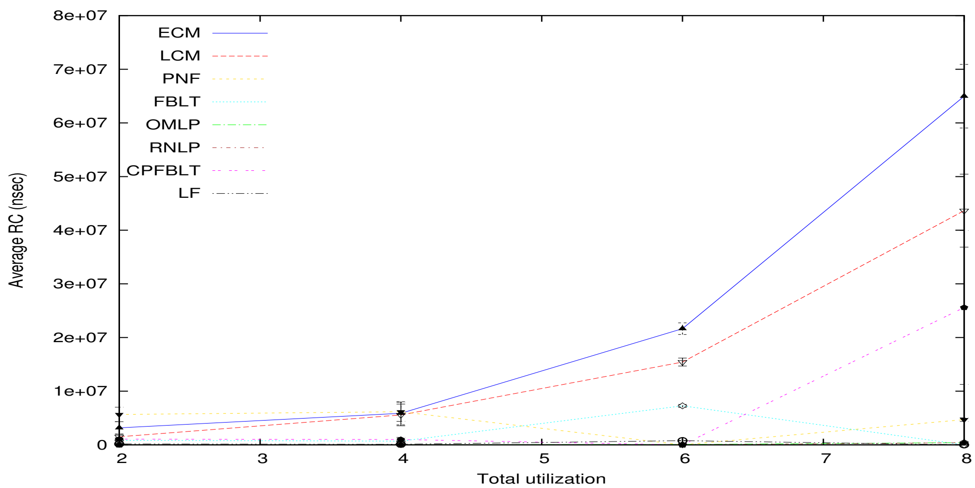


Figure C.111: Avg_RC for Tasksets 111, 381, 651 and 921

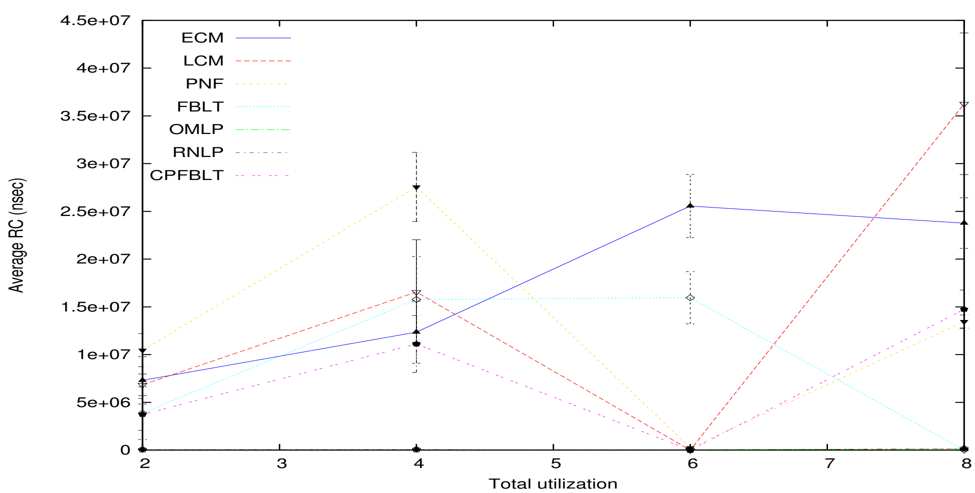


Figure C.112: Avg_RC for Tasksets 112, 382, 652 and 922

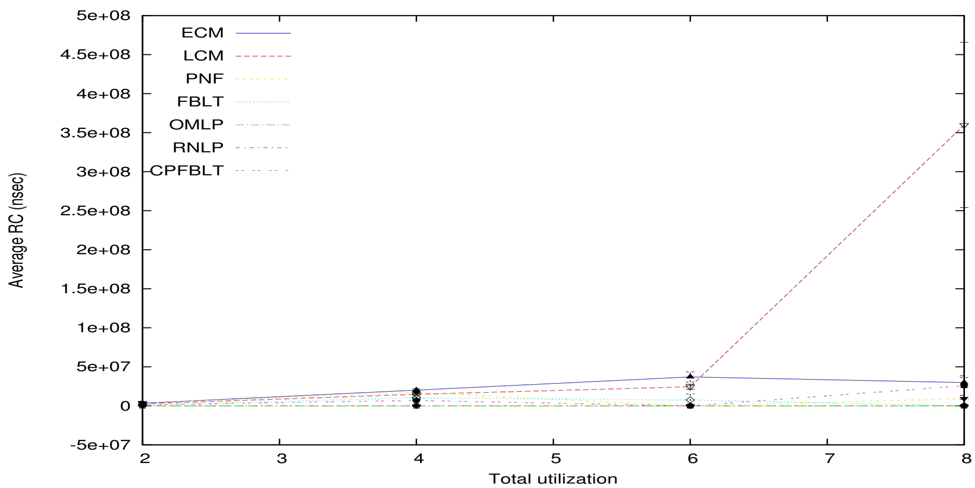


Figure C.113: Avg_RC for Tasksets 113, 383, 653 and 923

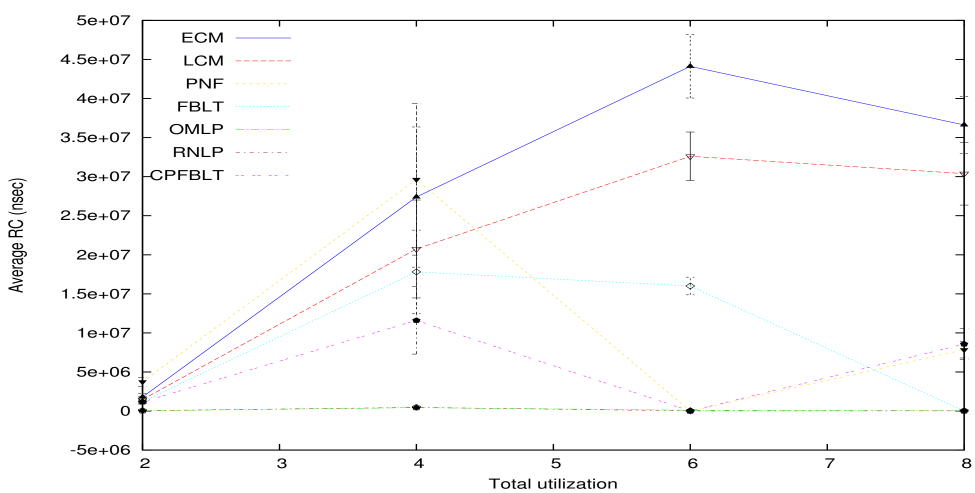


Figure C.114: Avg_RC for Tasksets 114, 384, 654 and 924

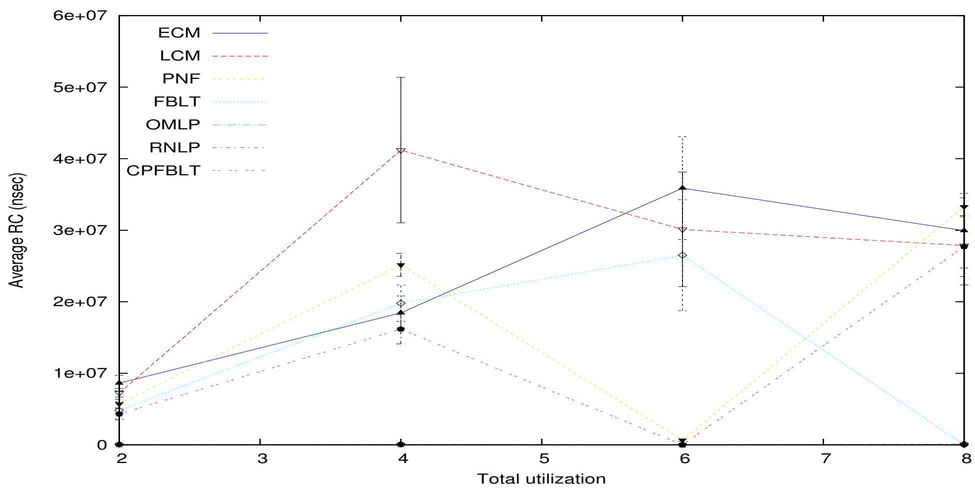


Figure C.115: Avg_RC for Tasksets 115, 385, 655 and 925

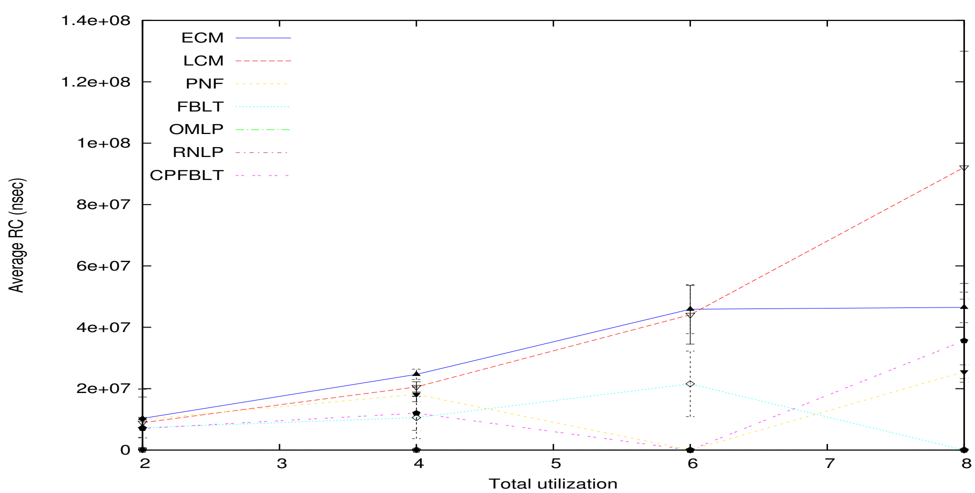


Figure C.116: Avg_RC for Tasksets 116, 386, 656 and 926

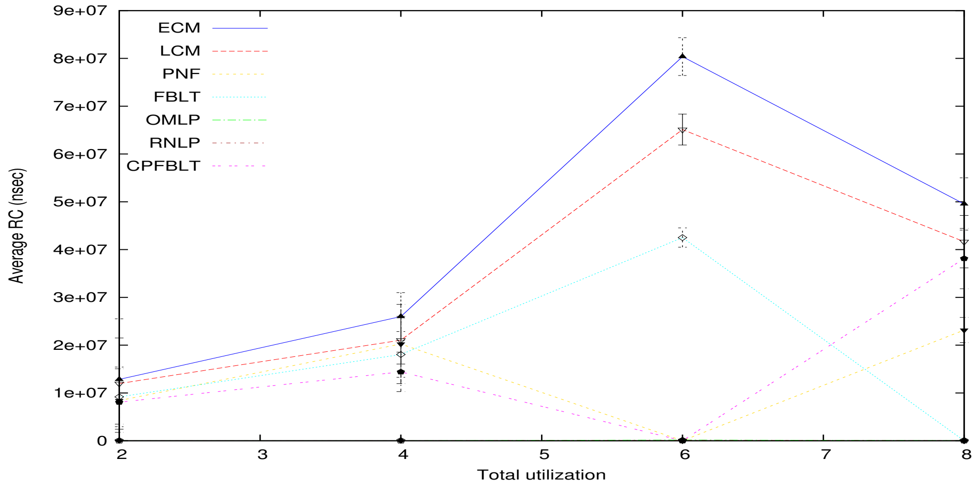


Figure C.117: Avg_RC for Tasksets 117, 387, 657 and 927

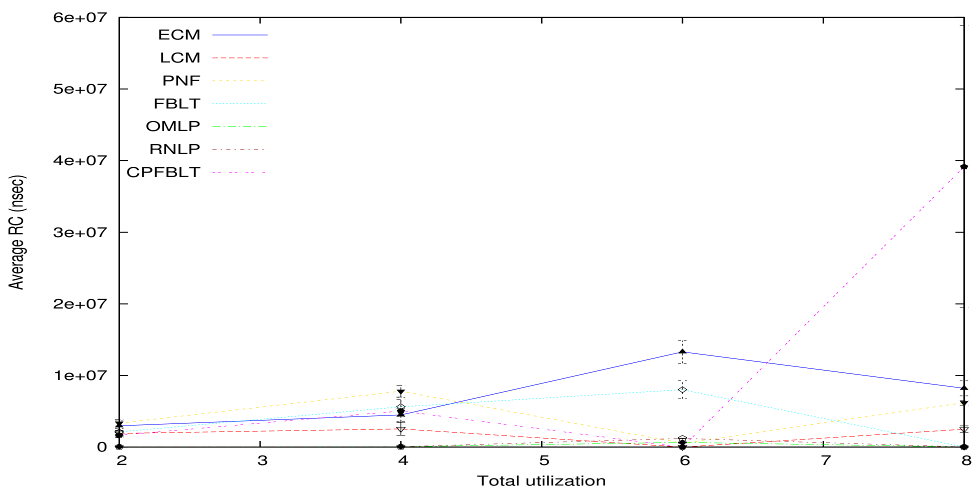


Figure C.118: Avg_RC for Tasksets 118, 388, 658 and 928

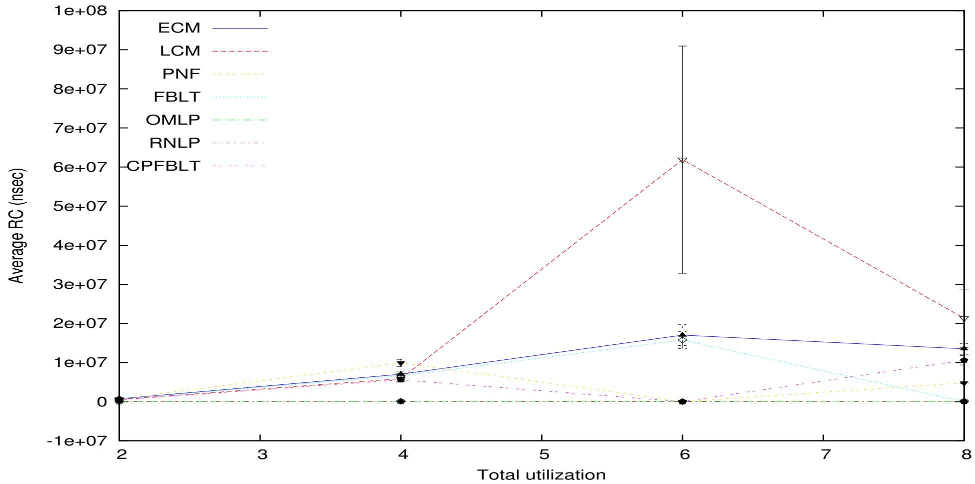


Figure C.119: Avg_RC for Tasksets 119, 389, 659 and 929

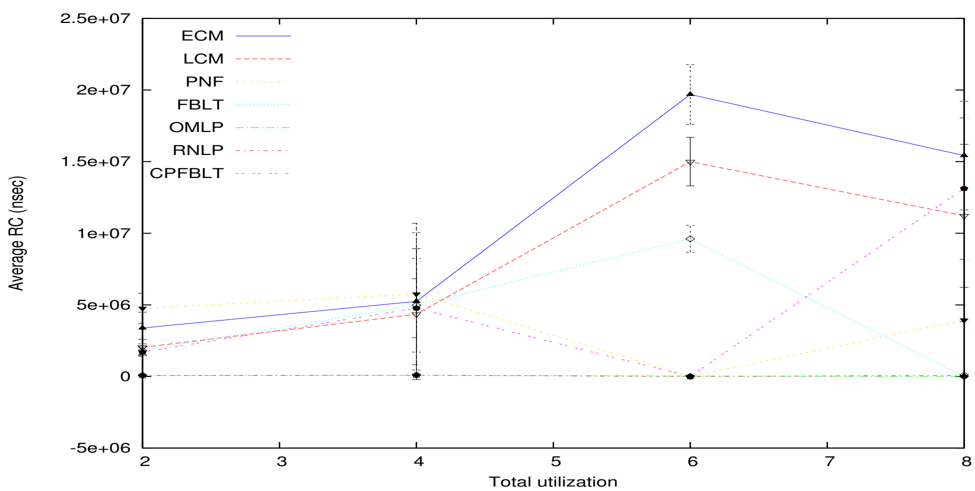


Figure C.120: Avg_RC for Tasksets 120, 390, 660 and 930

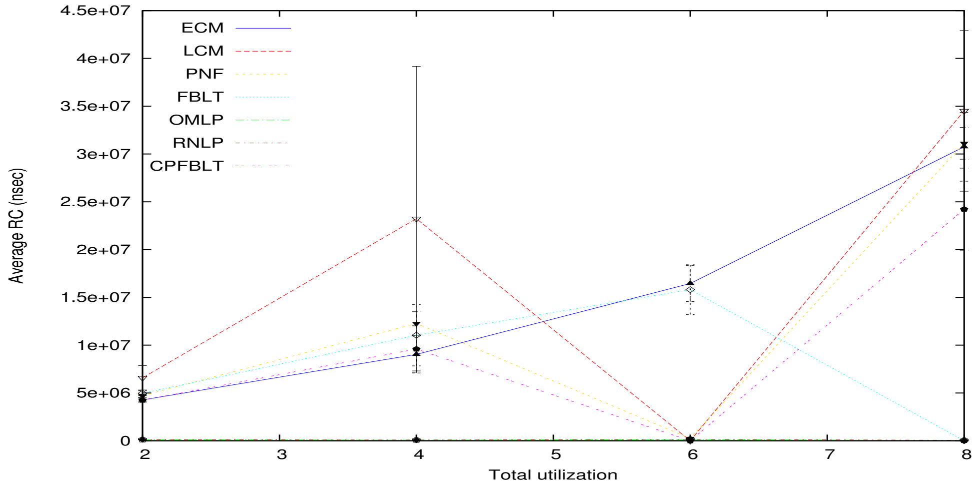


Figure C.121: Avg_RC for Tasksets 121, 391, 661 and 931

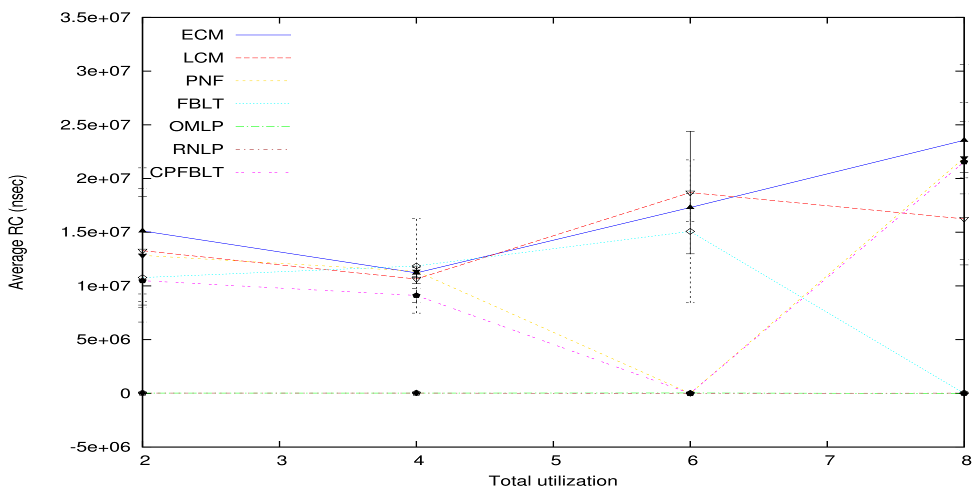


Figure C.122: Avg_RC for Tasksets 122, 392, 662 and 932

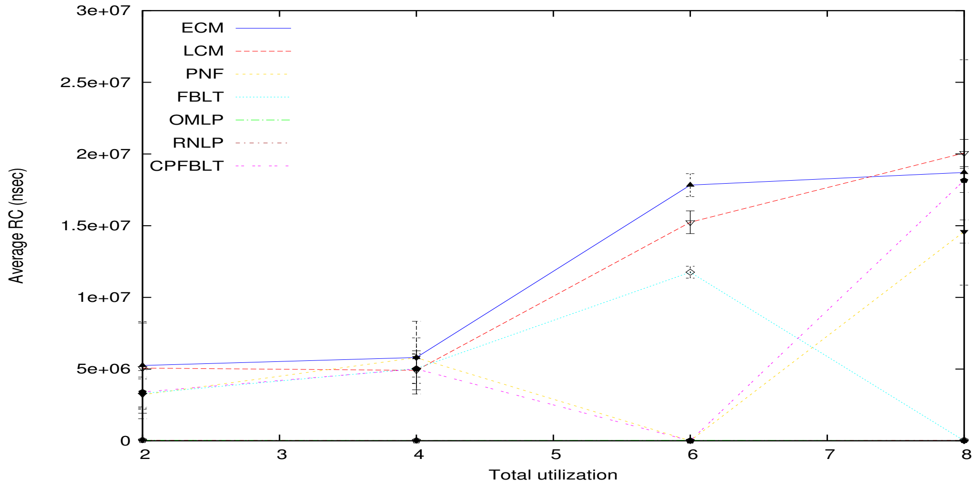


Figure C.123: Avg_RC for Tasksets 123, 393, 663 and 933

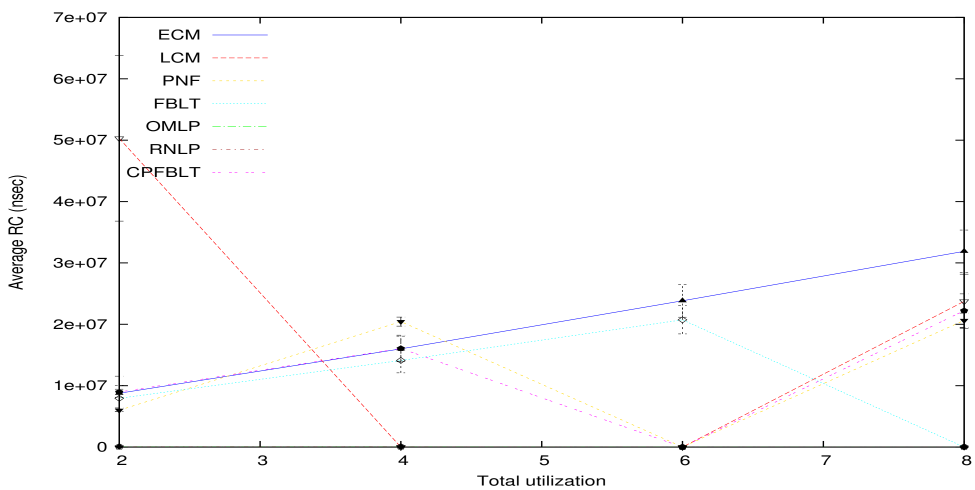


Figure C.124: Avg_RC for Tasksets 124, 394, 664 and 934

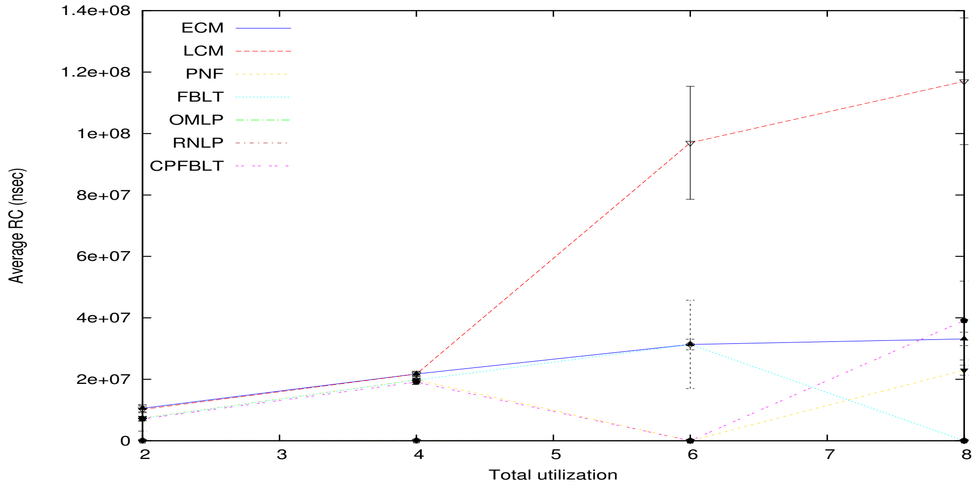


Figure C.125: Avg_RC for Tasksets 125, 395, 665 and 935

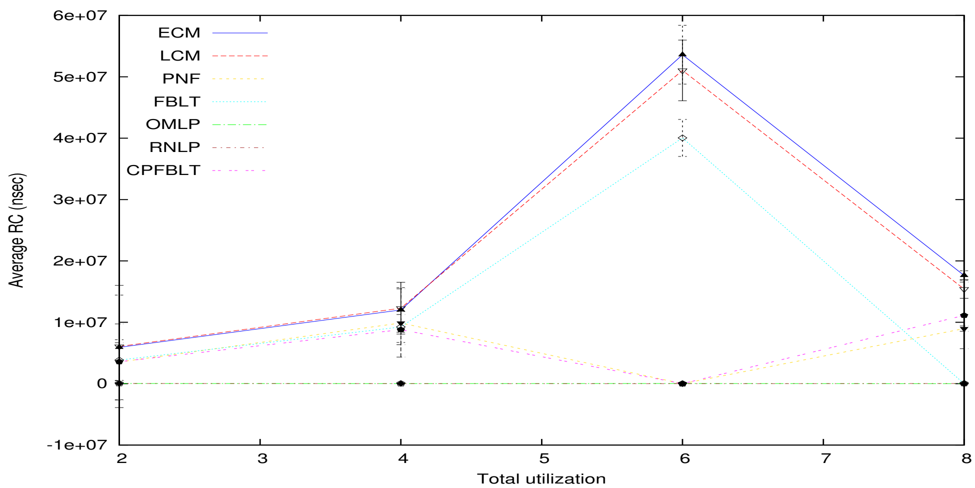


Figure C.126: Avg_RC for Tasksets 126, 396, 666 and 936

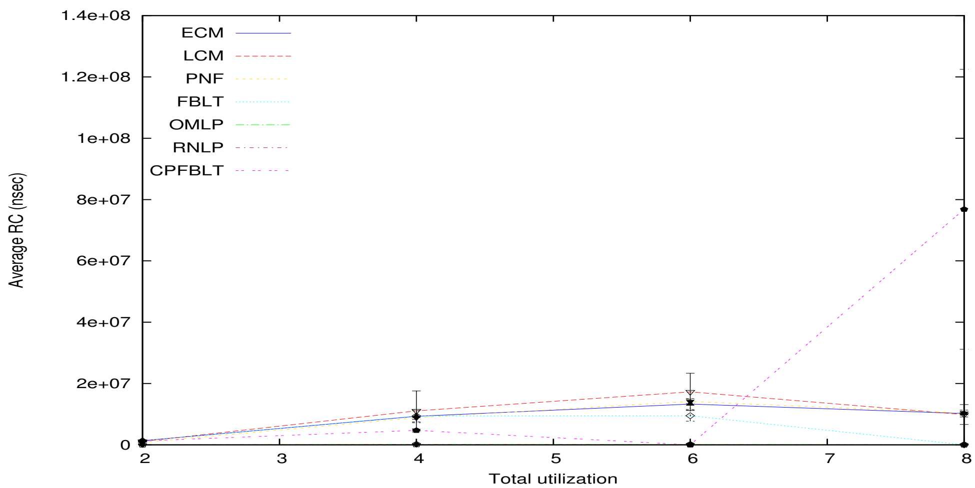


Figure C.127: Avg_RC for Tasksets 127, 397, 667 and 937

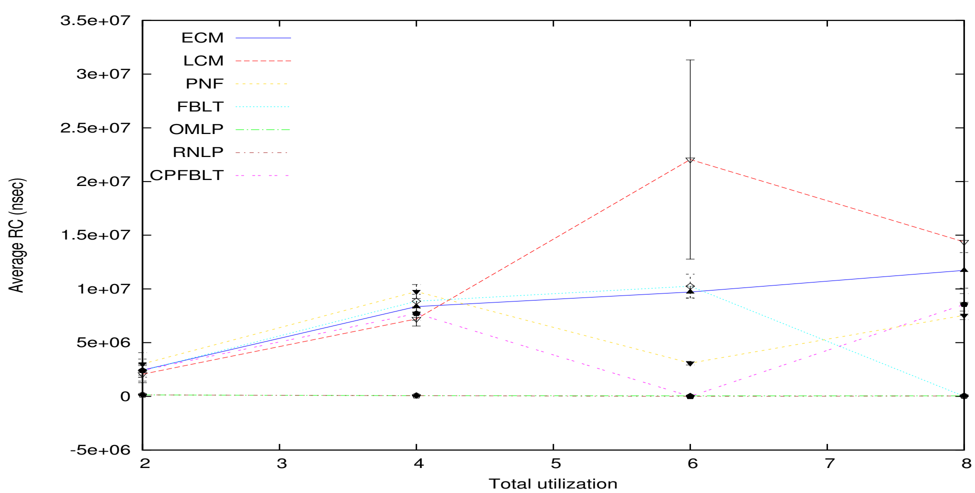


Figure C.128: Avg_RC for Tasksets 128, 398, 668 and 938

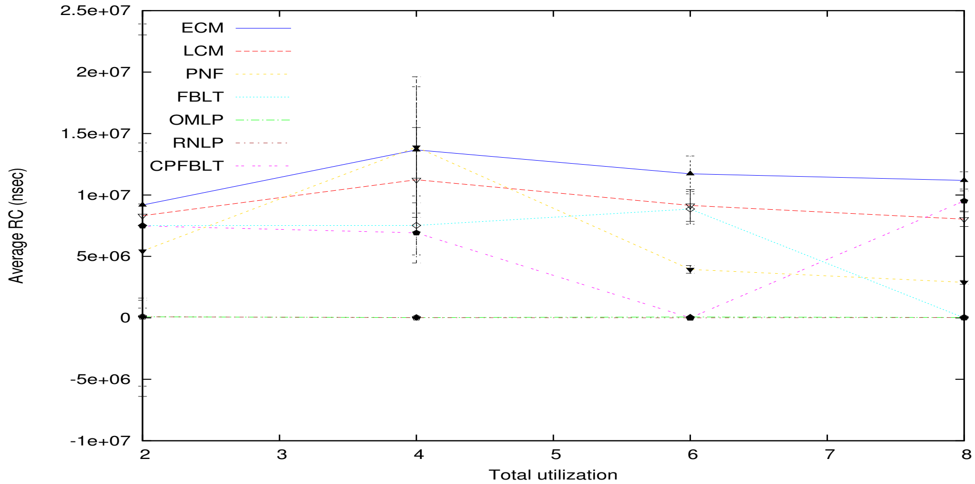


Figure C.129: Avg_RC for Tasksets 129, 399, 669 and 939

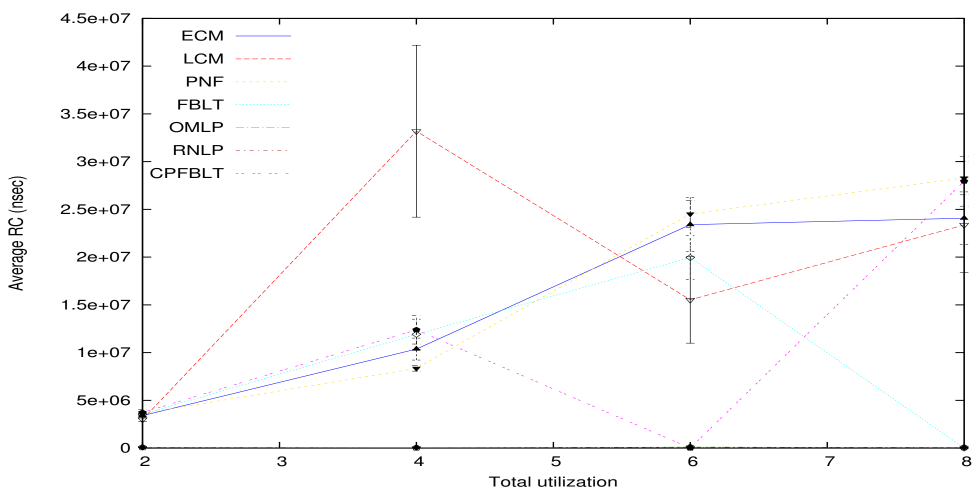


Figure C.130: Avg_RC for Tasksets 130, 400, 670 and 940

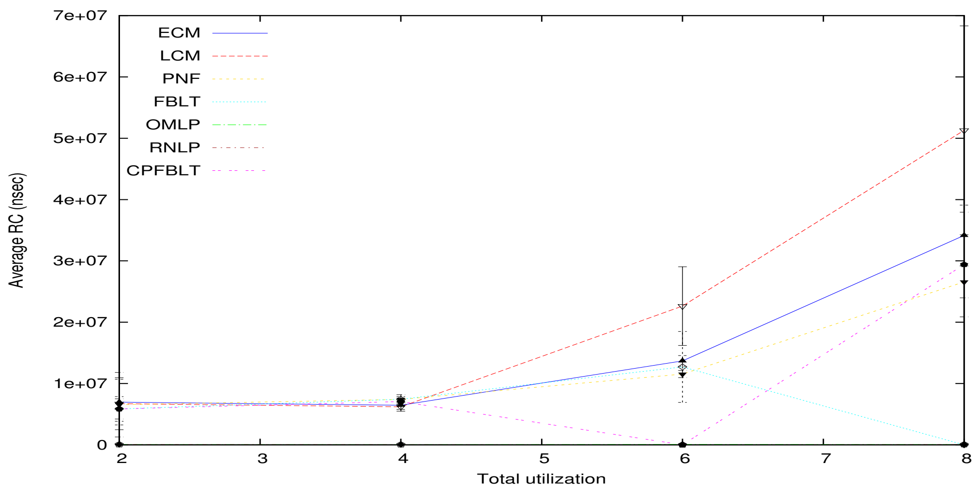


Figure C.131: Avg_RC for Tasksets 131, 401, 671 and 941

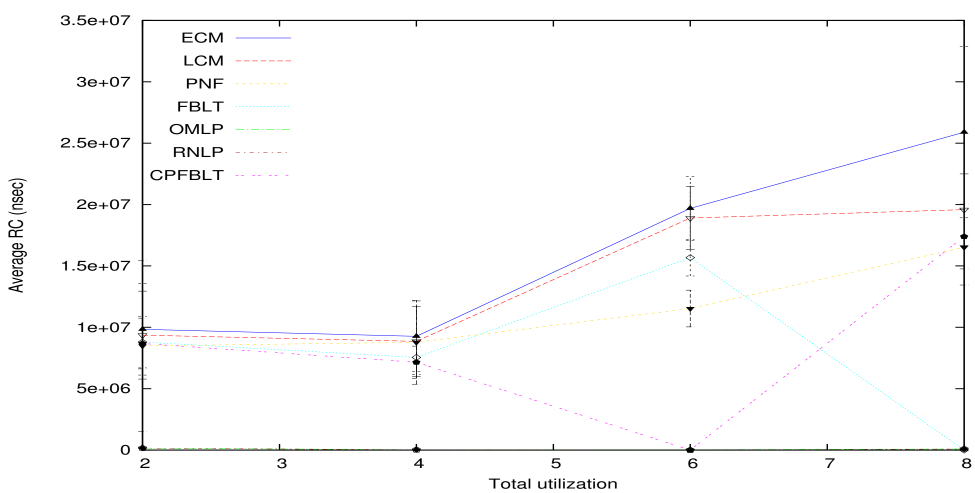


Figure C.132: Avg_RC for Tasksets 132, 402, 672 and 942

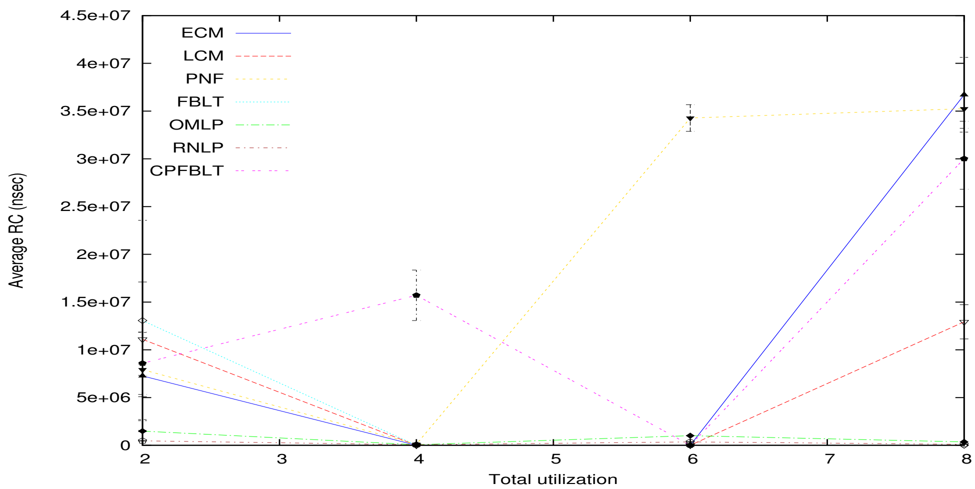


Figure C.133: Avg_RC for Tasksets 133, 403, 673 and 943

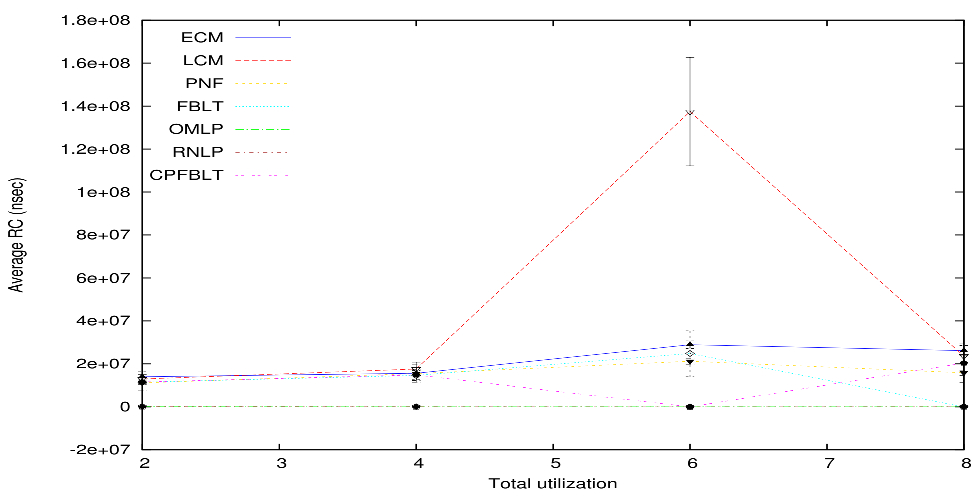


Figure C.134: Avg_RC for Tasksets 134, 404, 674 and 944

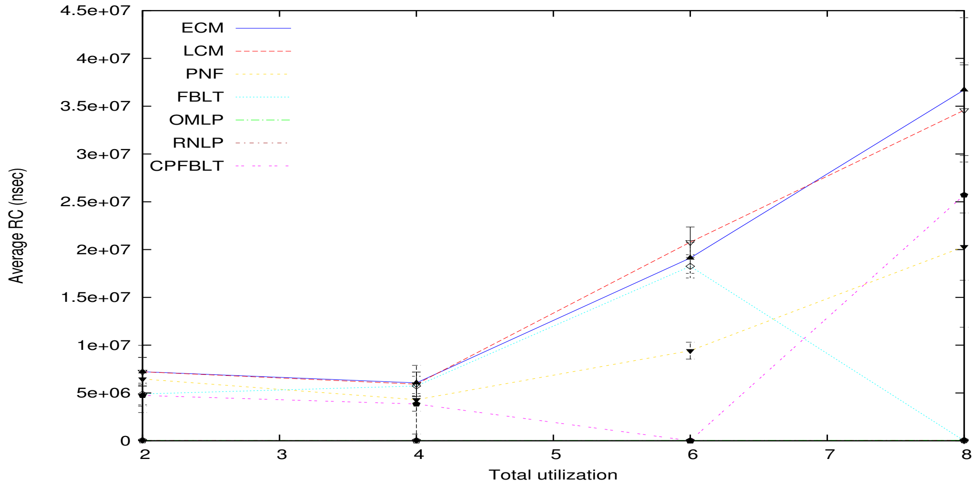


Figure C.135: Avg_RC for Tasksets 135, 405, 675 and 945

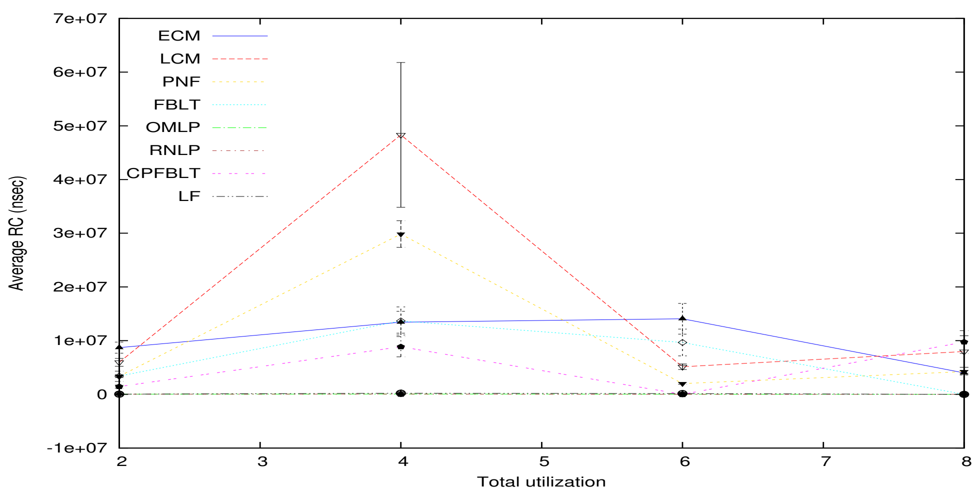


Figure C.136: Avg_RC for Tasksets 136, 406, 676 and 946

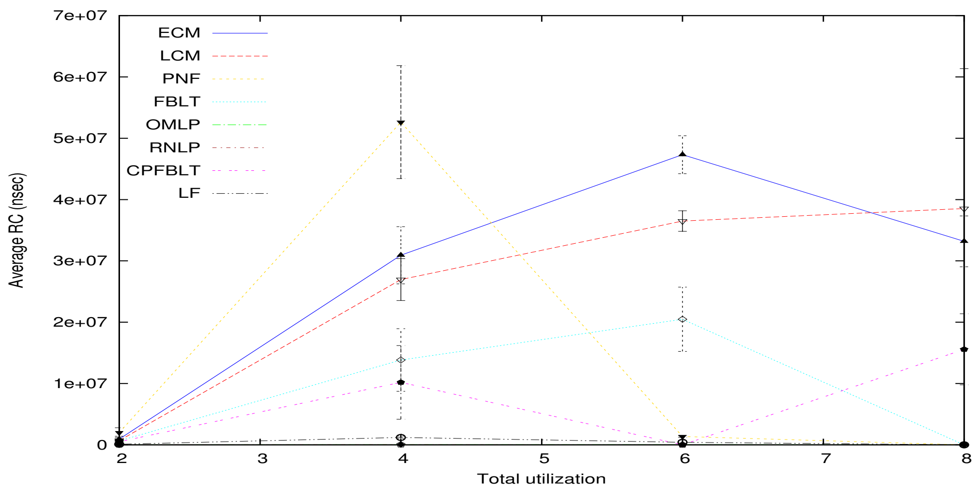


Figure C.137: Avg_RC for Tasksets 137, 407, 677 and 947

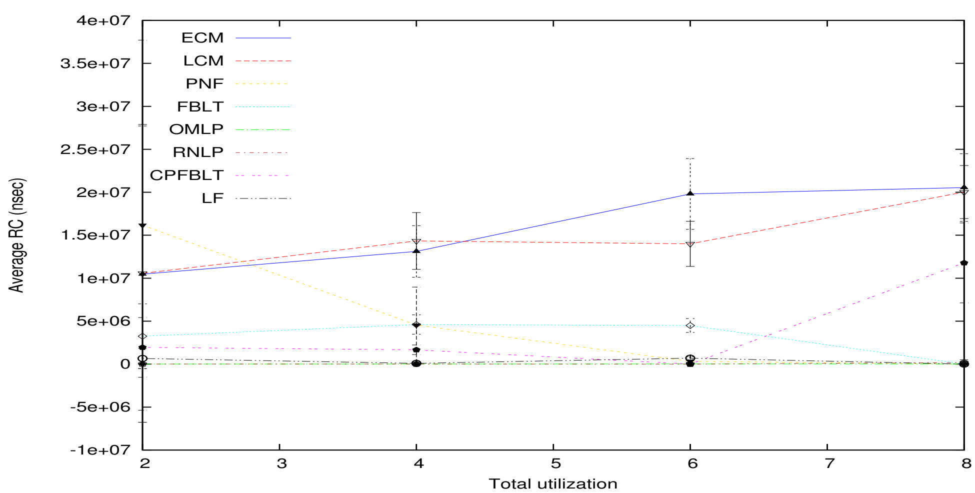


Figure C.138: Avg_RC for Tasksets 138, 408, 678 and 948

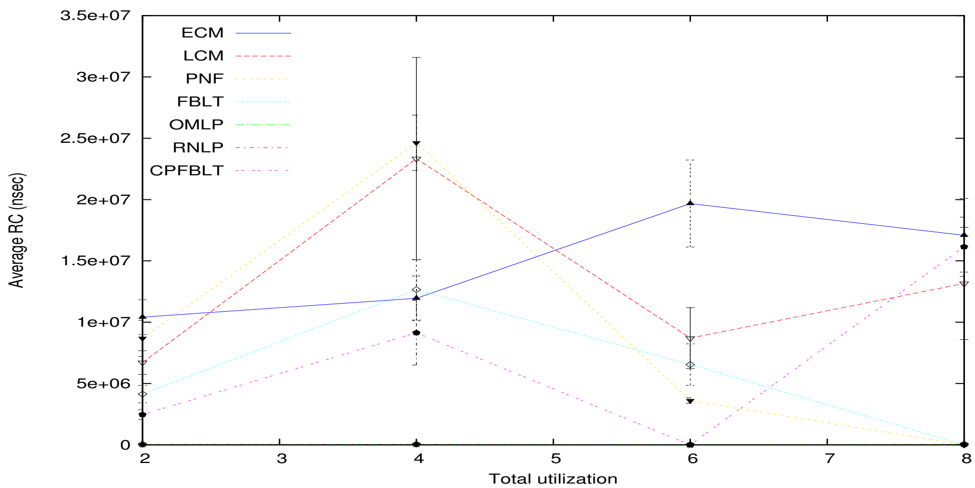


Figure C.139: Avg_RC for Tasksets 139, 409, 679 and 949

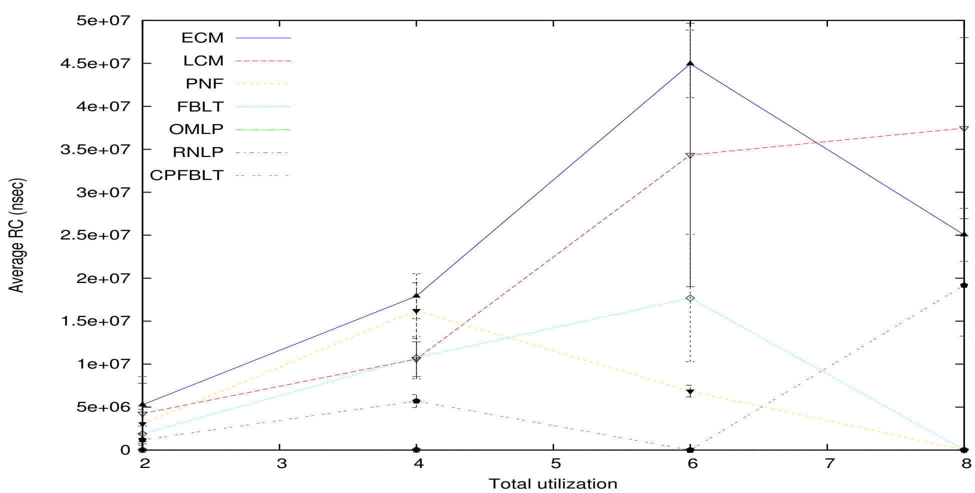


Figure C.140: Avg_RC for Tasksets 140, 410, 680 and 950

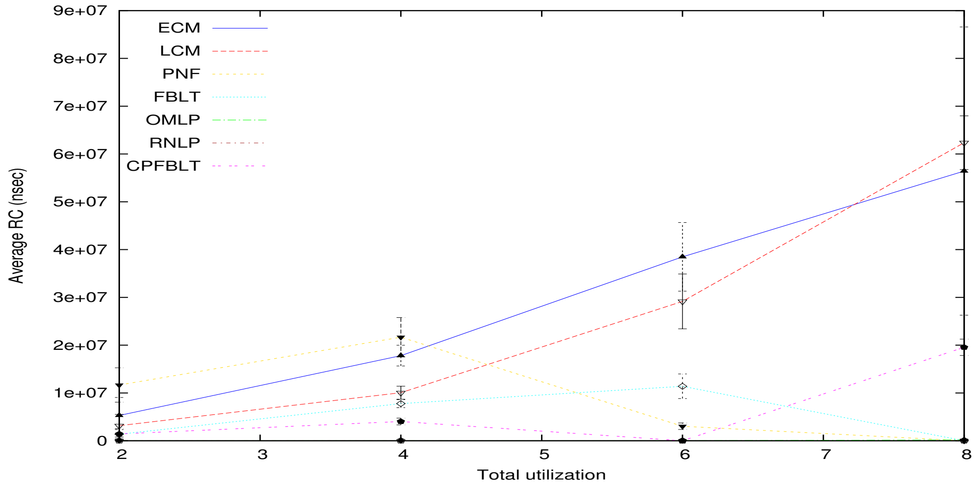


Figure C.141: Avg_RC for Tasksets 141, 411, 681 and 951

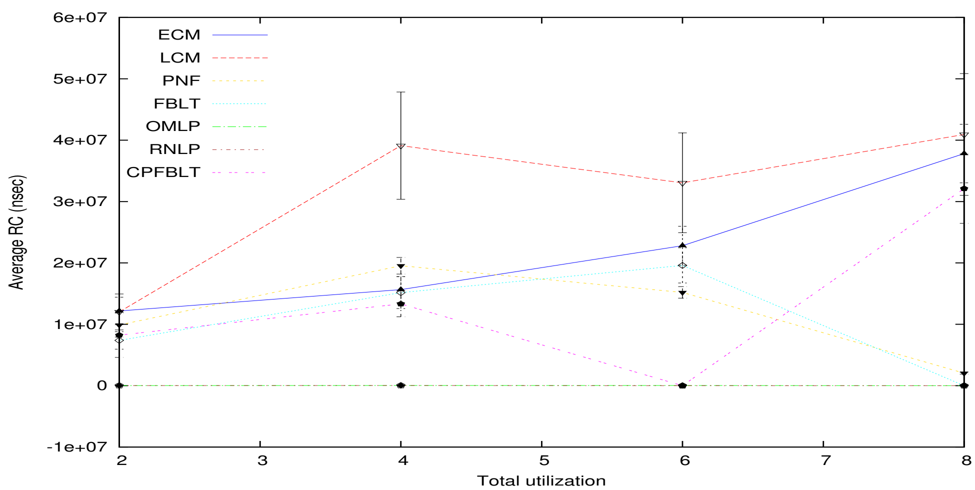


Figure C.142: Avg_RC for Tasksets 142, 412, 682 and 952

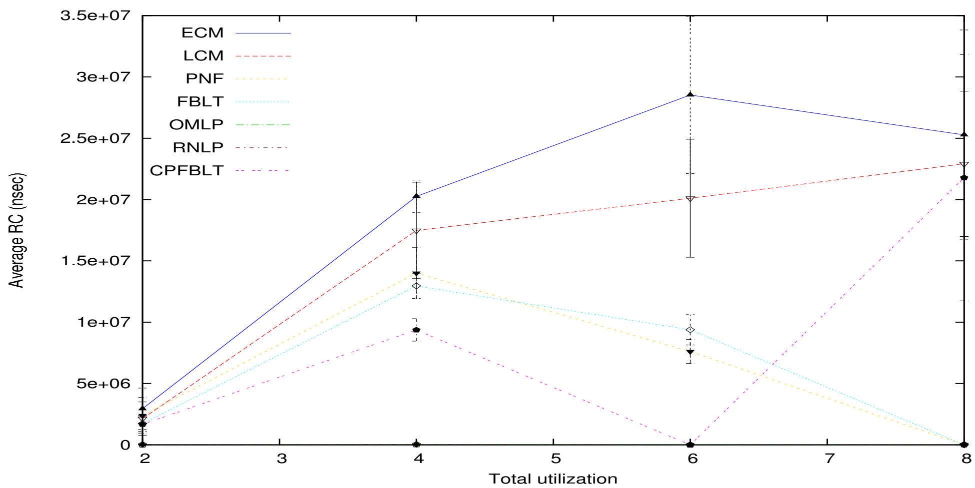


Figure C.143: Avg_RC for Tasksets 143, 413, 683 and 953

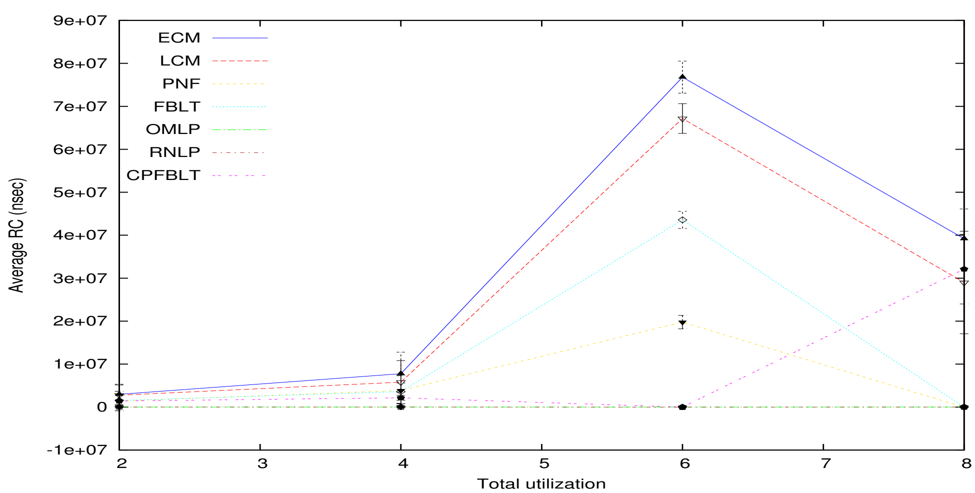


Figure C.144: Avg_RC for Tasksets 144, 414, 684 and 954

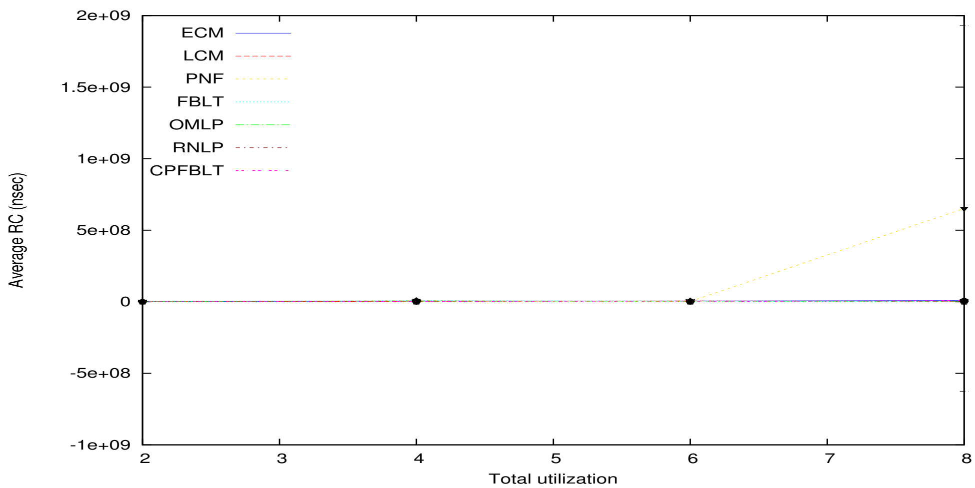


Figure C.145: Avg_RC for Tasksets 145, 415, 685 and 955

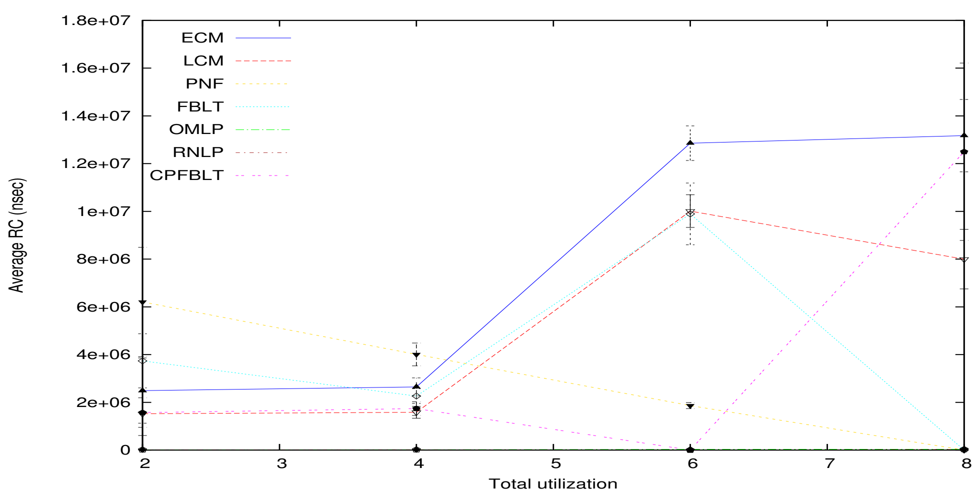


Figure C.146: Avg_RC for Tasksets 146, 416, 686 and 956

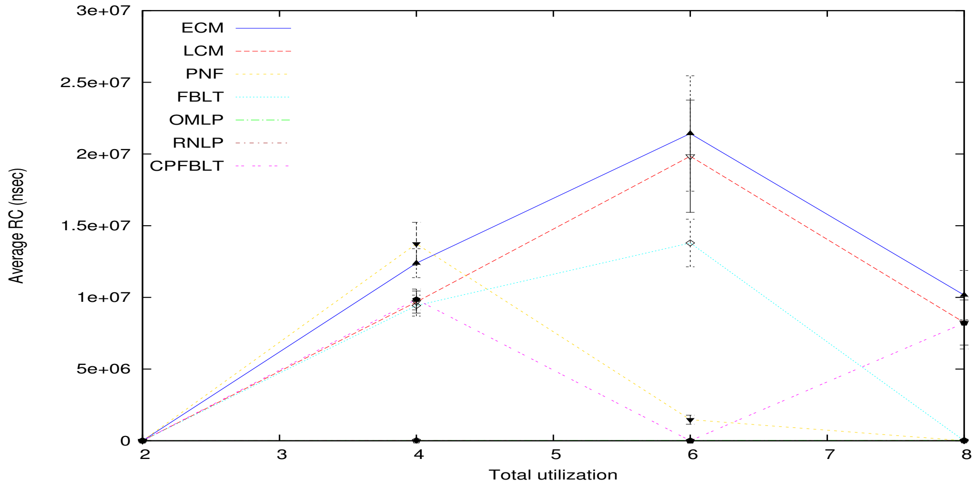


Figure C.147: Avg_RC for Tasksets 147, 417, 687 and 957

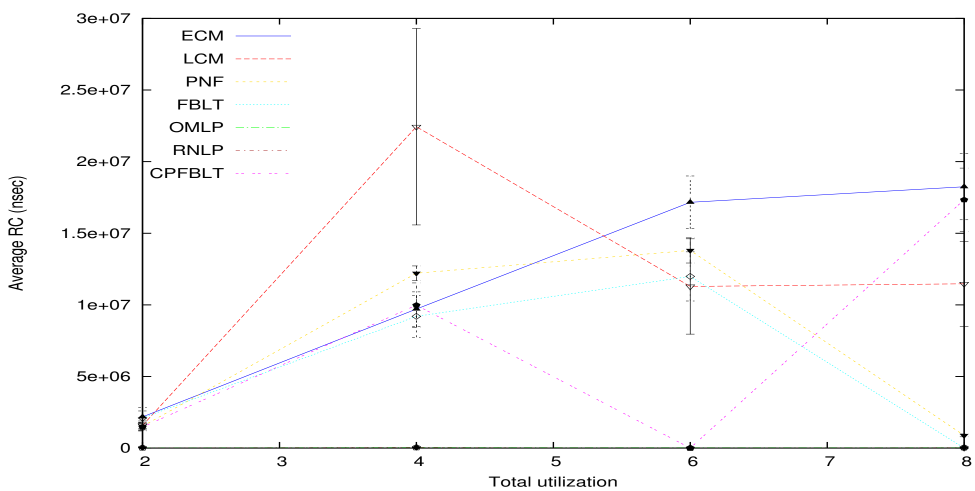


Figure C.148: Avg_RC for Tasksets 148, 418, 688 and 958

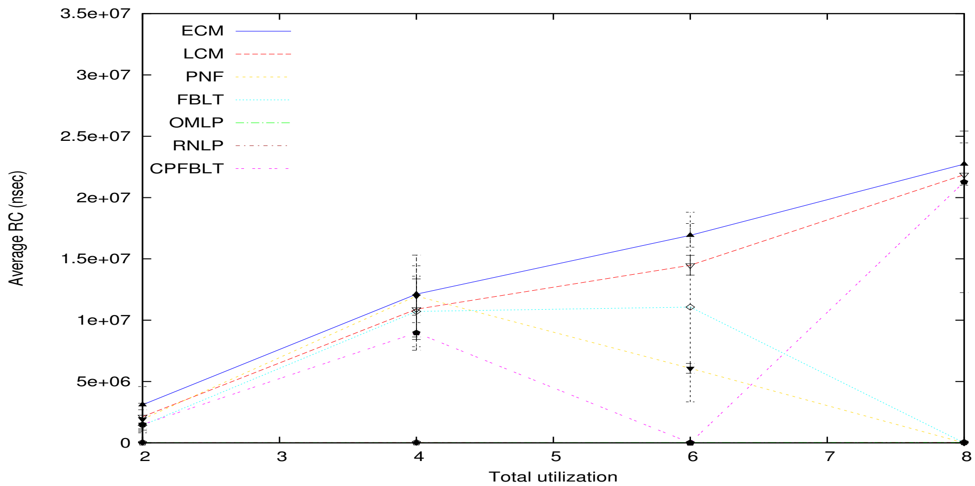


Figure C.149: Avg_RC for Tasksets 149, 419, 689 and 959

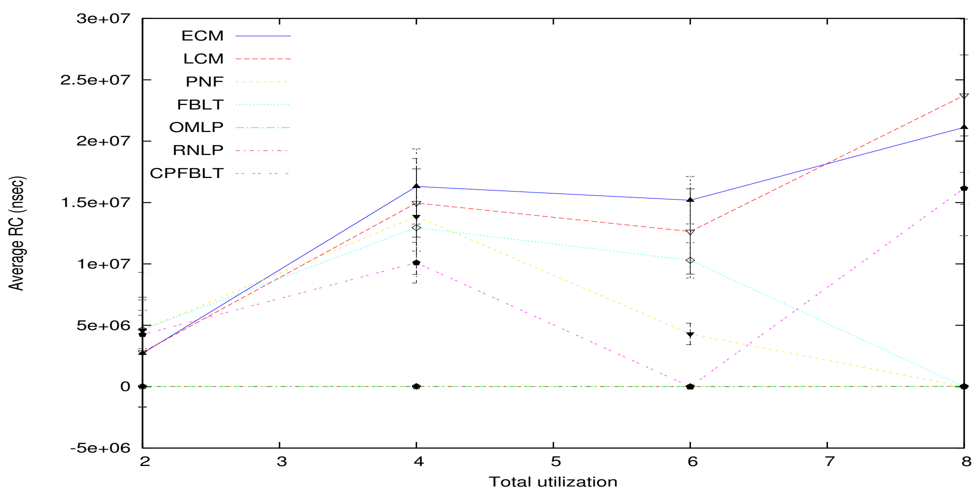


Figure C.150: Avg_RC for Tasksets 150, 420, 690 and 960

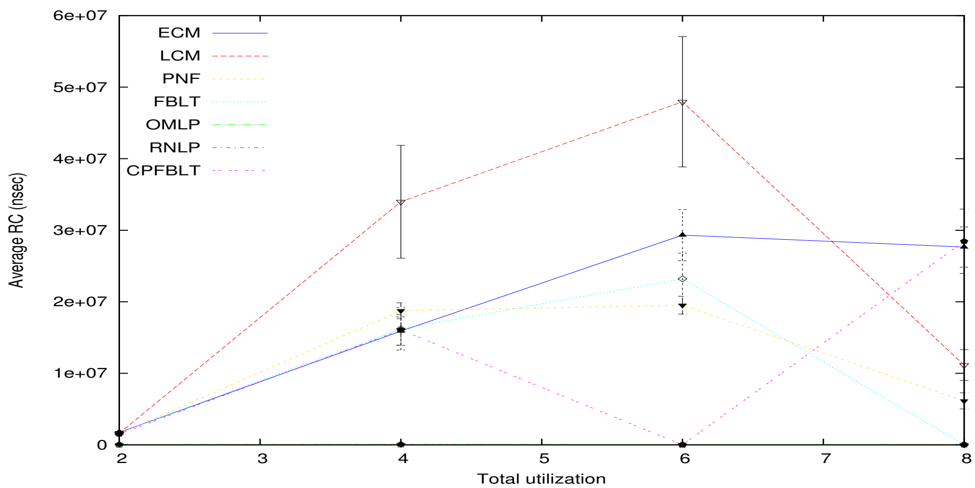


Figure C.151: Avg_RC for Tasksets 151, 421, 691 and 961

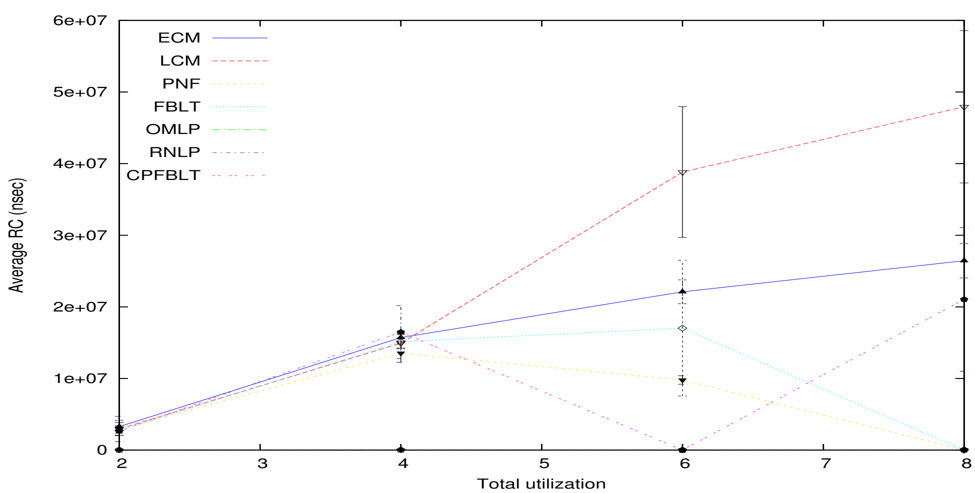


Figure C.152: Avg_RC for Tasksets 152, 422, 692 and 962

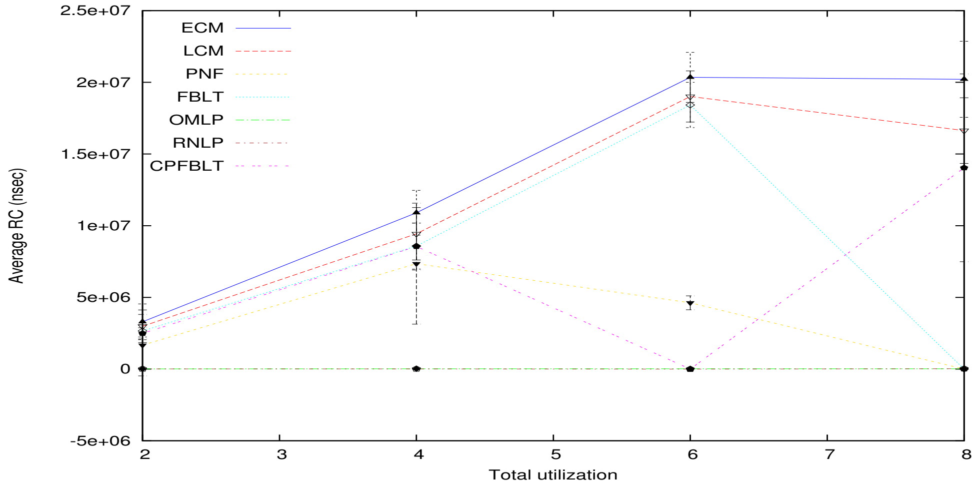


Figure C.153: Avg_RC for Tasksets 153, 423, 693 and 963

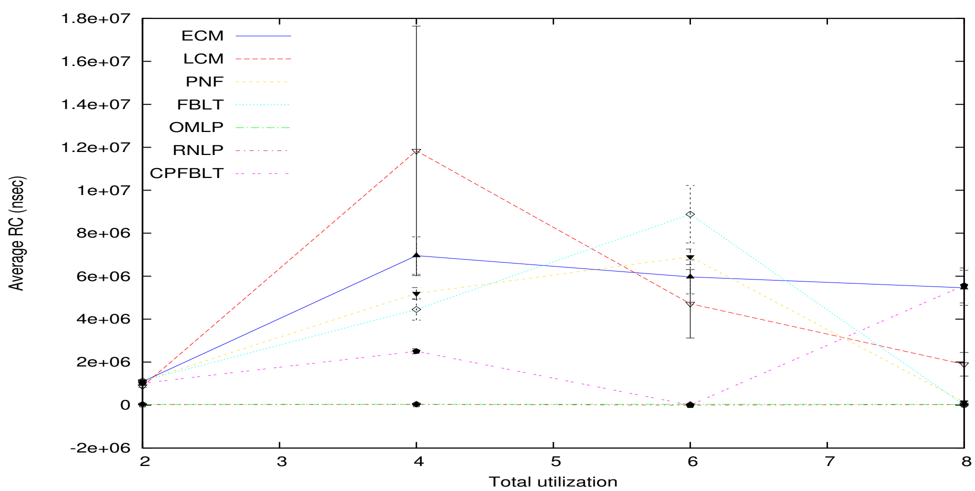


Figure C.154: Avg_RC for Tasksets 154, 424, 694 and 964

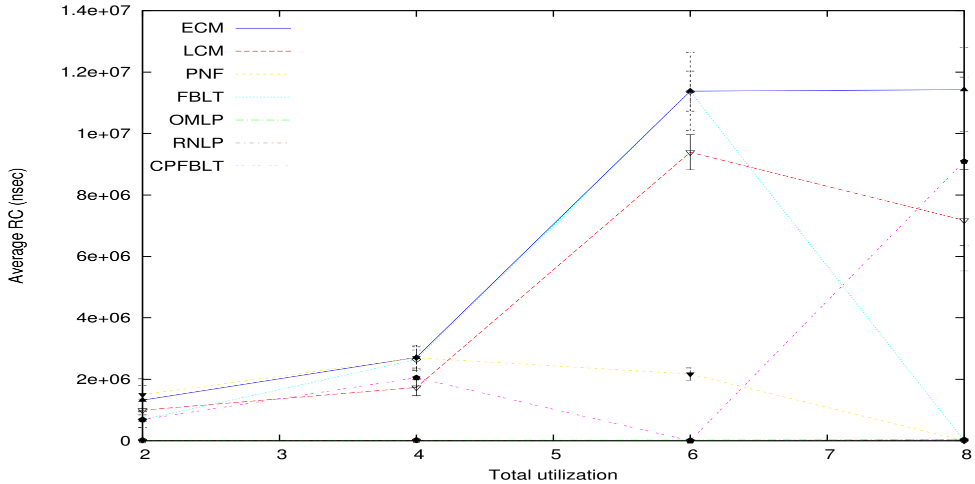


Figure C.155: Avg_RC for Tasksets 155, 425, 695 and 965

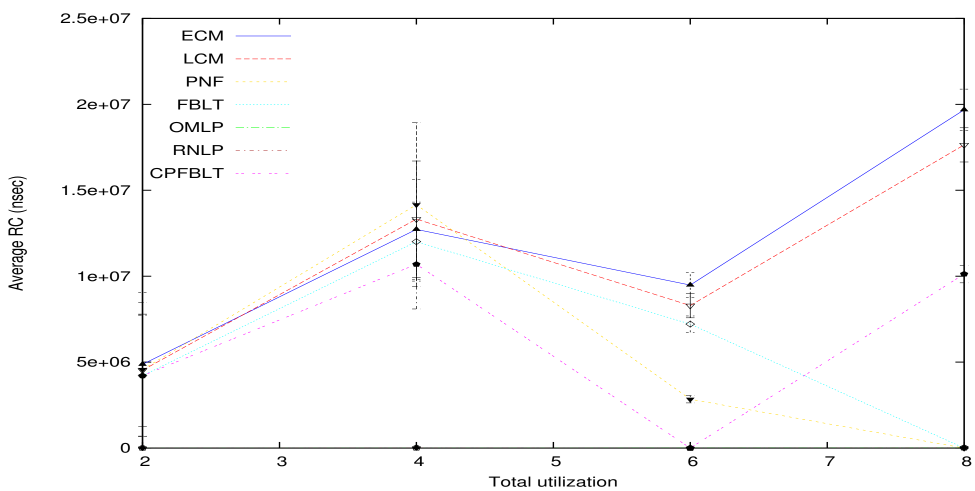


Figure C.156: Avg_RC for Tasksets 156, 426, 696 and 966

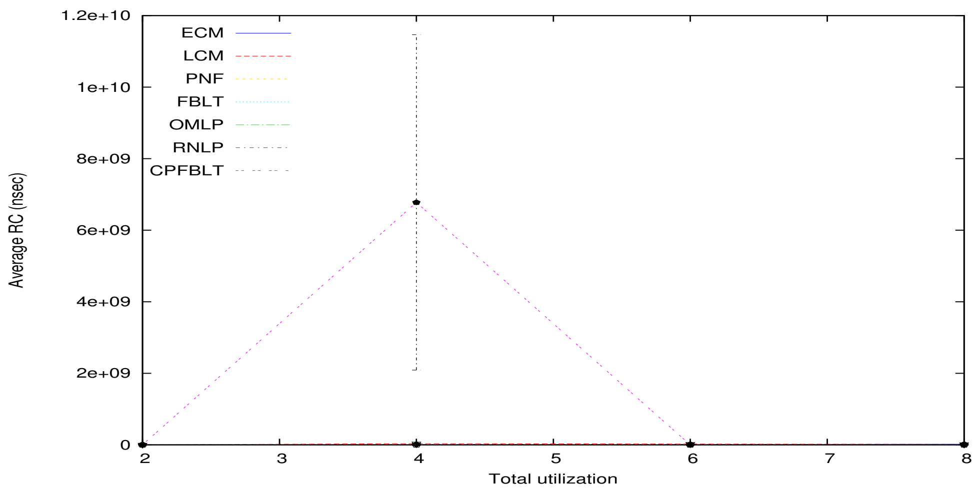


Figure C.157: Avg_RC for Tasksets 157, 427, 697 and 967

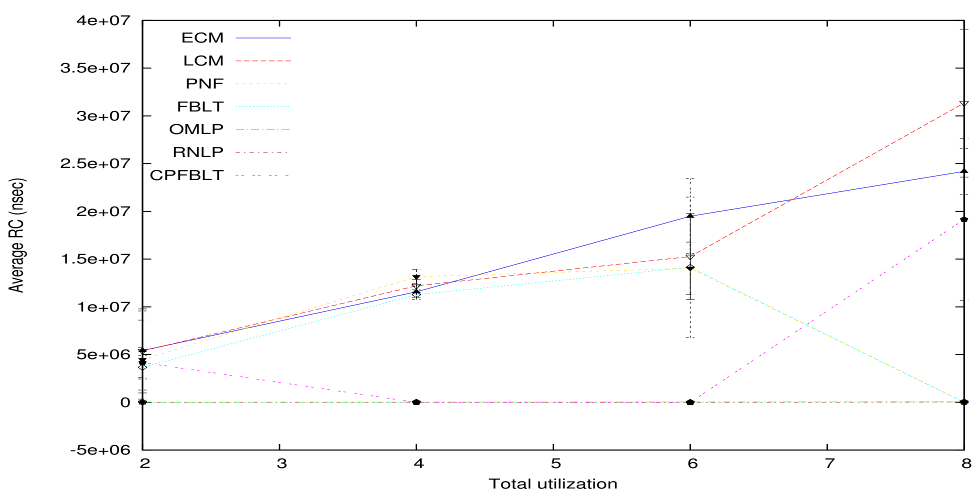


Figure C.158: Avg_RC for Tasksets 158, 428, 698 and 968

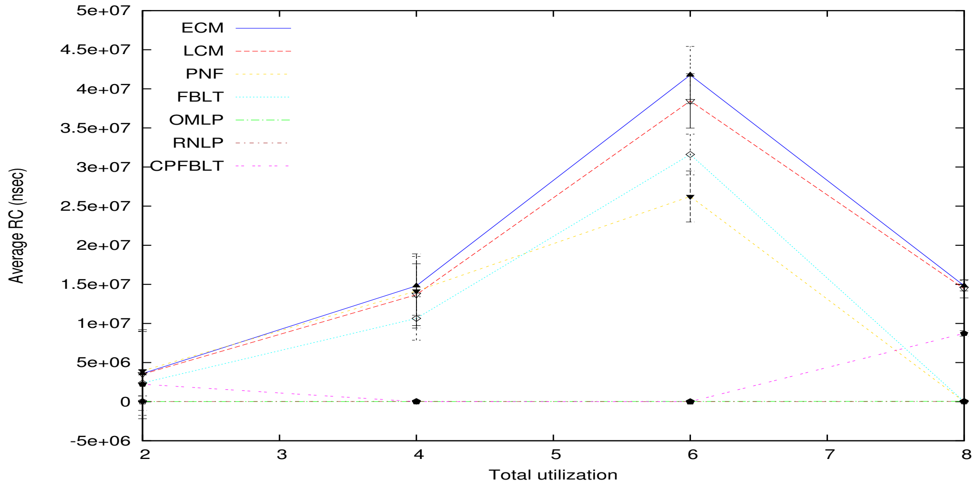


Figure C.159: Avg_RC for Tasksets 159, 429, 699 and 969

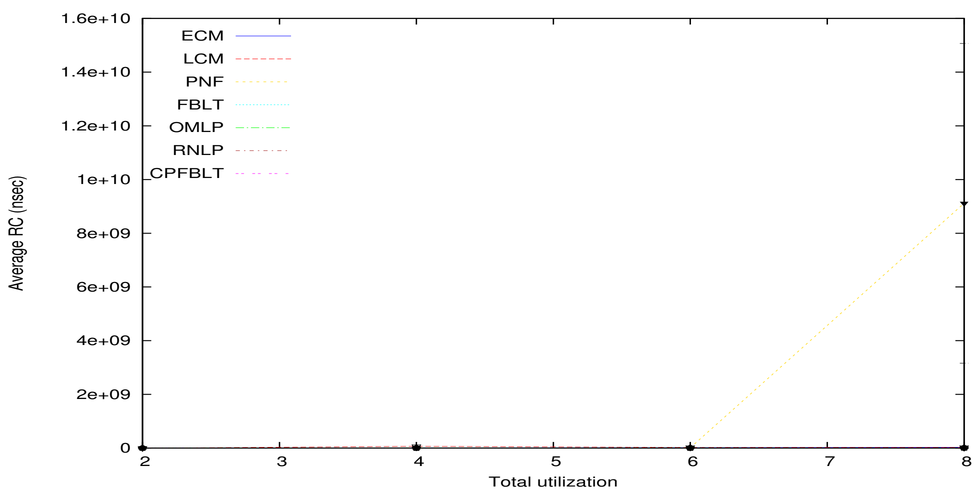


Figure C.160: Avg_RC for Tasksets 160, 430, 700 and 970

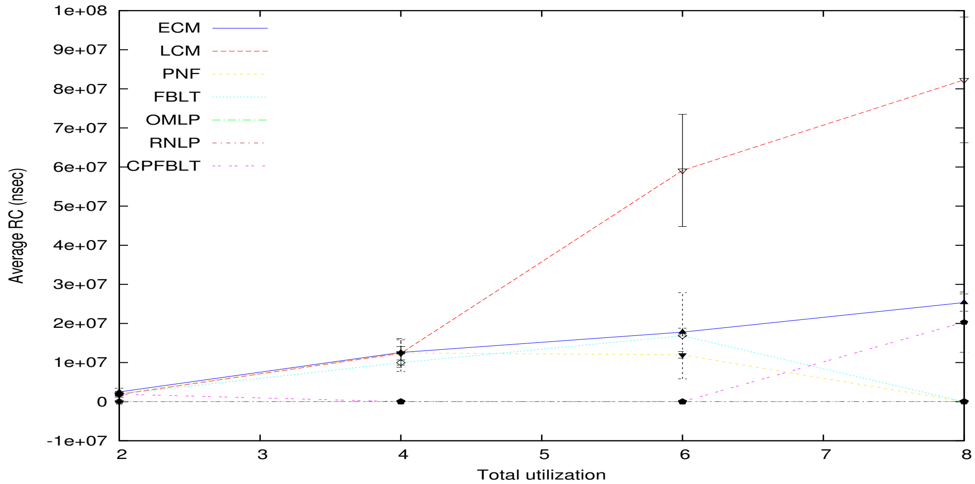


Figure C.161: Avg_RC for Tasksets 161, 431, 701 and 971

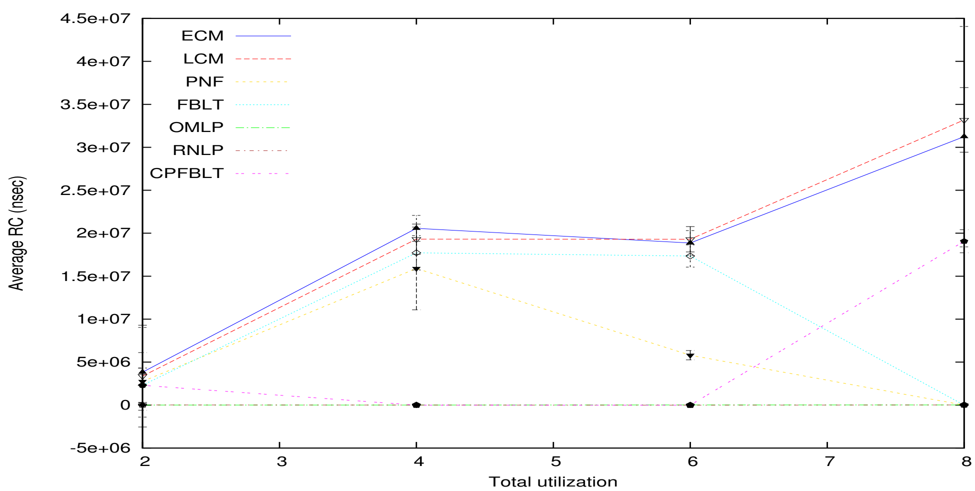


Figure C.162: Avg_RC for Tasksets 162, 432, 702 and 972

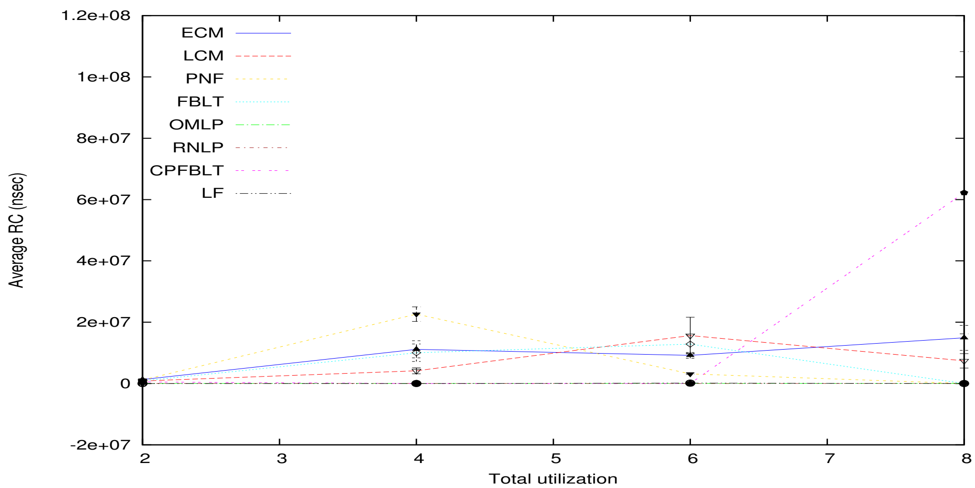


Figure C.163: Avg_RC for Tasksets 163, 433, 703 and 973

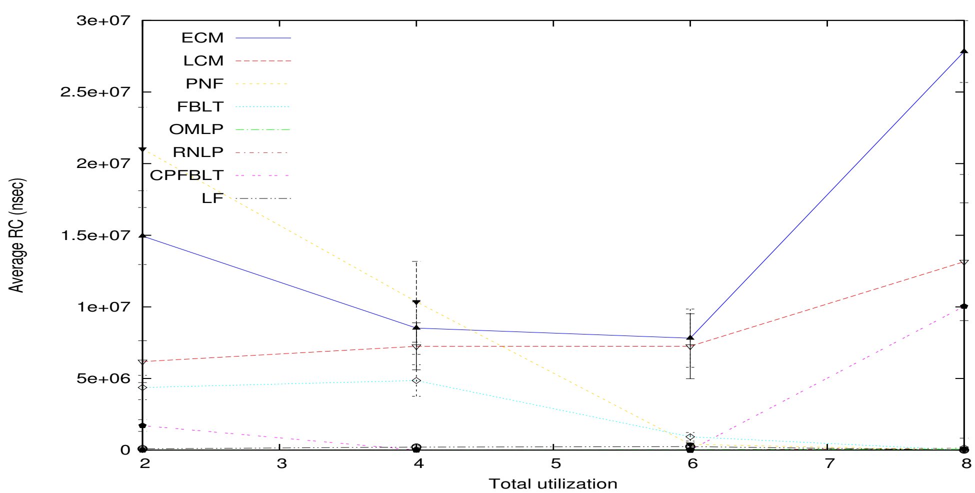


Figure C.164: Avg_RC for Tasksets 164, 434, 704 and 974

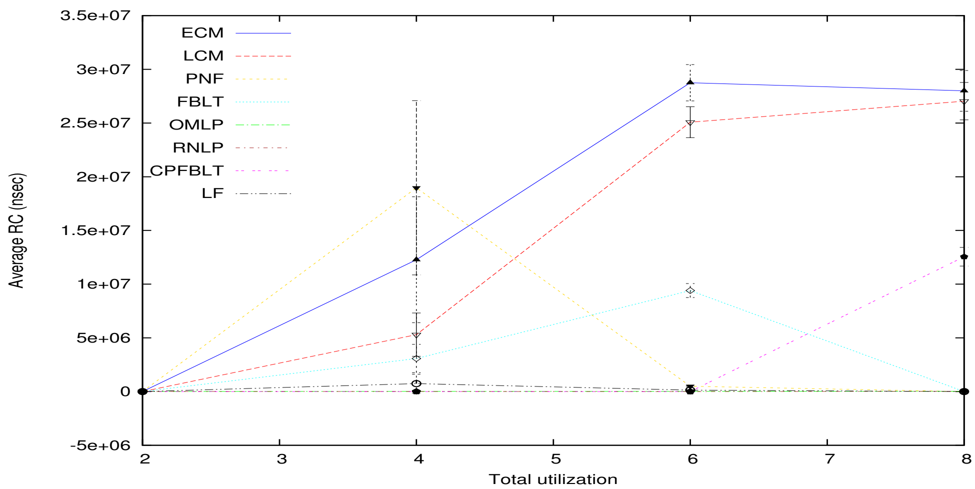


Figure C.165: Avg_RC for Tasksets 165, 435, 705 and 975

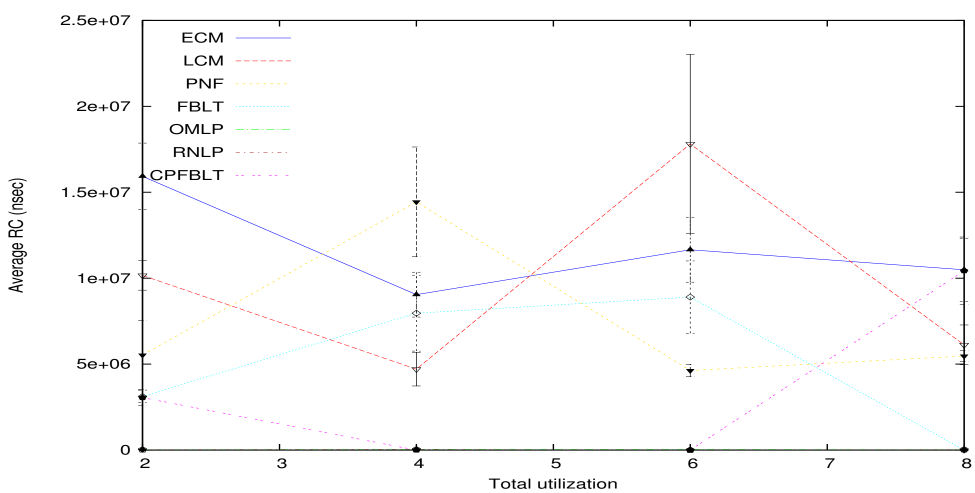


Figure C.166: Avg_RC for Tasksets 166, 436, 706 and 976

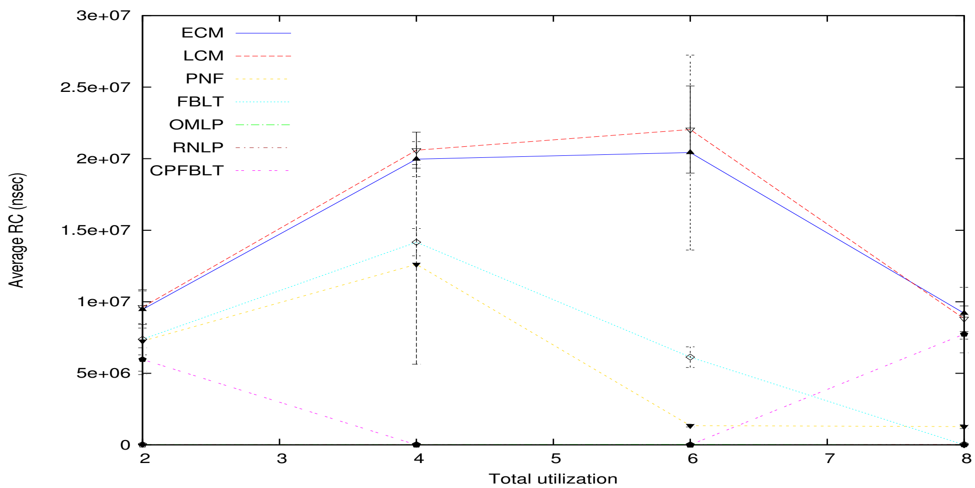


Figure C.167: Avg_RC for Tasksets 167, 437, 707 and 977

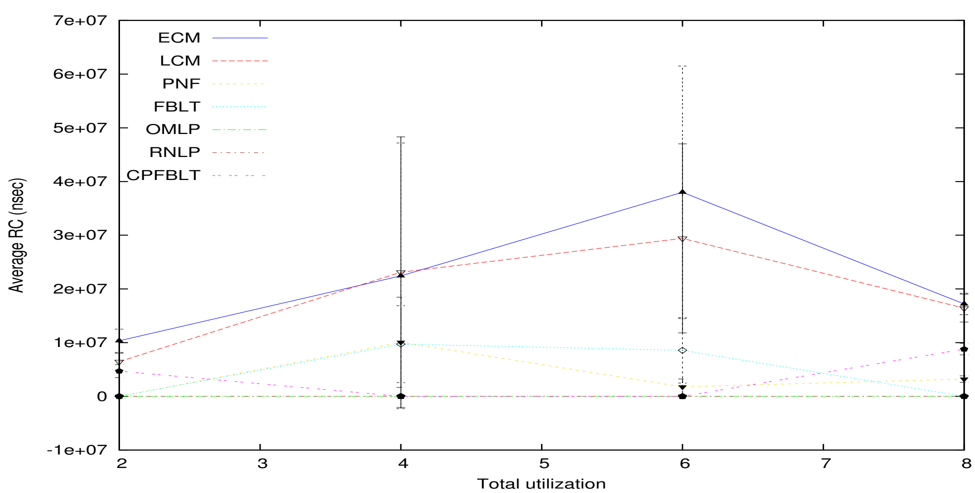


Figure C.168: Avg_RC for Tasksets 168, 438, 708 and 978

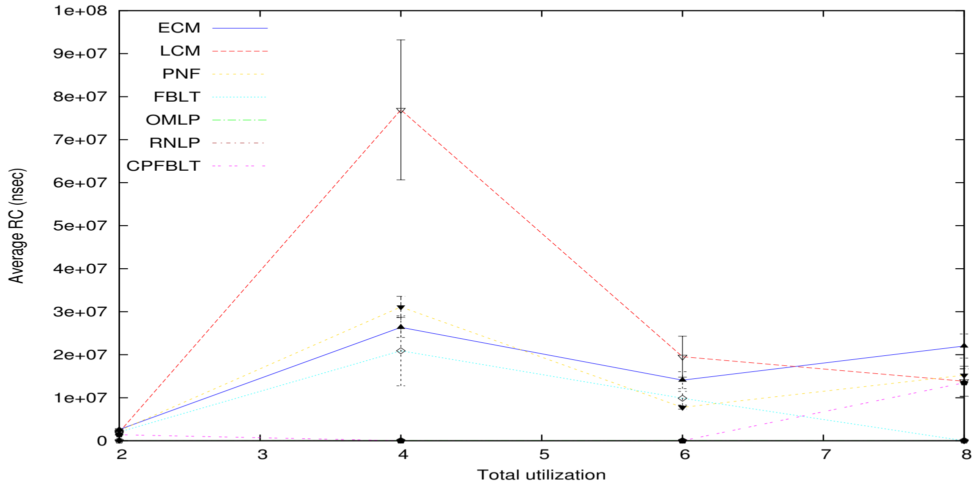


Figure C.169: Avg_RC for Tasksets 169, 439, 709 and 979

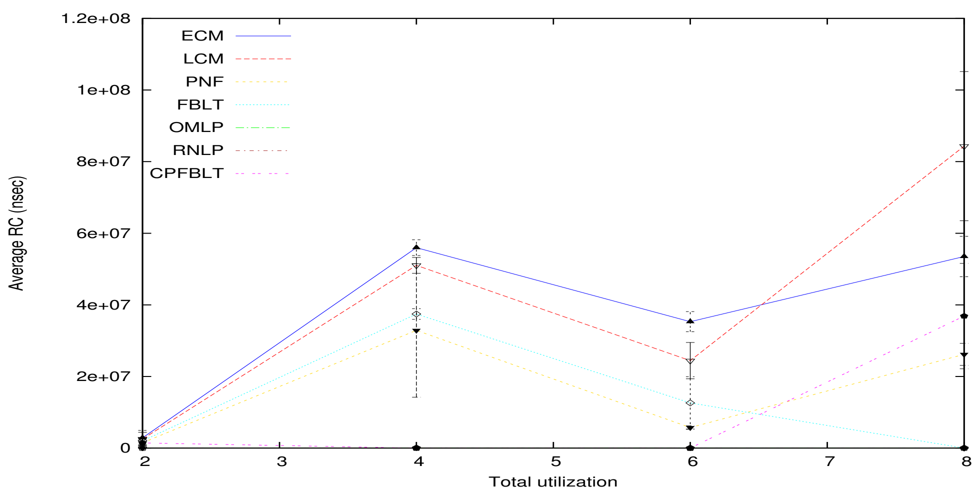


Figure C.170: Avg_RC for Tasksets 170, 440, 710 and 980

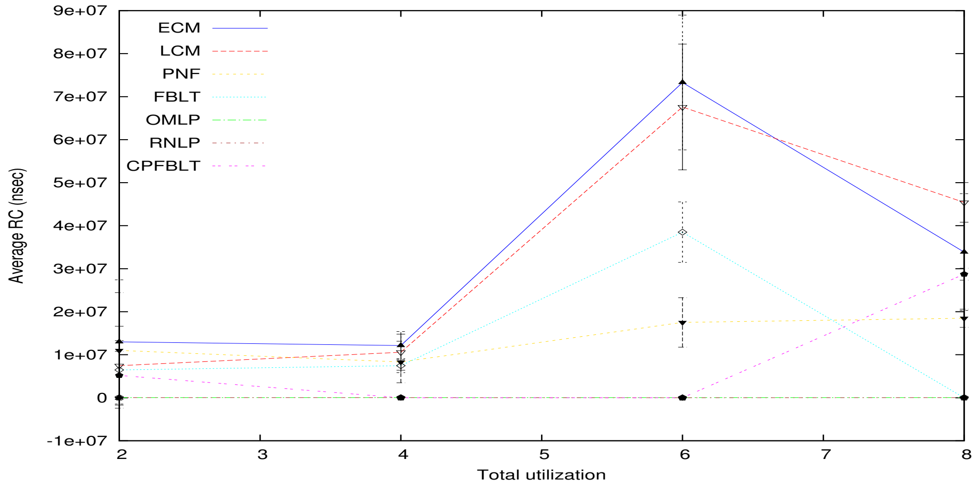


Figure C.171: Avg_RC for Tasksets 171, 441, 711 and 981

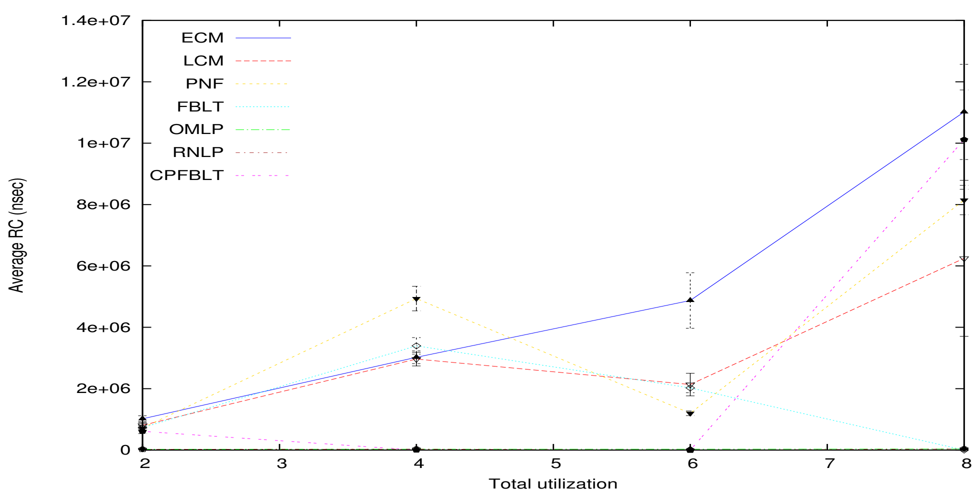


Figure C.172: Avg_RC for Tasksets 172, 442, 712 and 982

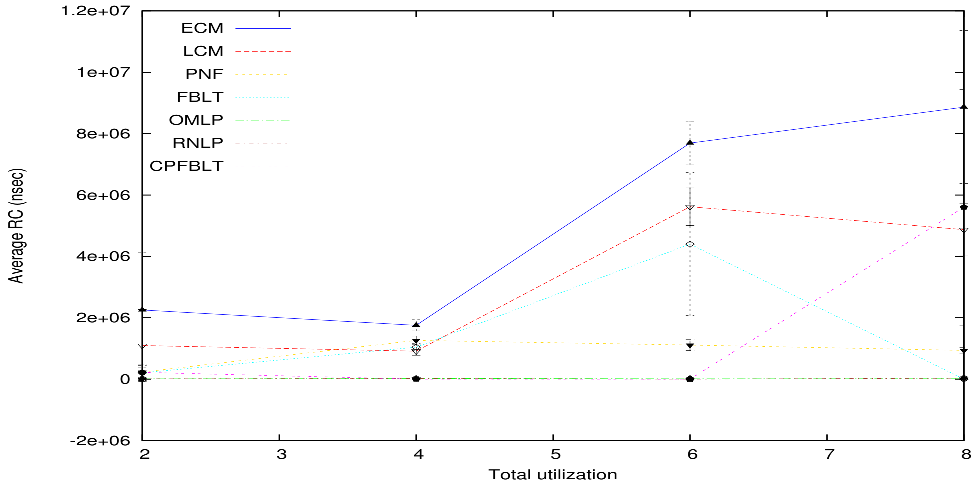


Figure C.173: Avg_RC for Tasksets 173, 443, 713 and 983

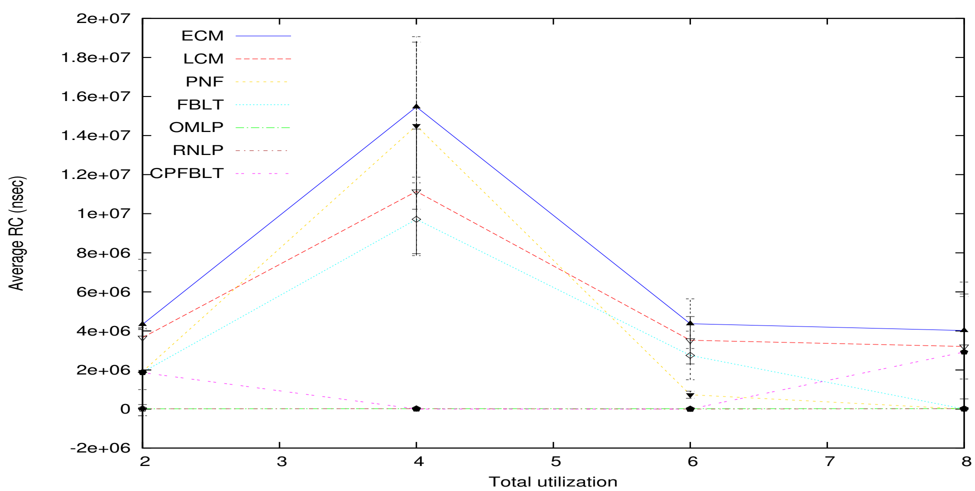


Figure C.174: Avg_RC for Tasksets 174, 444, 714 and 984

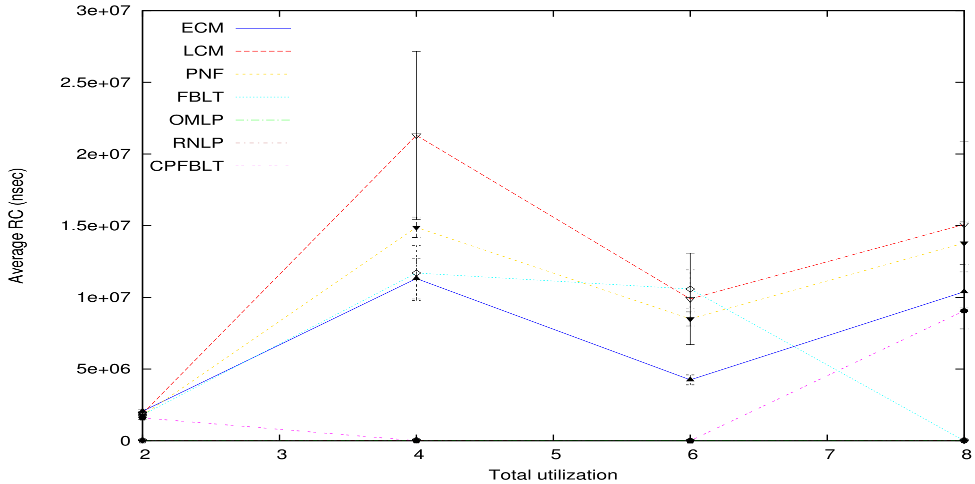


Figure C.175: Avg_RC for Tasksets 175, 445, 715 and 985

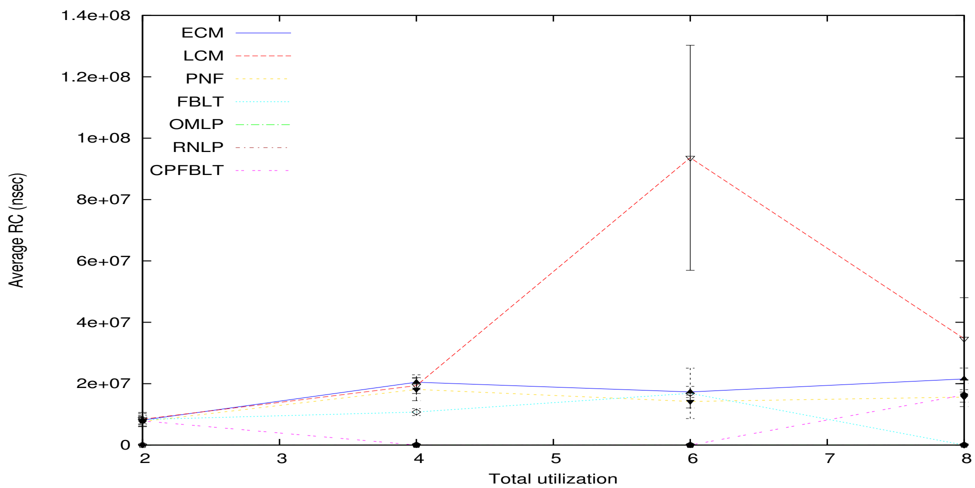


Figure C.176: Avg_RC for Tasksets 176, 446, 716 and 986

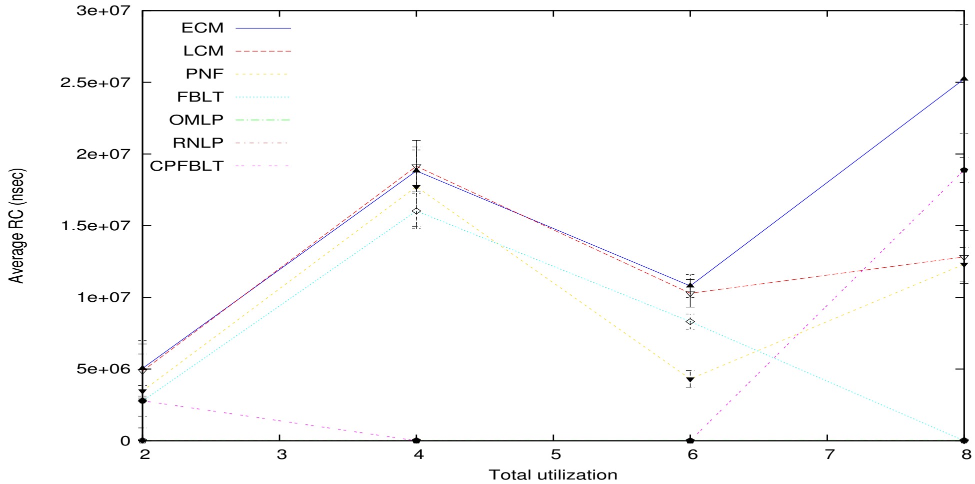


Figure C.177: Avg_RC for Tasksets 177, 447, 717 and 987

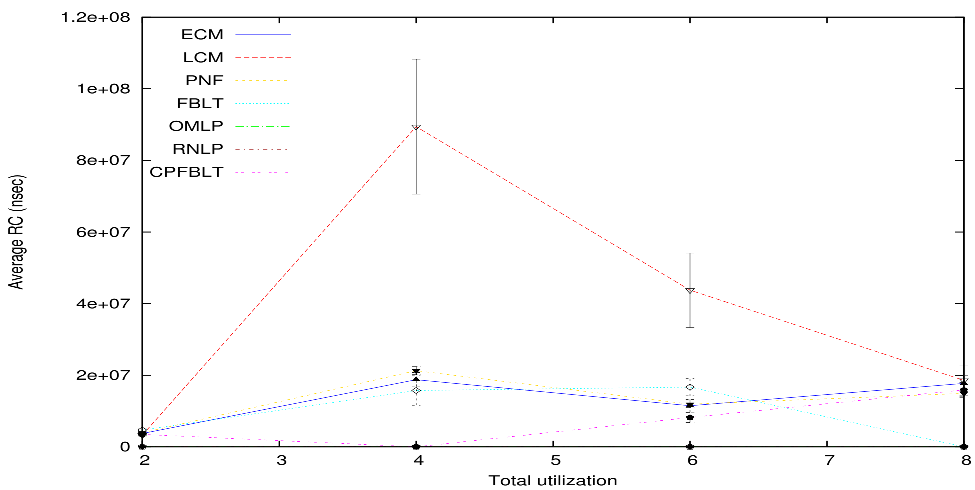


Figure C.178: Avg_RC for Tasksets 178, 448, 718 and 988

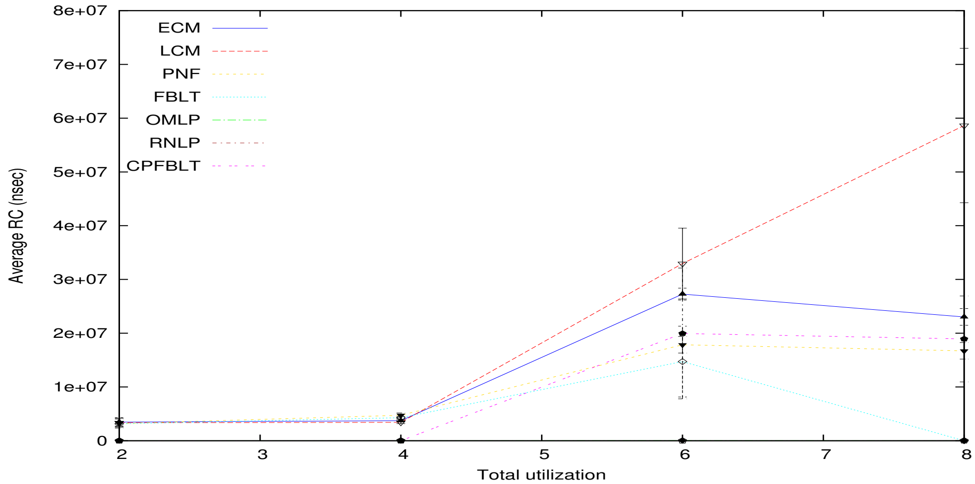


Figure C.179: Avg_RC for Tasksets 179, 449, 719 and 989

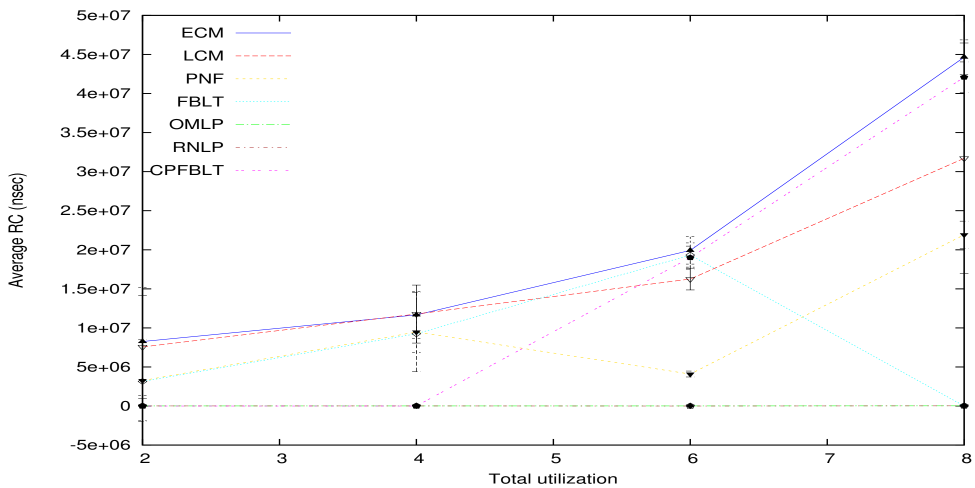


Figure C.180: Avg_RC for Tasksets 180, 450, 720 and 990

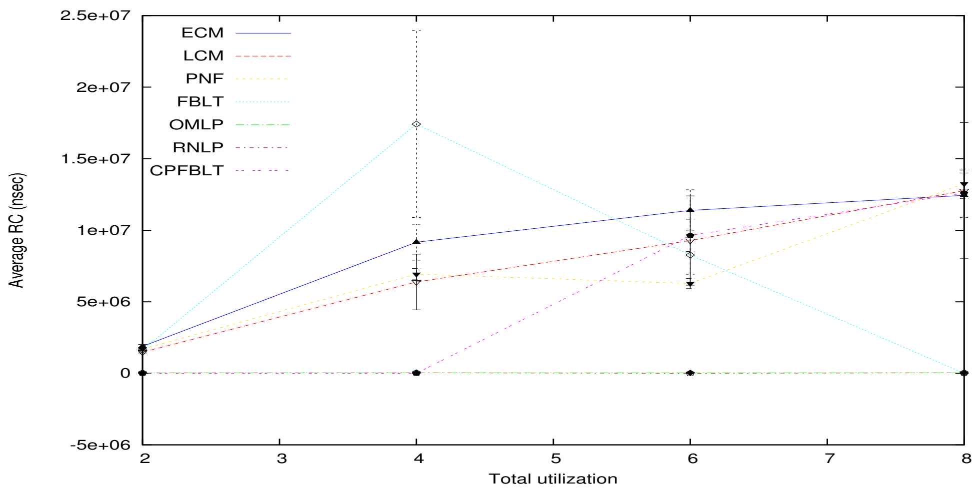


Figure C.181: Avg_RC for Tasksets 181, 451, 721 and 991

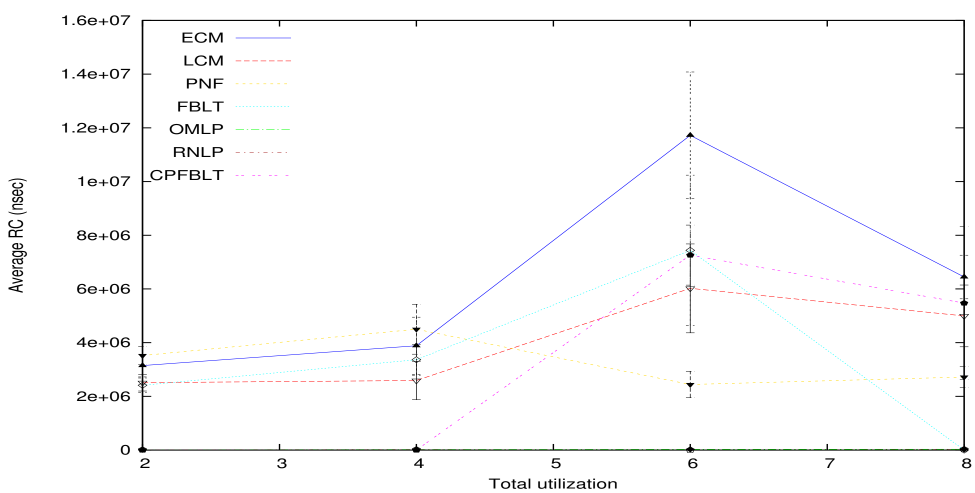


Figure C.182: Avg_RC for Tasksets 182, 452, 722 and 992

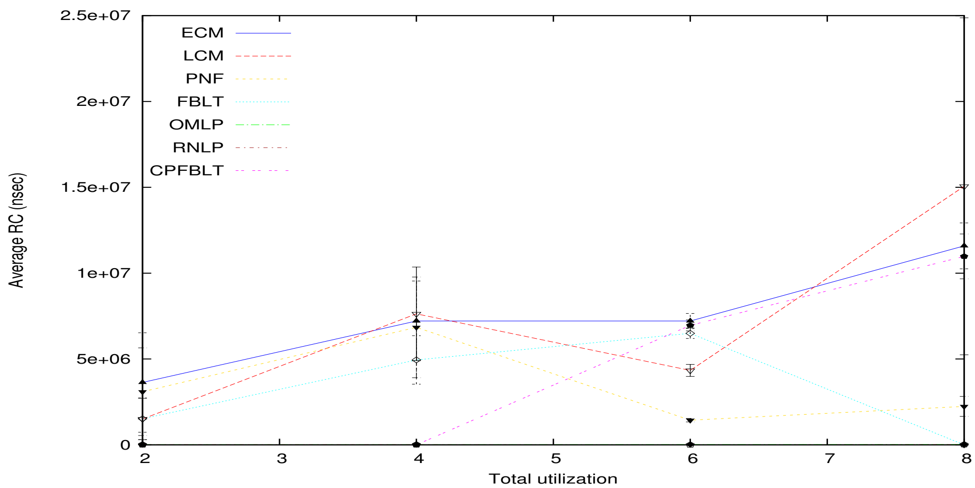


Figure C.183: Avg_RC for Tasksets 183, 453, 723 and 993

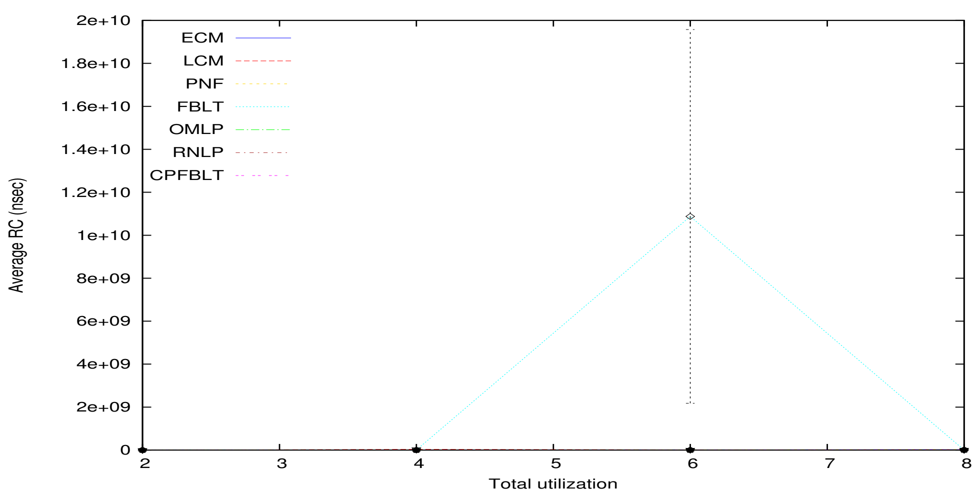


Figure C.184: Avg_RC for Tasksets 184, 454, 724 and 994

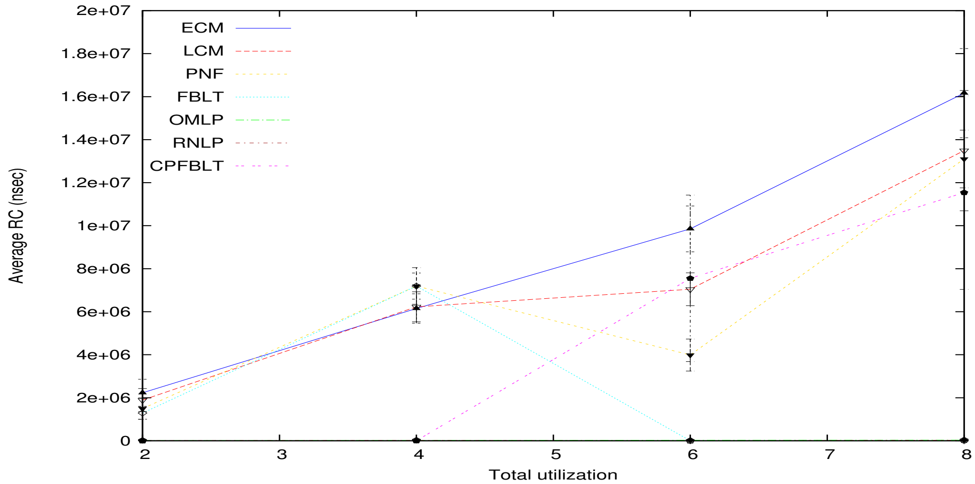


Figure C.185: Avg_RC for Tasksets 185, 455, 725 and 995

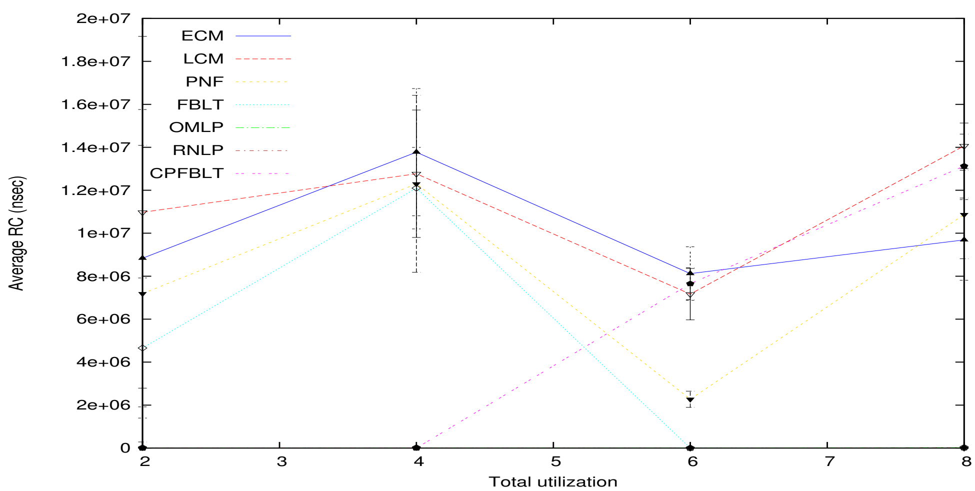


Figure C.186: Avg_RC for Tasksets 186, 456, 726 and 996

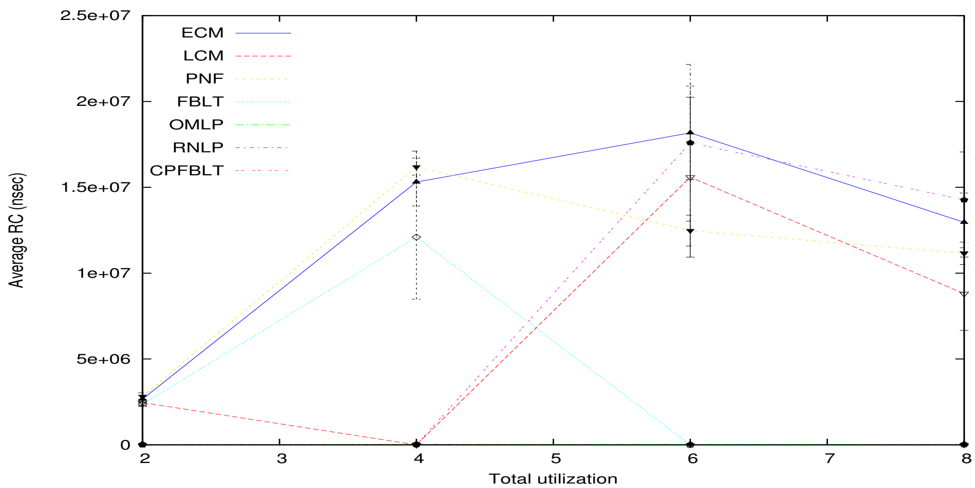


Figure C.187: Avg_RC for Tasksets 187, 457, 727 and 997

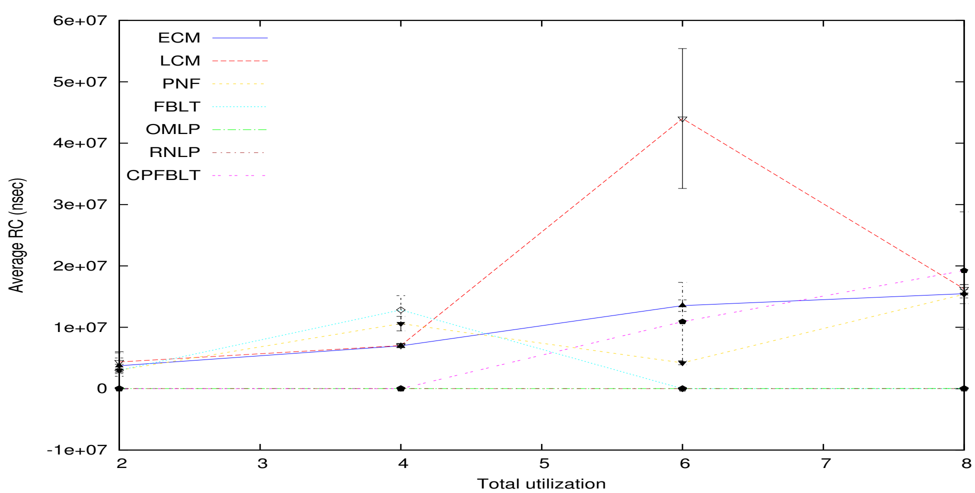


Figure C.188: Avg_RC for Tasksets 188, 458, 728 and 998

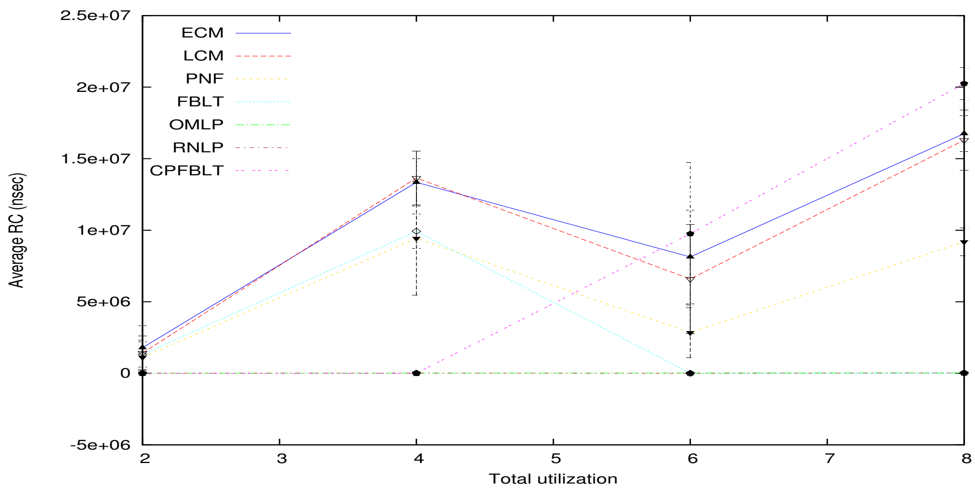


Figure C.189: Avg_RC for Tasksets 189, 459, 729 and 999

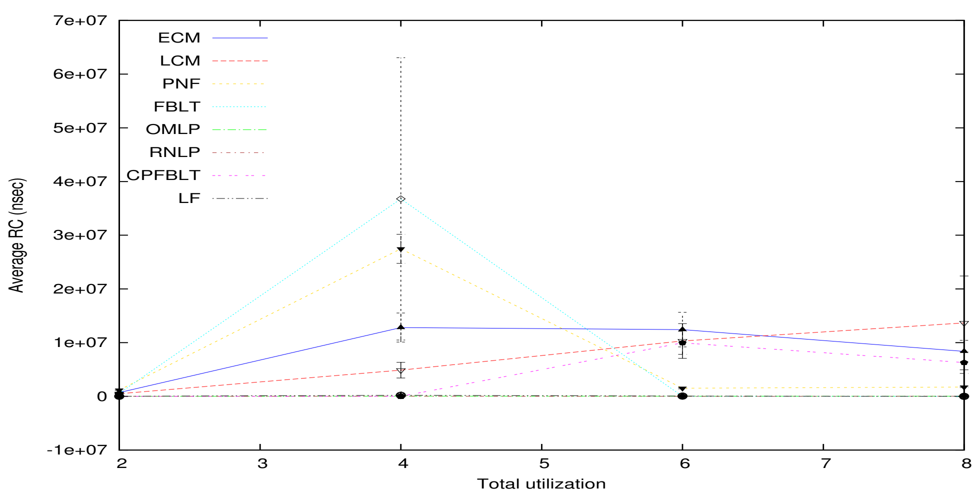


Figure C.190: Avg_RC for Tasksets 190, 460, 730 and 1000

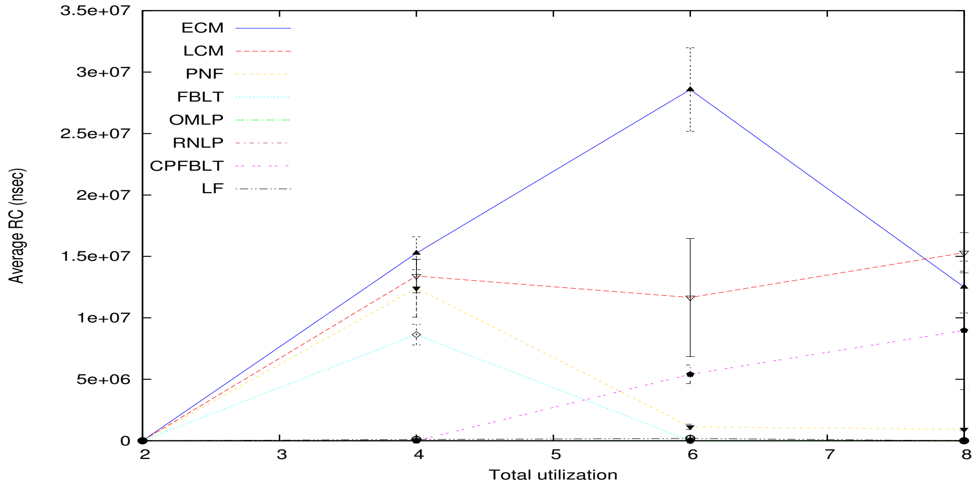


Figure C.191: Avg_RC for Tasksets 191, 461, 731 and 1001

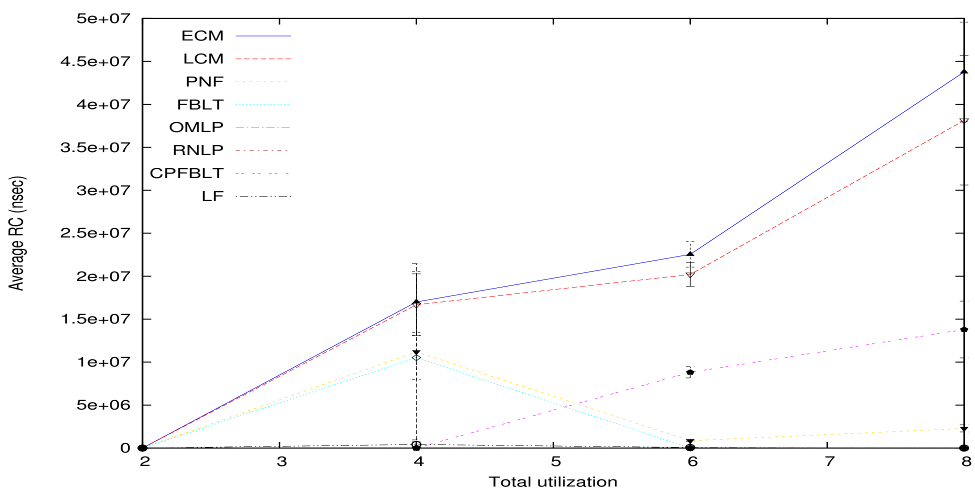


Figure C.192: Avg_RC for Tasksets 192, 462, 732 and 1002

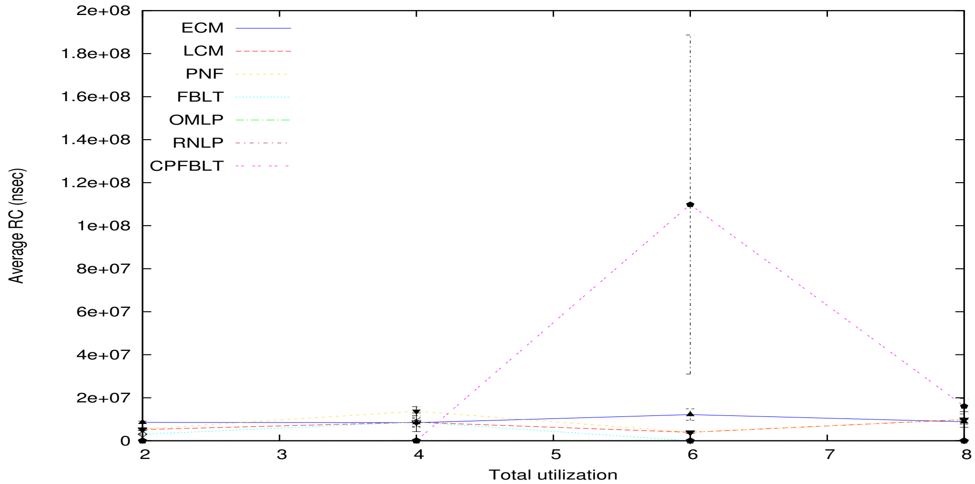


Figure C.193: Avg_RC for Tasksets 193, 463, 733 and 1003

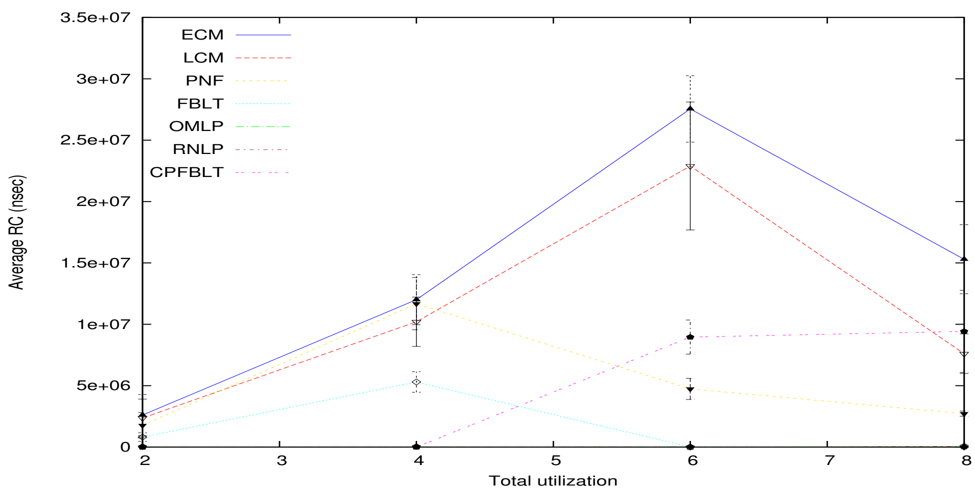


Figure C.194: Avg_RC for Tasksets 194, 464, 734 and 1004

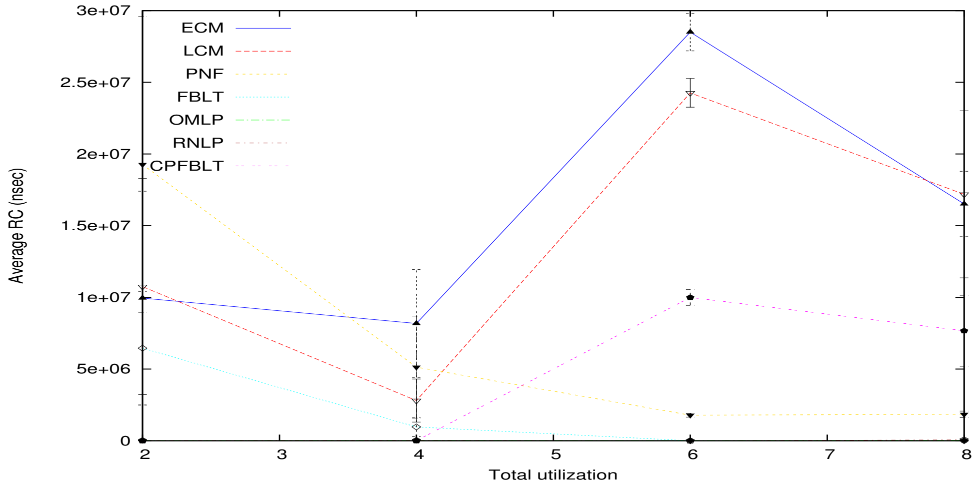


Figure C.195: Avg_RC for Tasksets 195, 465, 735 and 1005

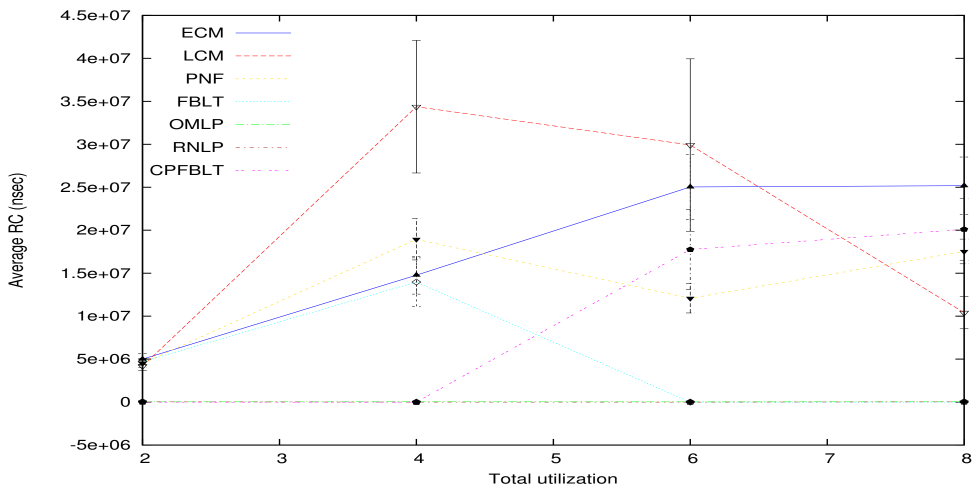


Figure C.196: Avg_RC for Tasksets 196, 466, 736 and 1006

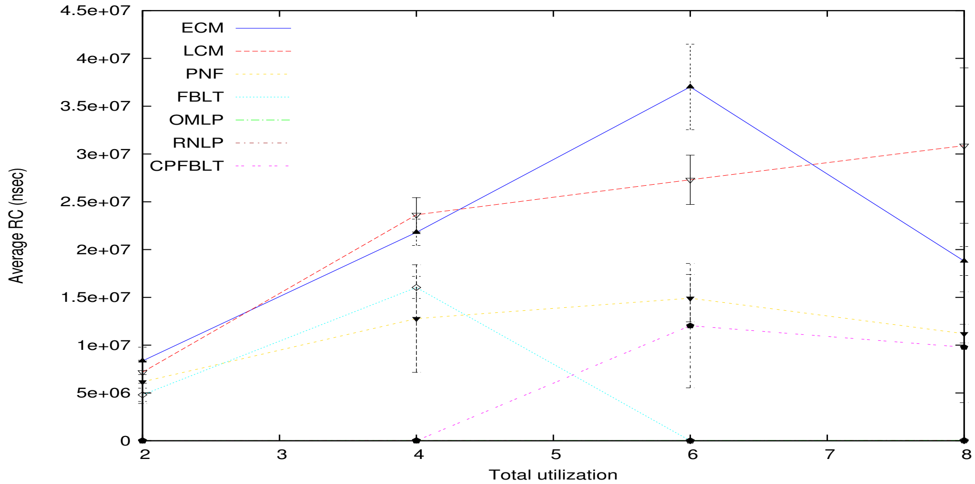


Figure C.197: Avg_RC for Tasksets 197, 467, 737 and 1007

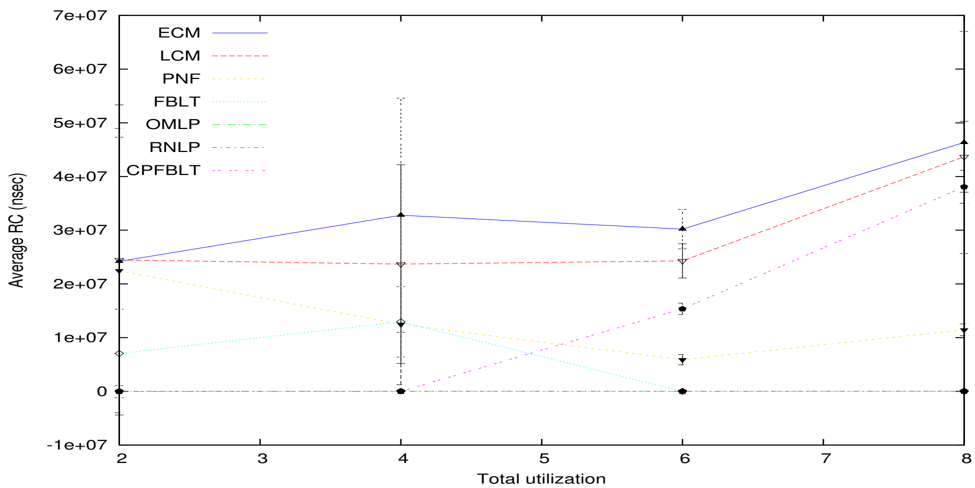


Figure C.198: Avg_RC for Tasksets 198, 468, 738 and 1008

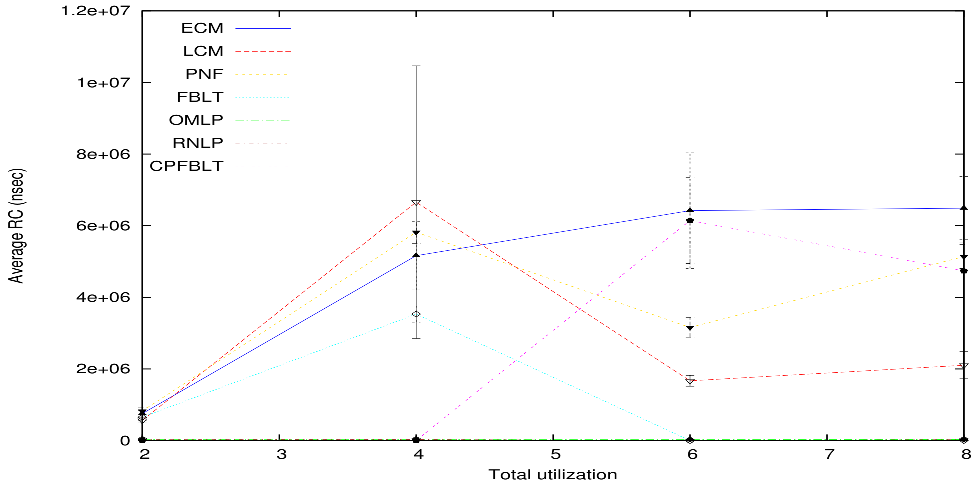


Figure C.199: Avg_RC for Tasksets 199, 469, 739 and 1009

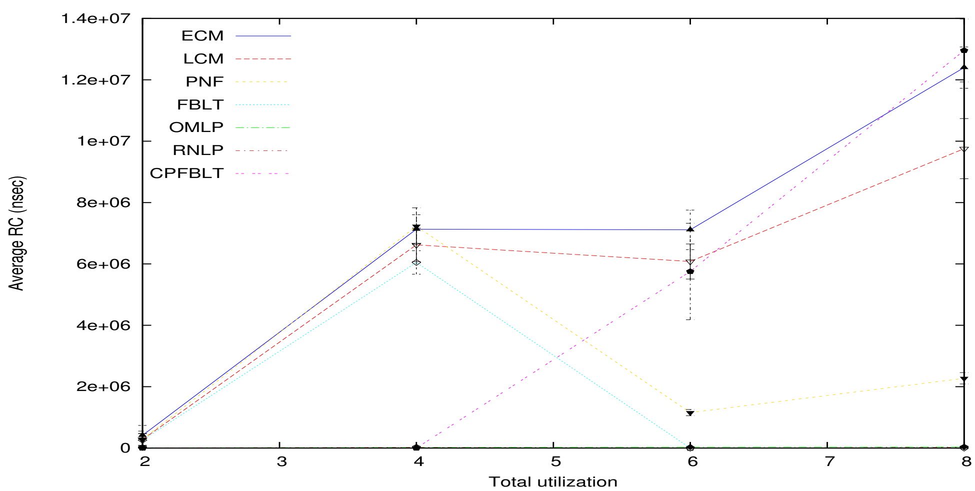


Figure C.200: Avg_RC for Tasksets 200, 470, 740 and 1010

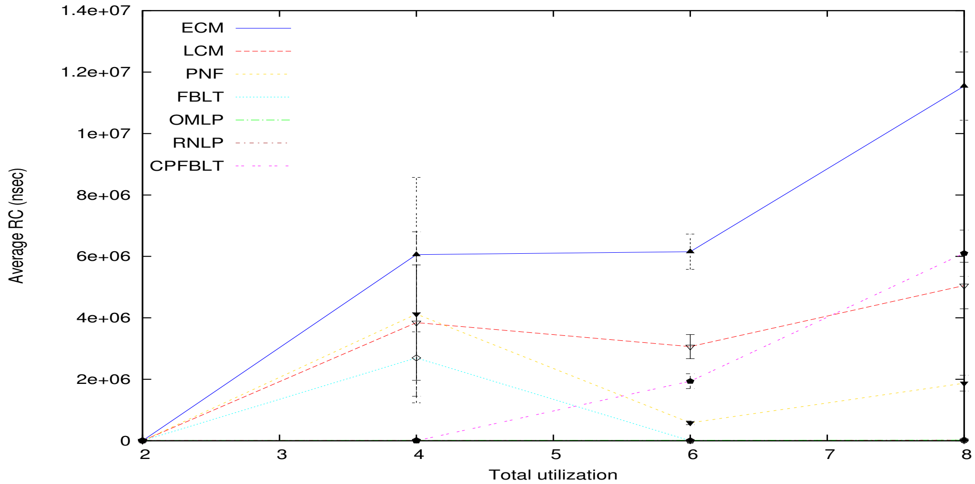


Figure C.201: Avg_RC for Tasksets 201, 471, 741 and 1011

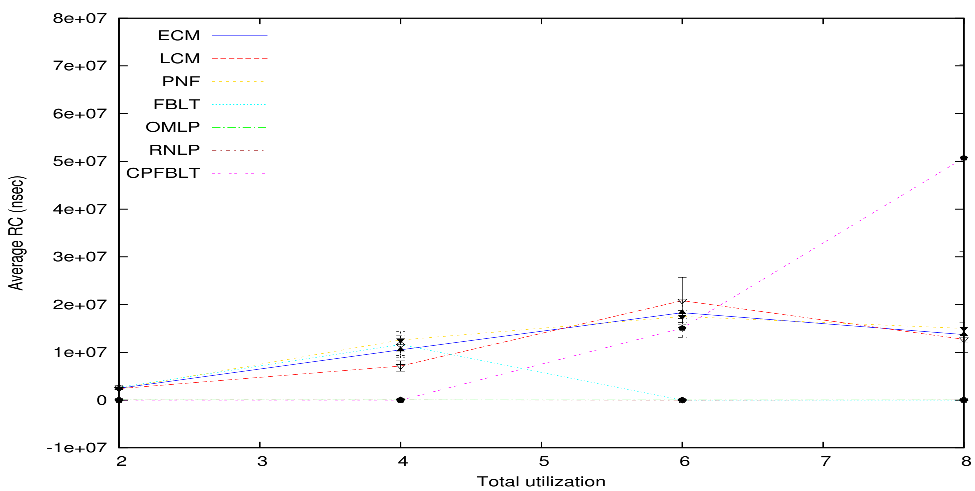


Figure C.202: Avg_RC for Tasksets 202, 472, 742 and 1012

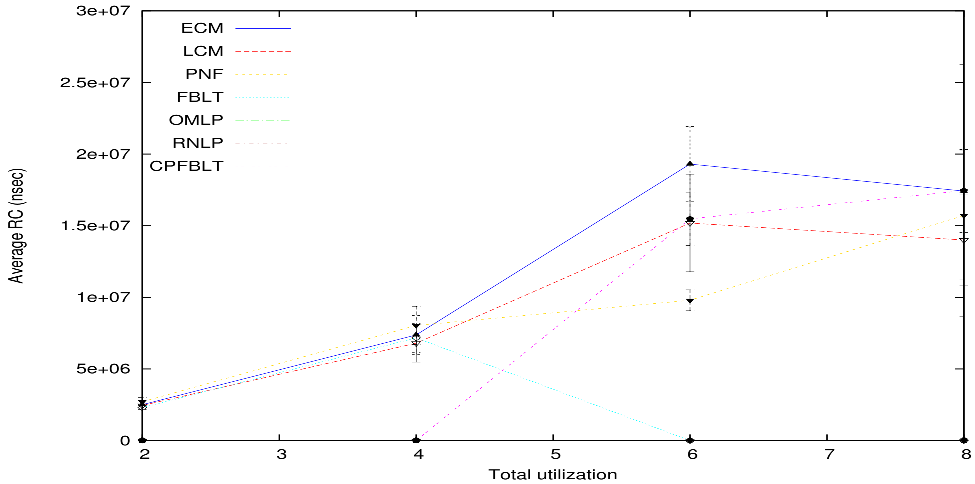


Figure C.203: Avg_RC for Tasksets 203, 473, 743 and 1013

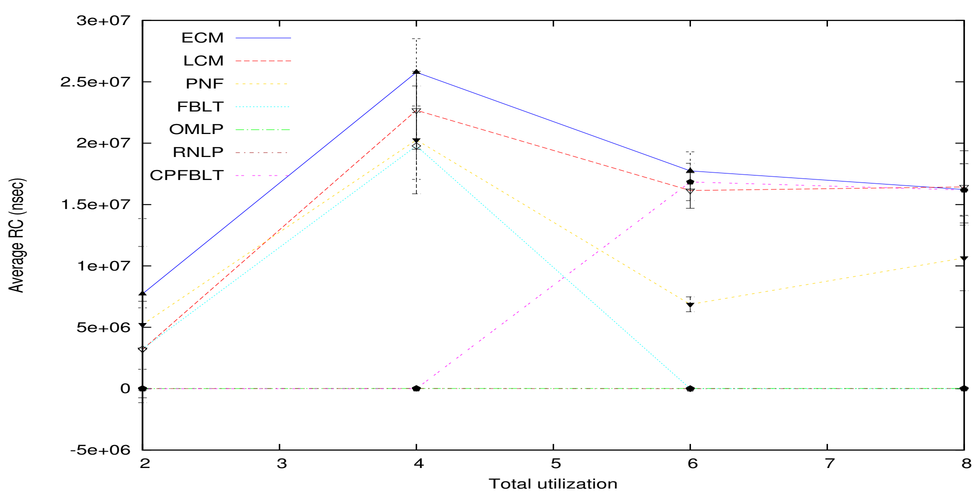


Figure C.204: Avg_RC for Tasksets 204, 474, 744 and 1014

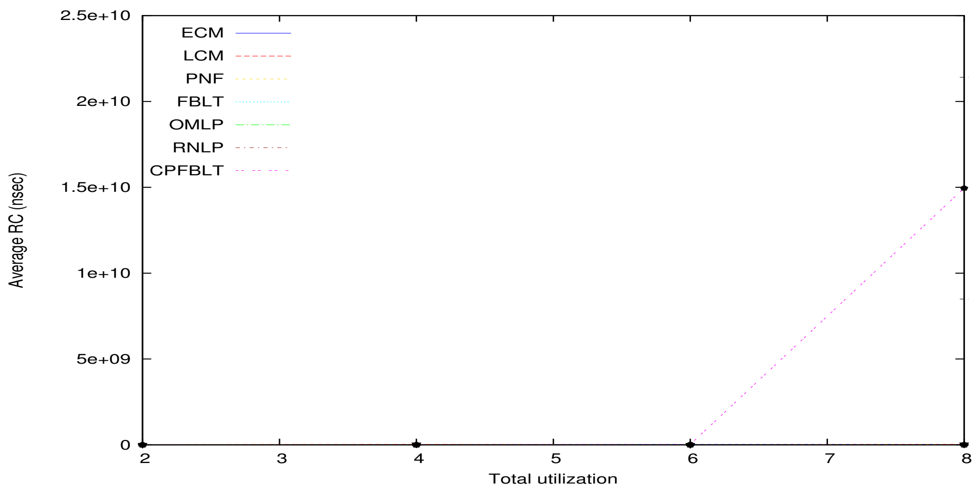


Figure C.205: Avg_RC for Tasksets 205, 475, 745 and 1015

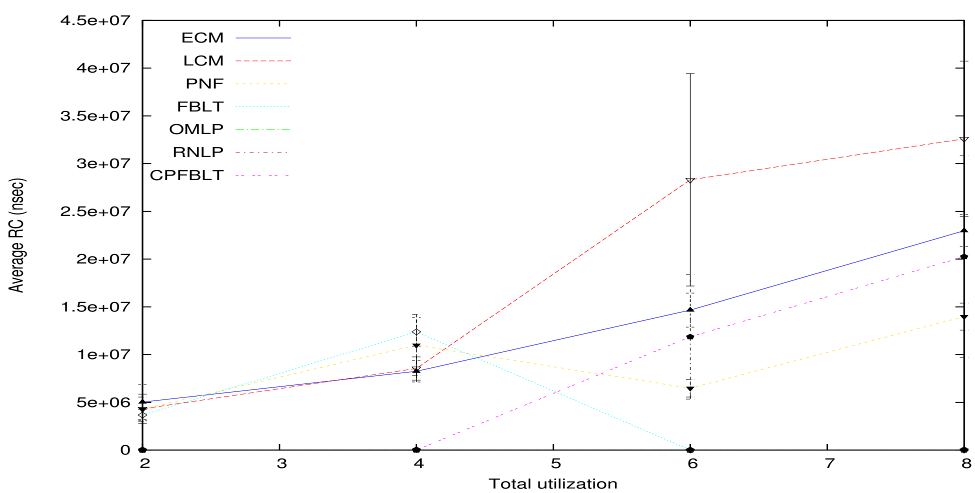


Figure C.206: Avg_RC for Tasksets 206, 476, 746 and 1016

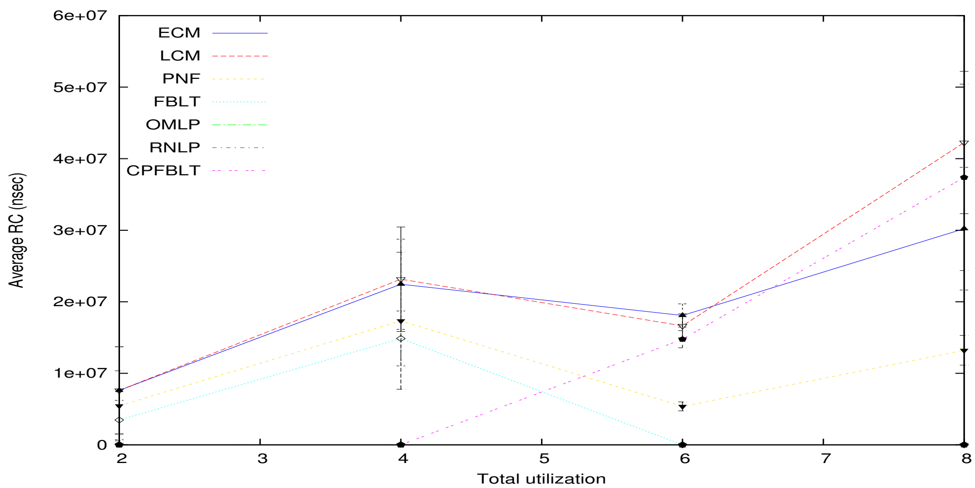


Figure C.207: Avg_RC for Tasksets 207, 477, 747 and 1017

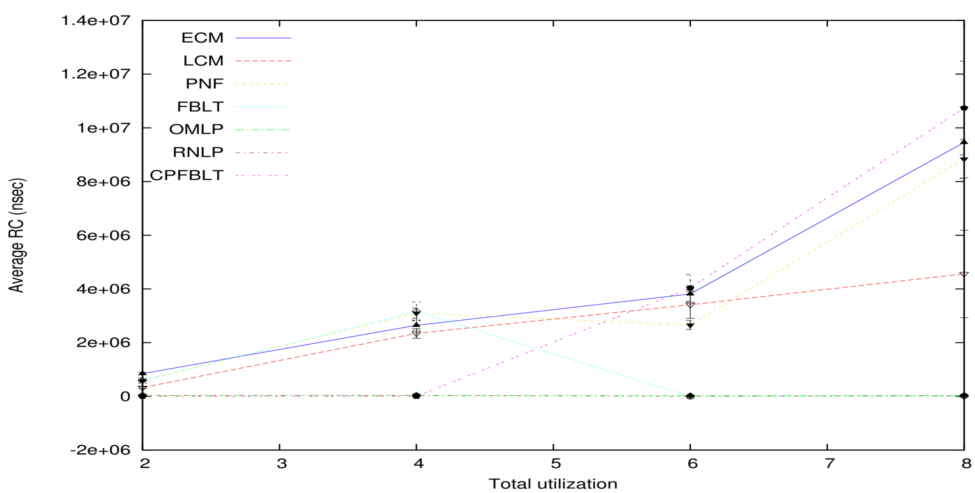


Figure C.208: Avg_RC for Tasksets 208, 478, 748 and 1018

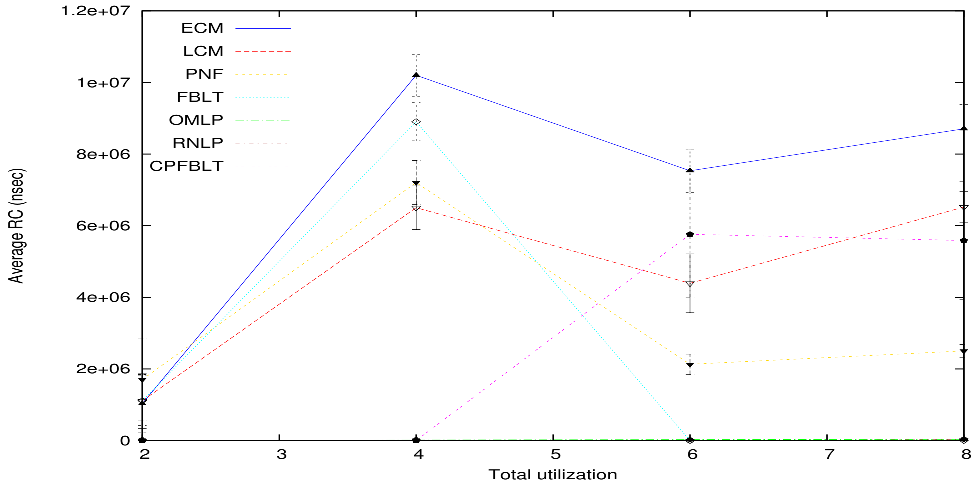


Figure C.209: Avg_RC for Tasksets 209, 479, 749 and 1019

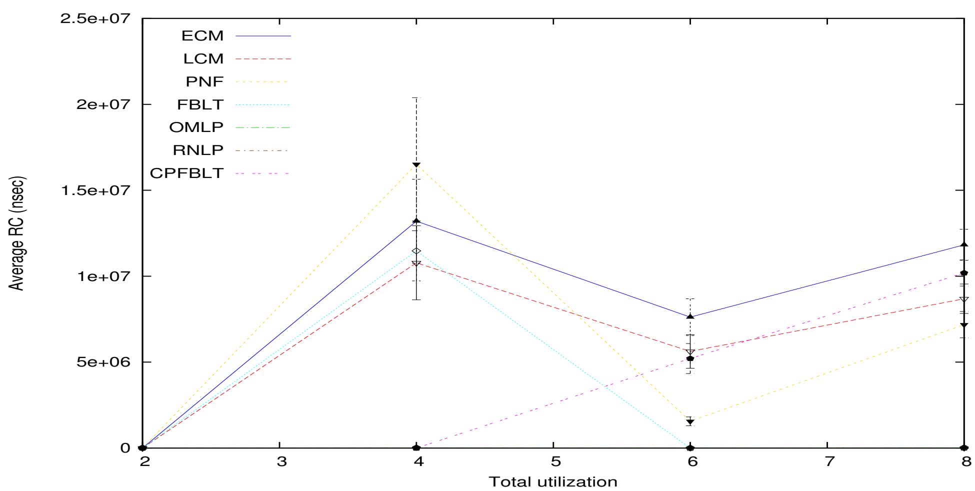


Figure C.210: Avg_RC for Tasksets 210, 480, 750 and 1020

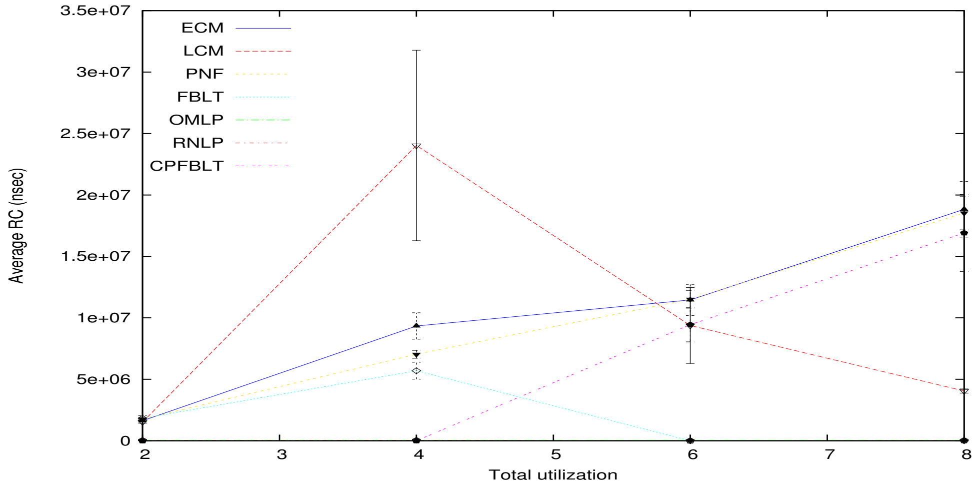


Figure C.211: Avg_RC for Tasksets 211, 481, 751 and 1021

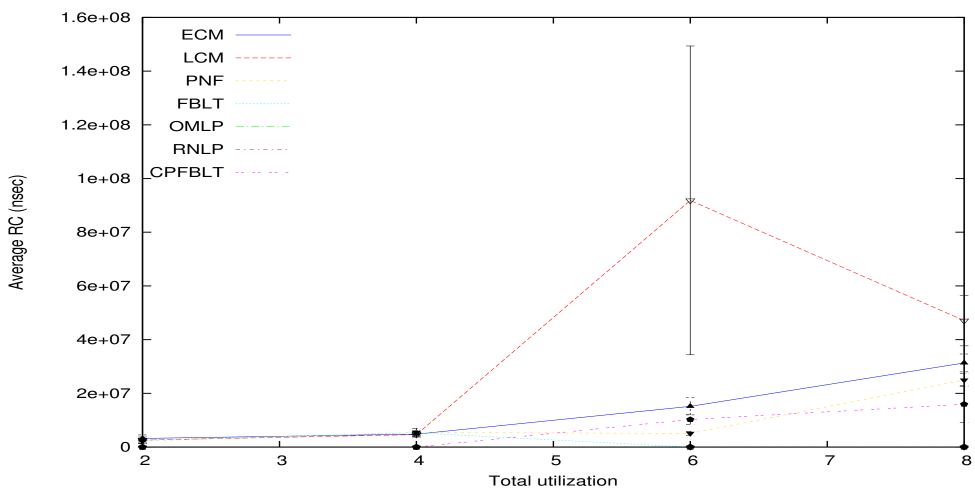


Figure C.212: Avg_RC for Tasksets 212, 482, 752 and 1022

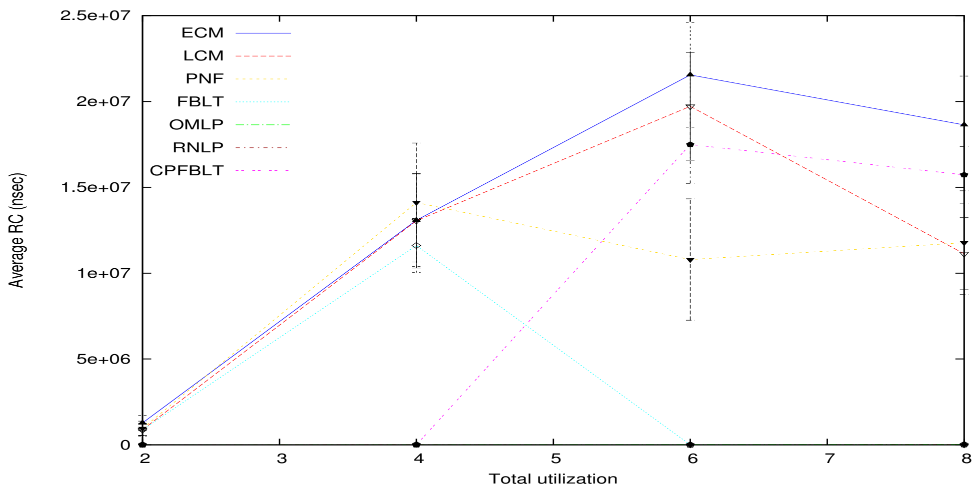


Figure C.213: Avg_RC for Tasksets 213, 483, 753 and 1023

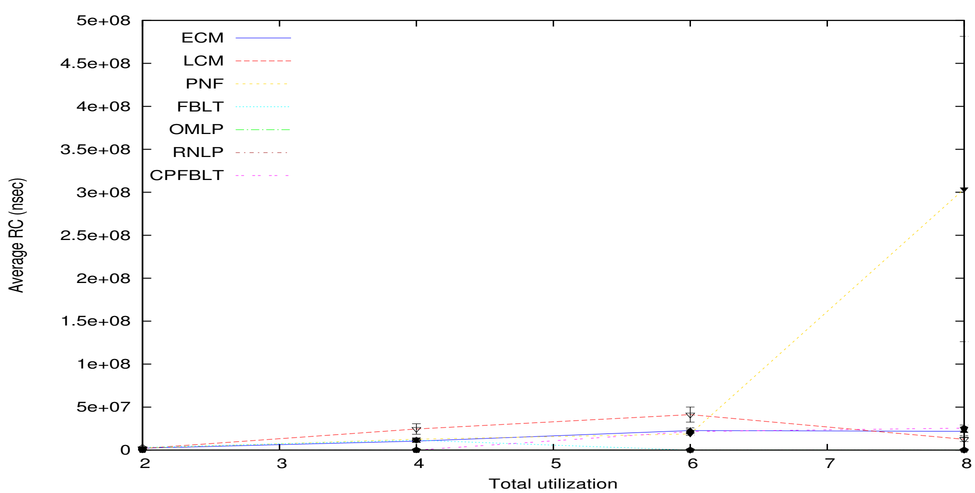


Figure C.214: Avg_RC for Tasksets 214, 484, 754 and 1024

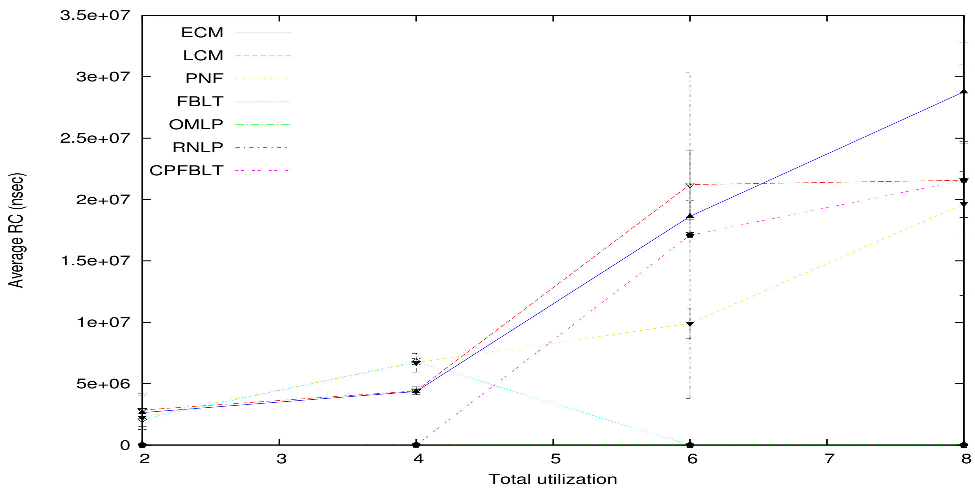


Figure C.215: Avg_RC for Tasksets 215, 485, 755 and 1025

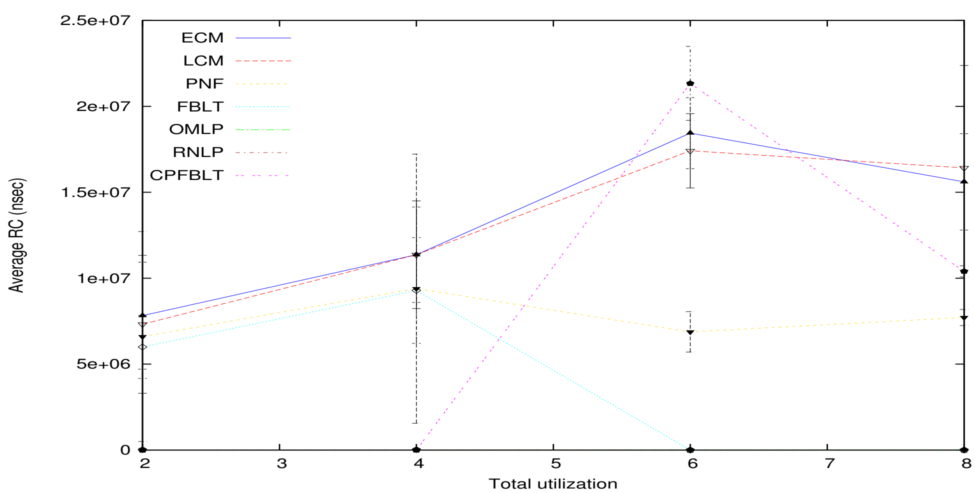


Figure C.216: Avg_RC for Tasksets 216, 486, 756 and 1026

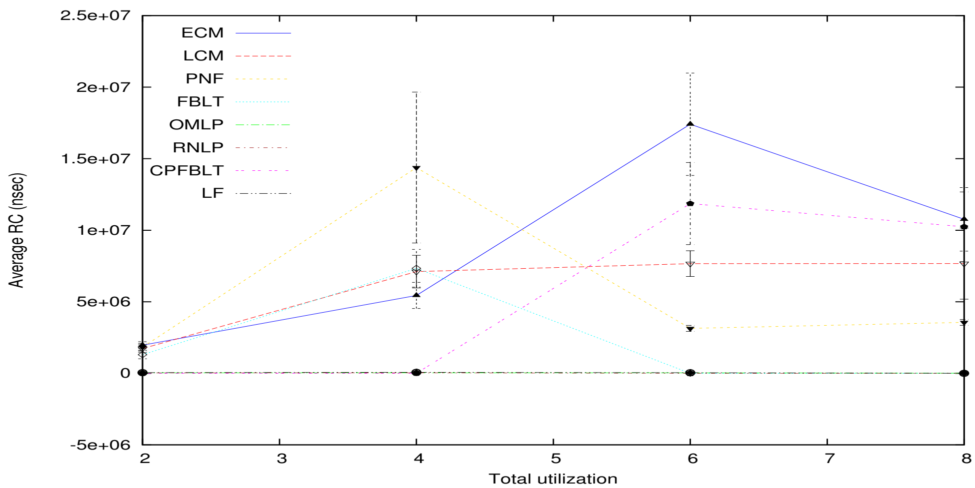


Figure C.217: Avg_RC for Tasksets 217, 487, 757 and 1027

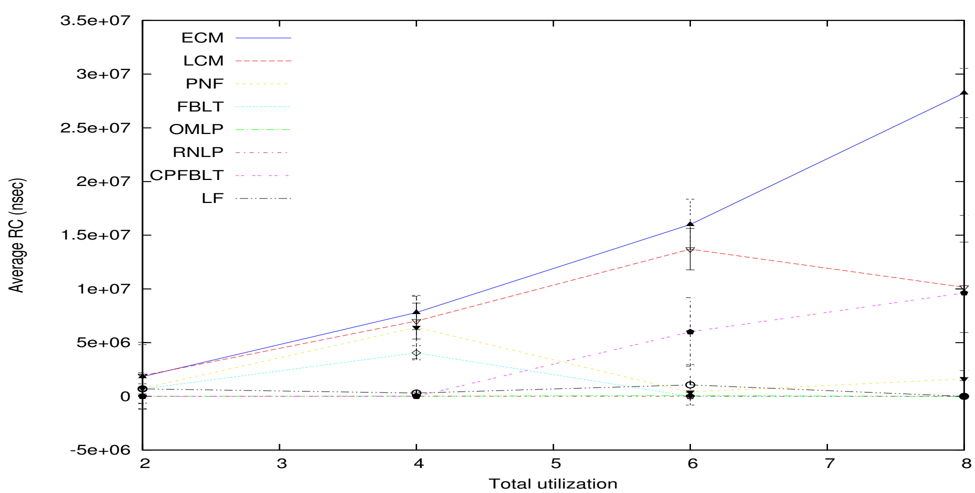


Figure C.218: Avg_RC for Tasksets 218, 488, 758 and 1028

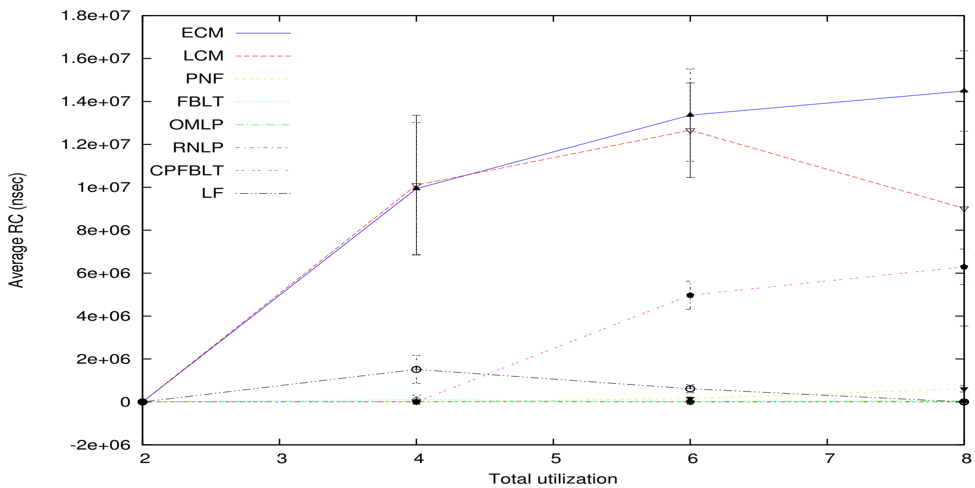


Figure C.219: Avg_RC for Tasksets 219, 489, 759 and 1029

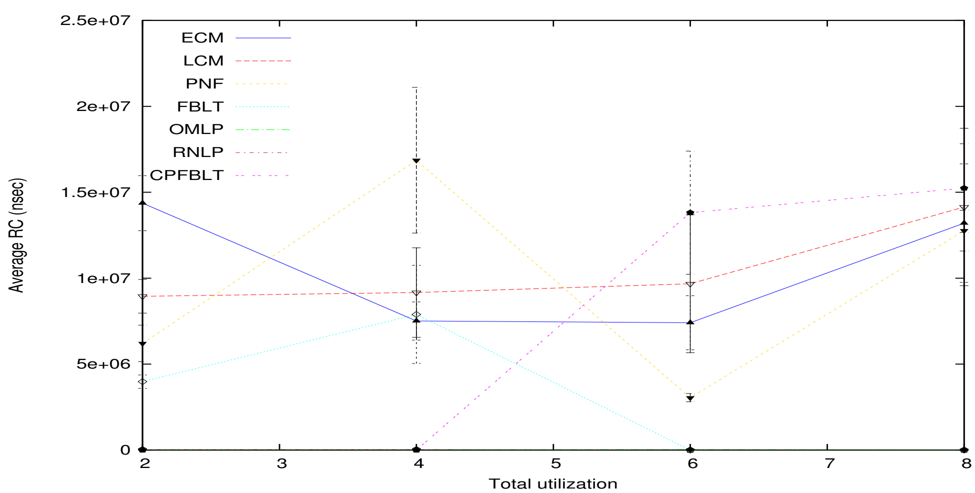


Figure C.220: Avg_RC for Tasksets 220, 490, 760 and 1030

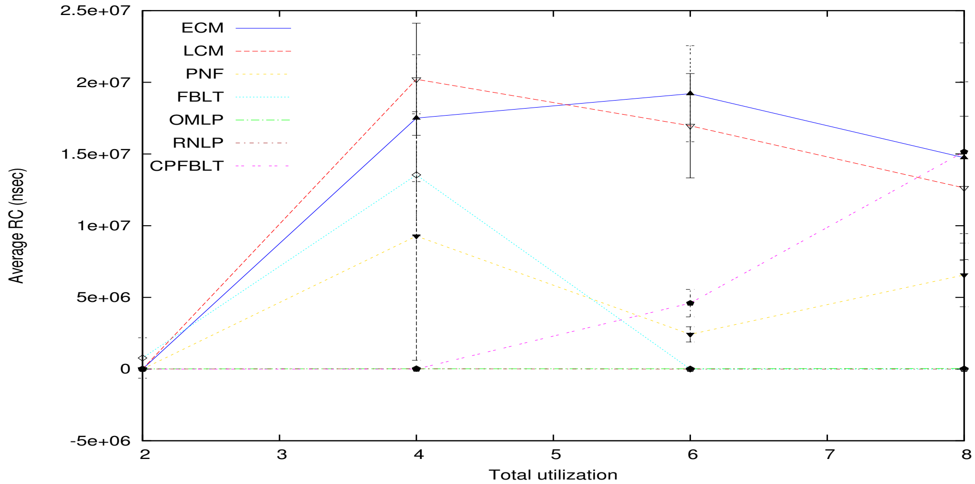


Figure C.221: Avg_RC for Tasksets 221, 491, 761 and 1031

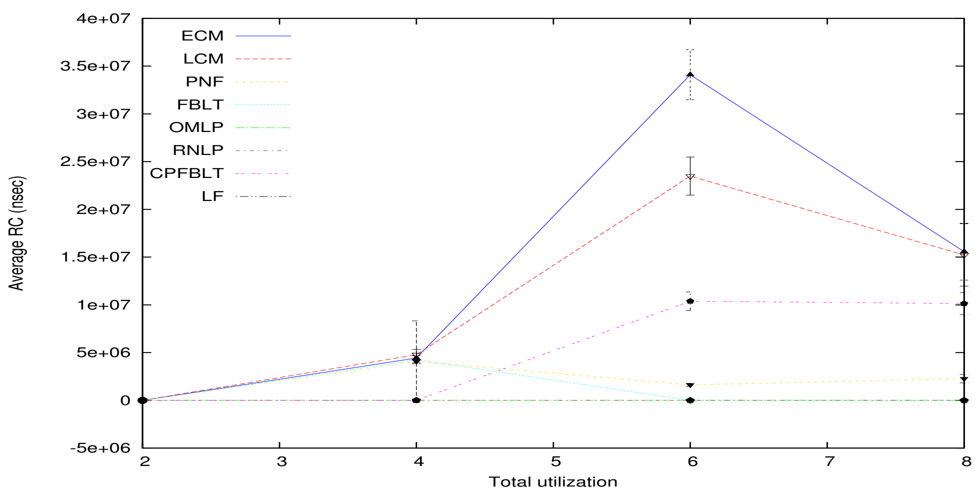


Figure C.222: Avg_RC for Tasksets 222, 492, 762 and 1032

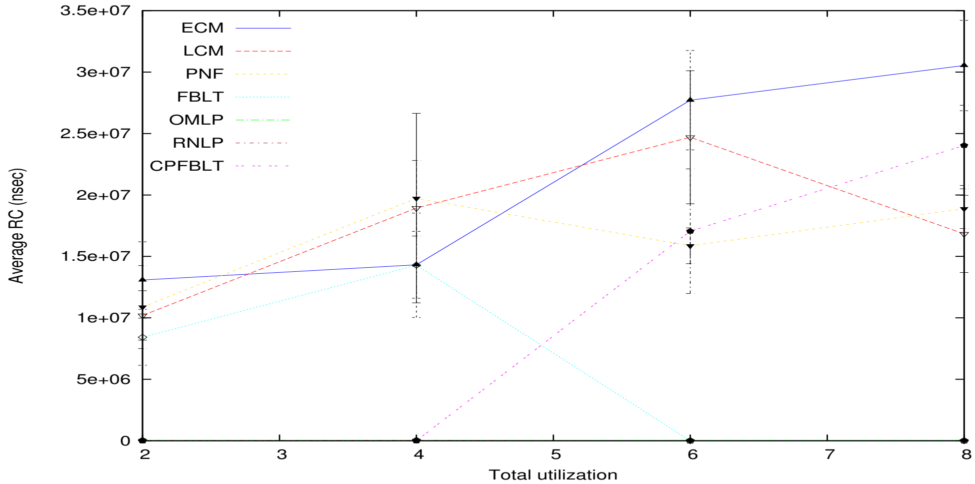


Figure C.223: Avg_RC for Tasksets 223, 493, 763 and 1033

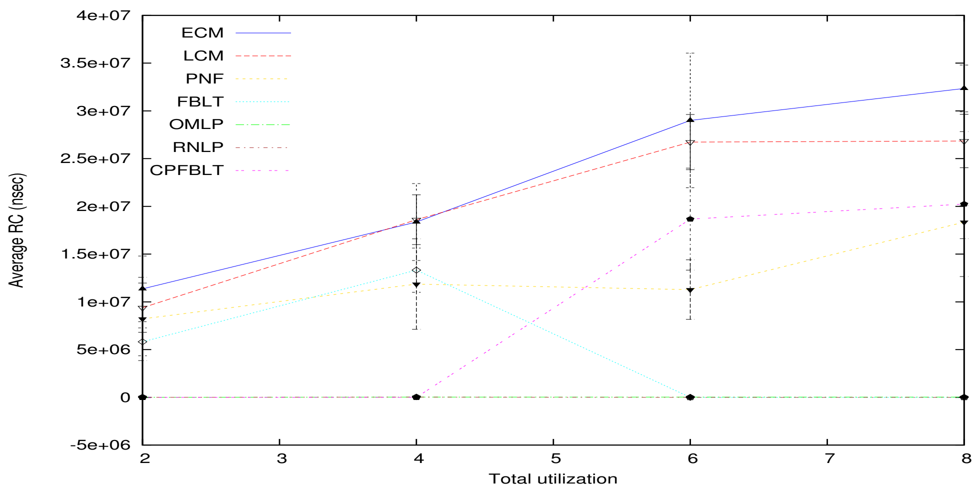


Figure C.224: Avg_RC for Tasksets 224, 494, 764 and 1034

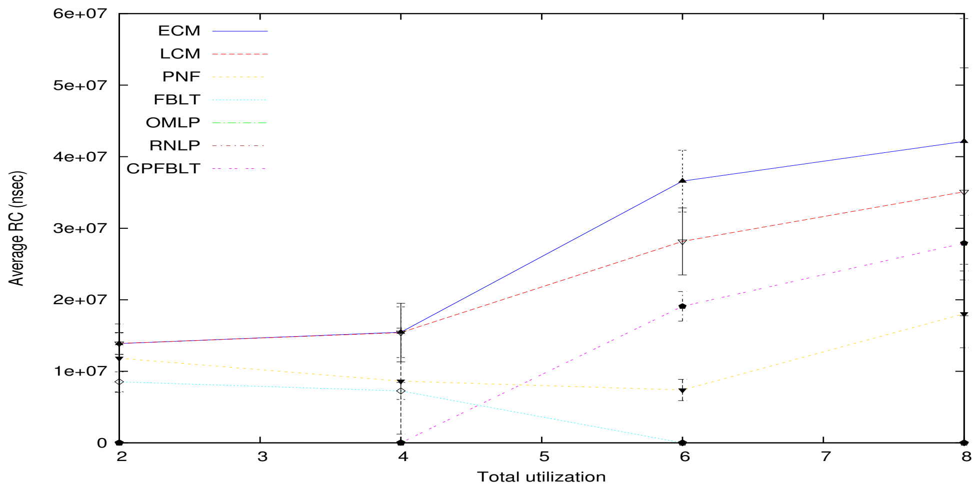


Figure C.225: Avg_RC for Tasksets 225, 495, 765 and 1035

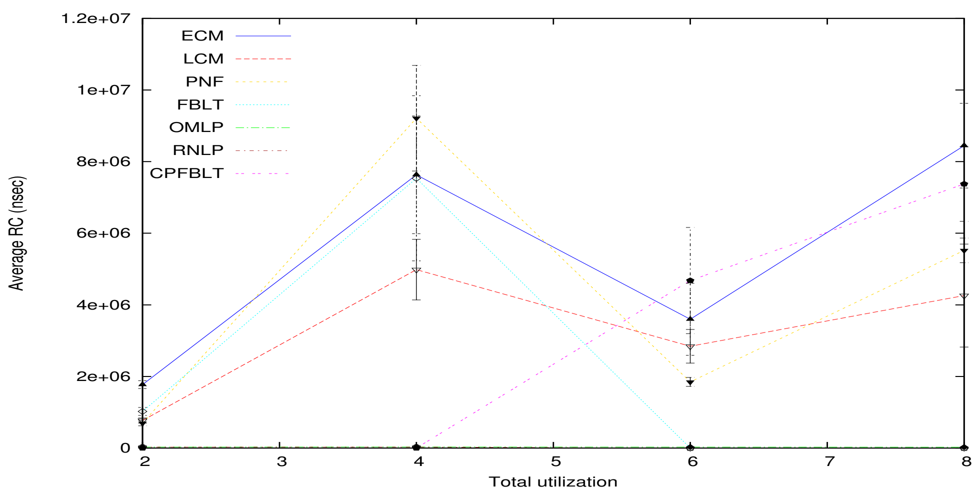


Figure C.226: Avg_RC for Tasksets 226, 496, 766 and 1036

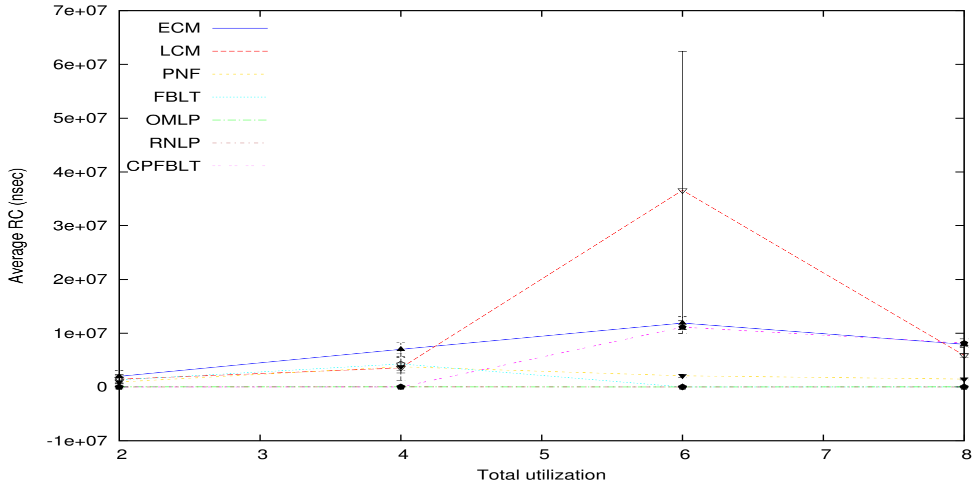


Figure C.227: Avg_RC for Tasksets 227, 497, 767 and 1037

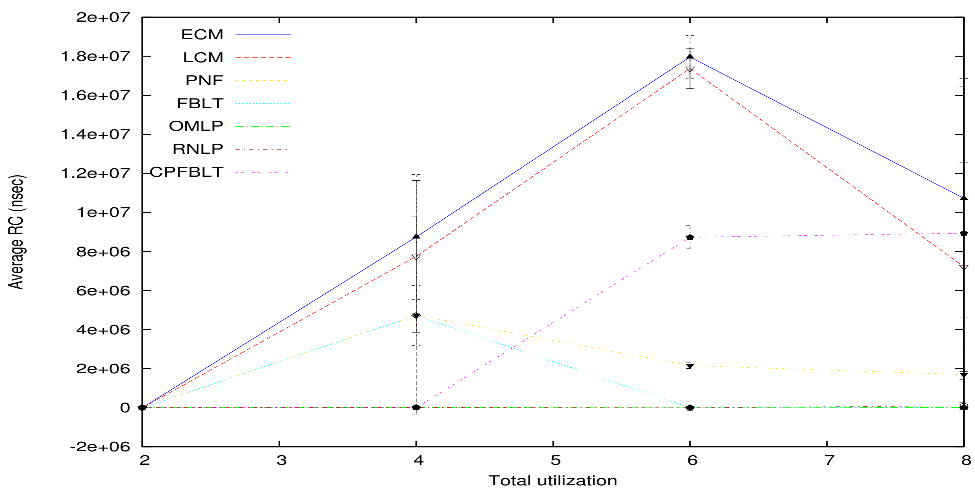


Figure C.228: Avg_RC for Tasksets 228, 498, 768 and 1038

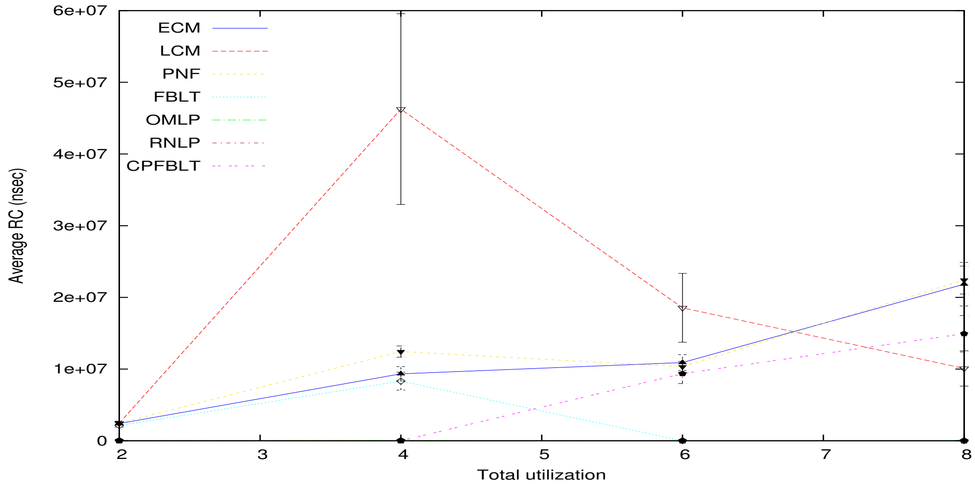


Figure C.229: Avg_RC for Tasksets 229, 499, 769 and 1039

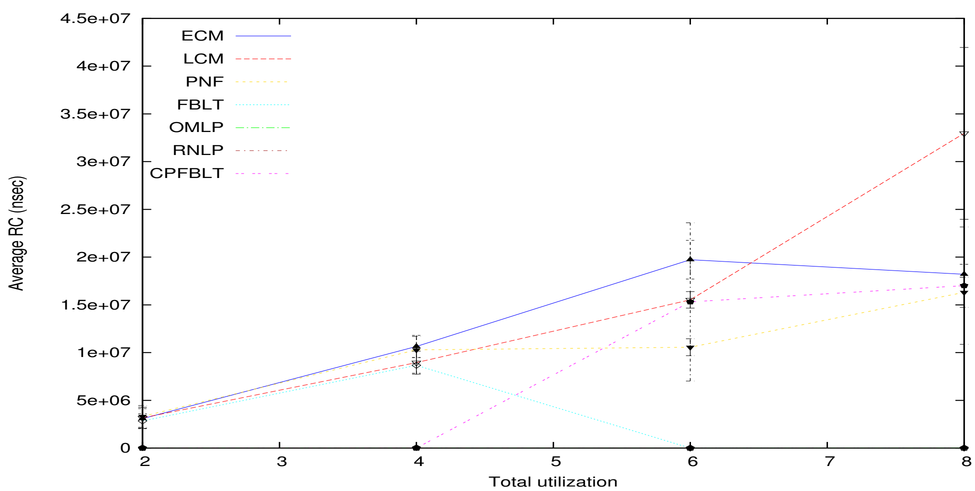


Figure C.230: Avg_RC for Tasksets 230, 500, 770 and 1040

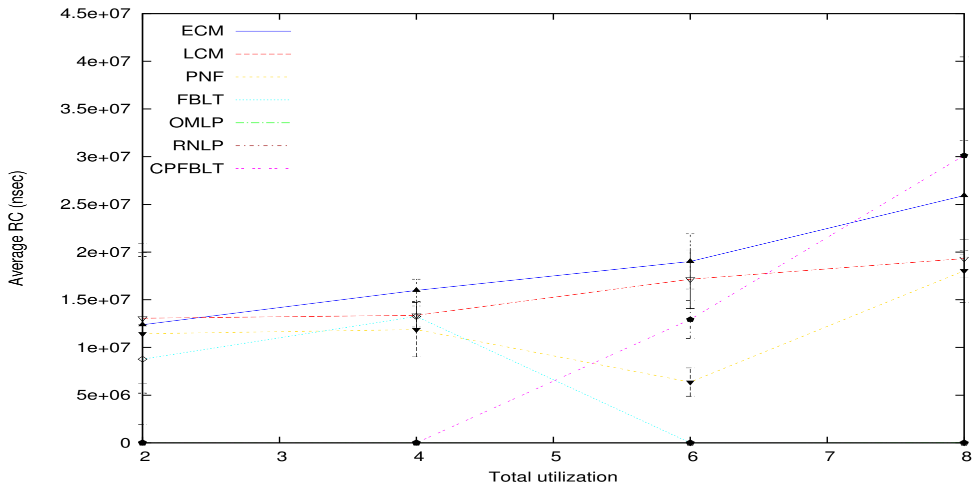


Figure C.231: Avg_RC for Tasksets 231, 501, 771 and 1041

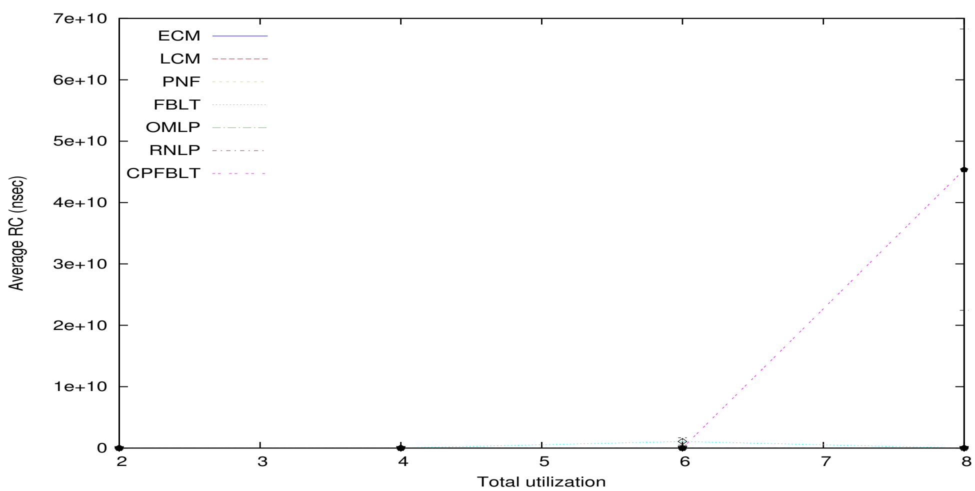


Figure C.232: Avg_RC for Tasksets 232, 502, 772 and 1042

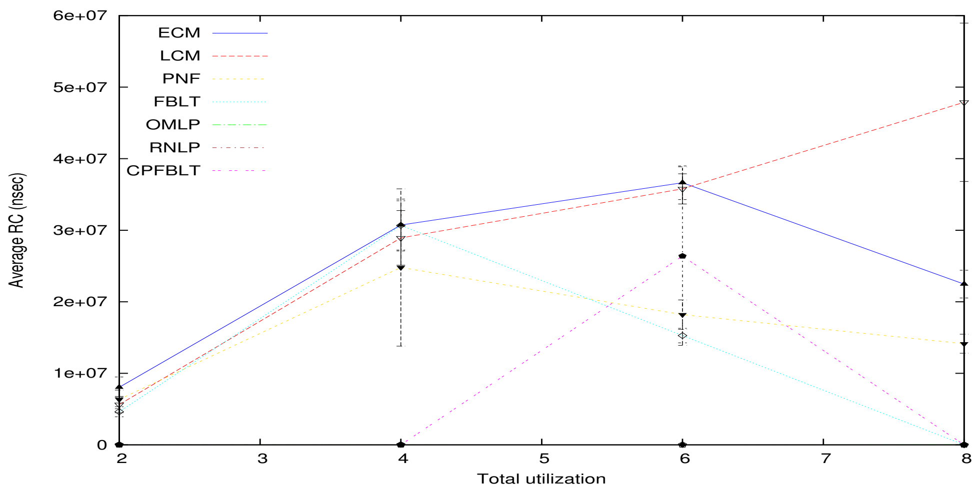


Figure C.233: Avg_RC for Tasksets 233, 503, 773 and 1043

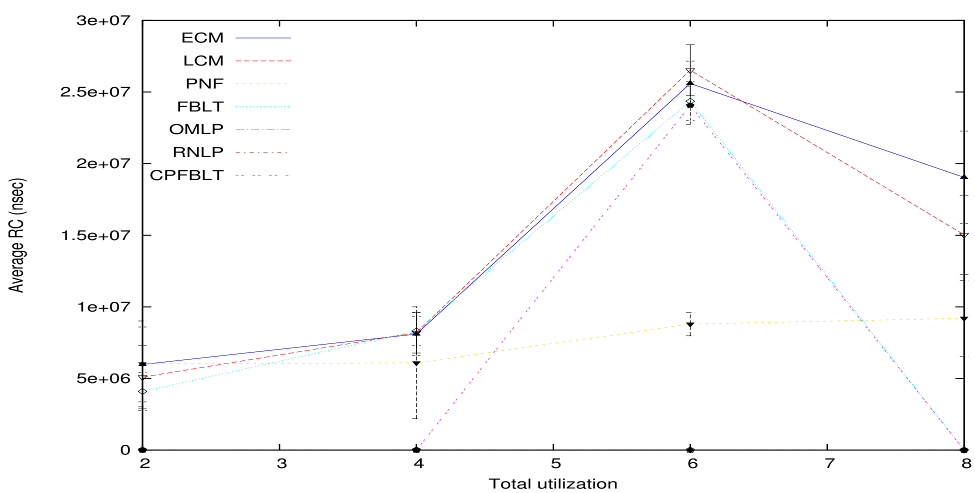


Figure C.234: Avg_RC for Tasksets 234, 504, 774 and 1044

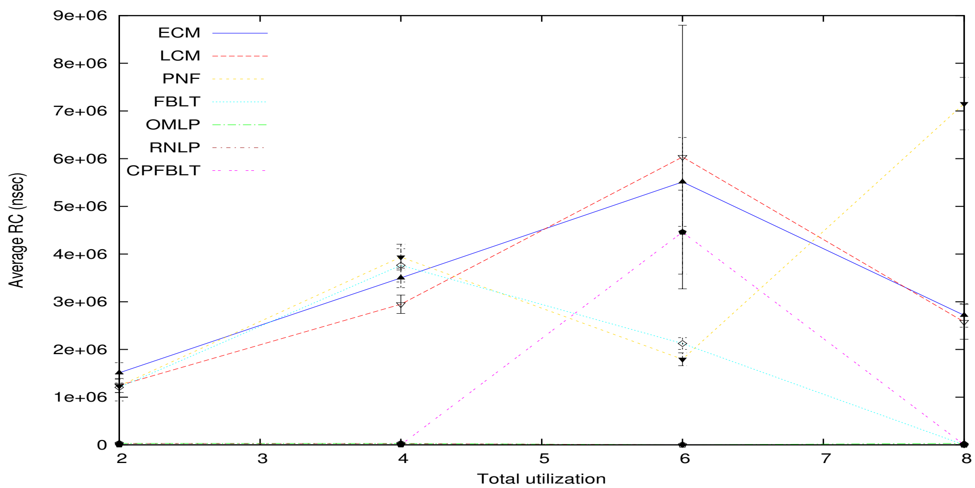


Figure C.235: Avg_RC for Tasksets 235, 505, 775 and 1045

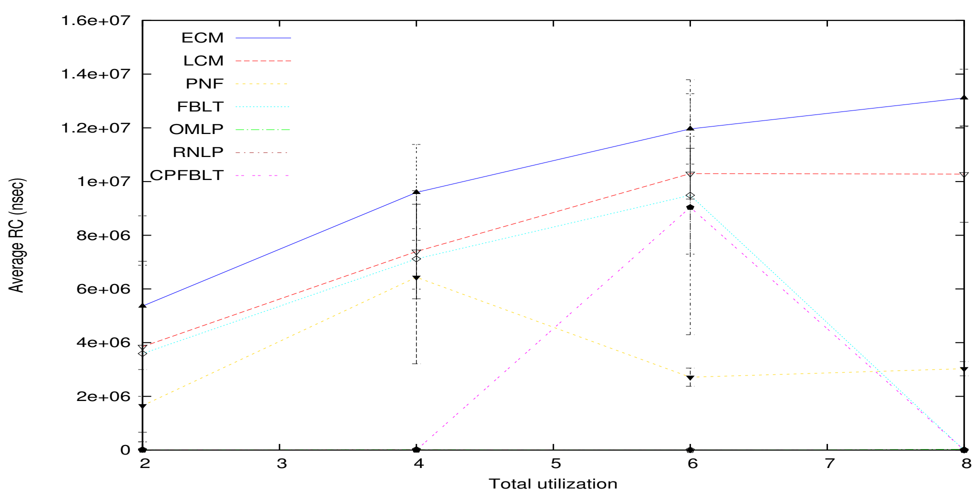


Figure C.236: Avg_RC for Tasksets 236, 506, 776 and 1046

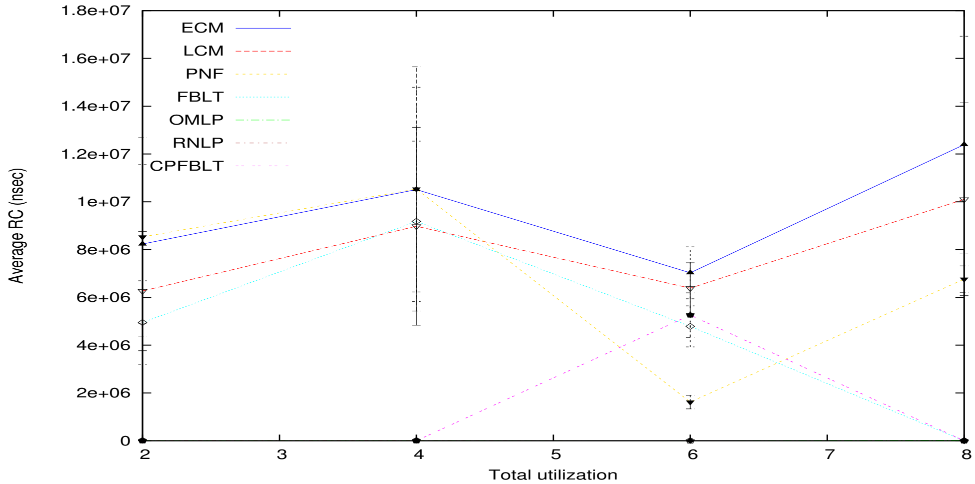


Figure C.237: Avg_RC for Tasksets 237, 507, 777 and 1047

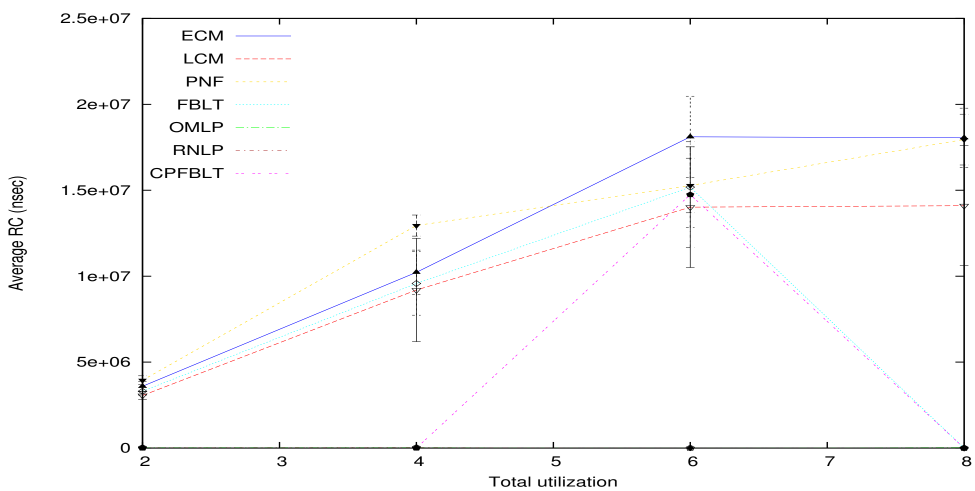


Figure C.238: Avg_RC for Tasksets 238, 508, 778 and 1048

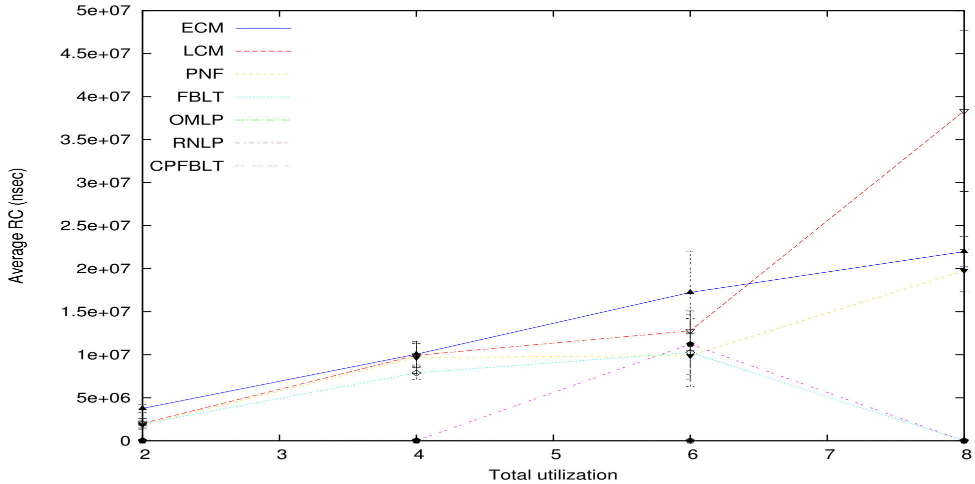


Figure C.239: Avg_RC for Tasksets 239, 509, 779 and 1049

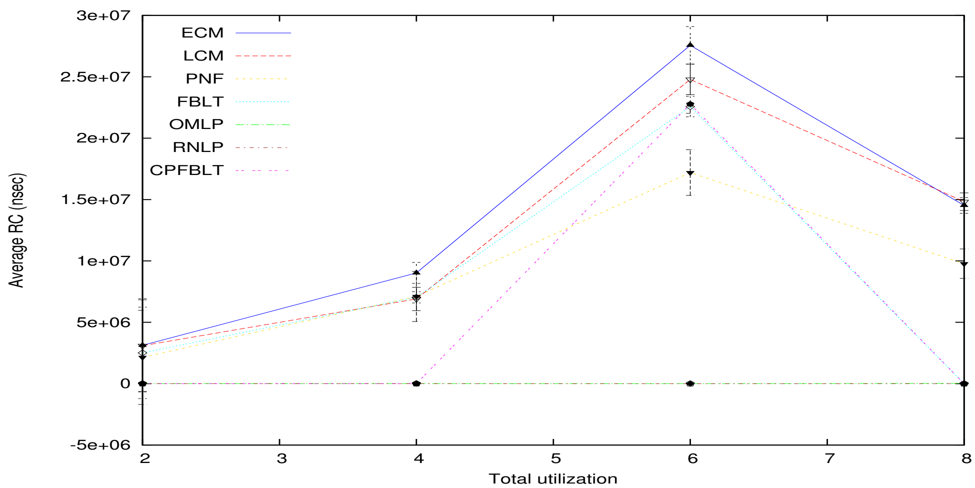


Figure C.240: Avg_RC for Tasksets 240, 510, 780 and 1050

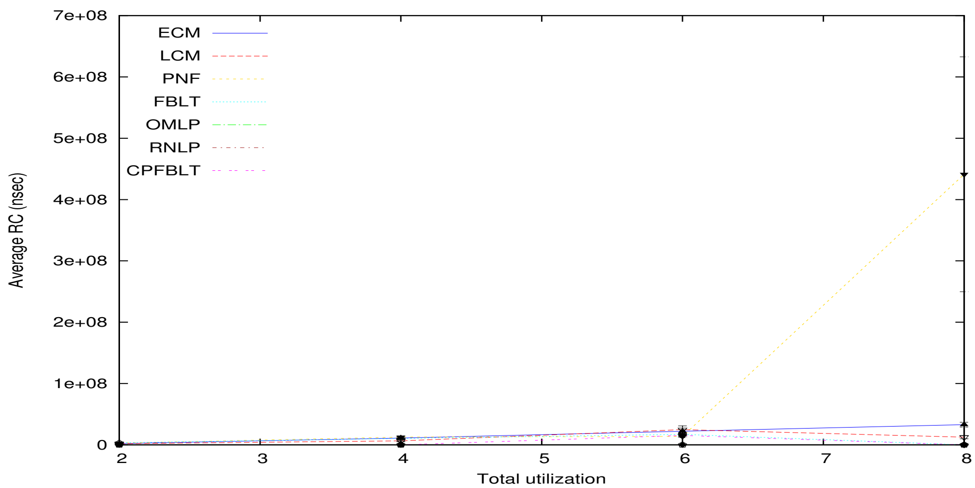


Figure C.241: Avg_RC for Tasksets 241, 511, 781 and 1051

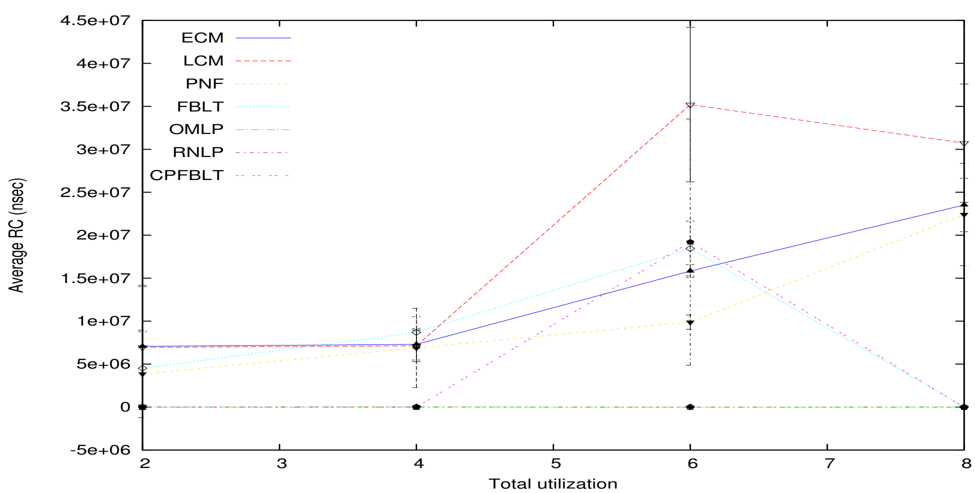


Figure C.242: Avg_RC for Tasksets 242, 512, 782 and 1052

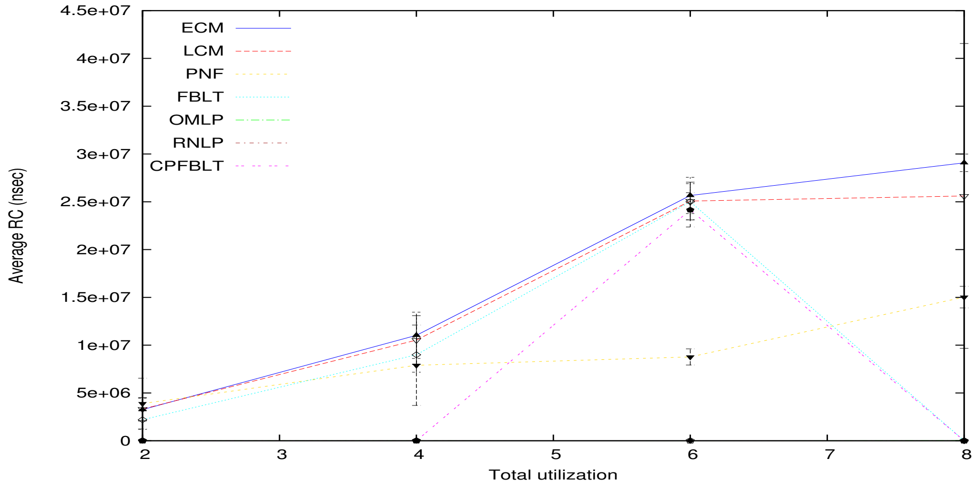


Figure C.243: Avg_RC for Tasksets 243, 513, 783 and 1053

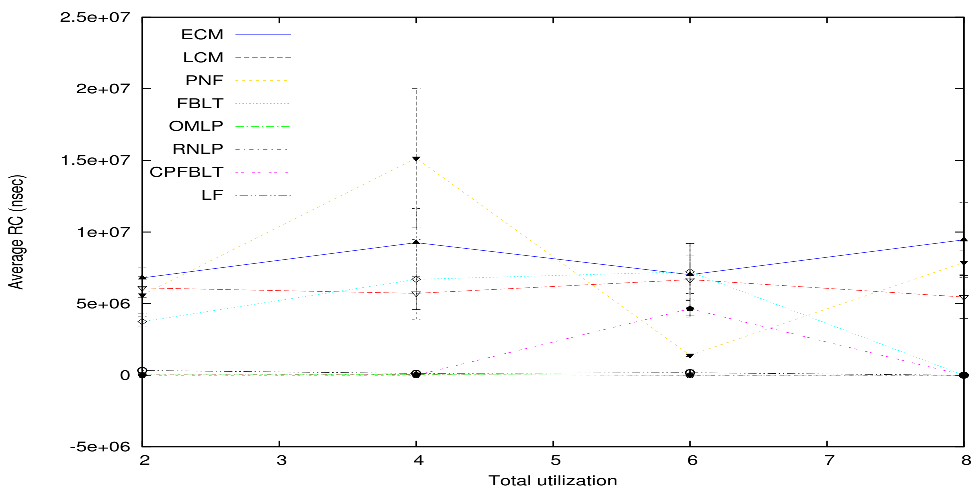


Figure C.244: Avg_RC for Tasksets 244, 514, 784 and 1054

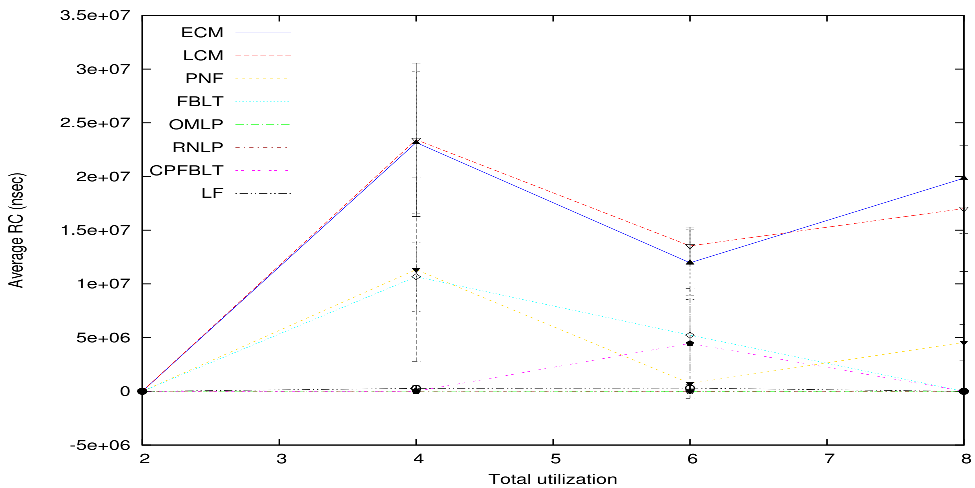


Figure C.245: Avg_RC for Tasksets 245, 515, 785 and 1055

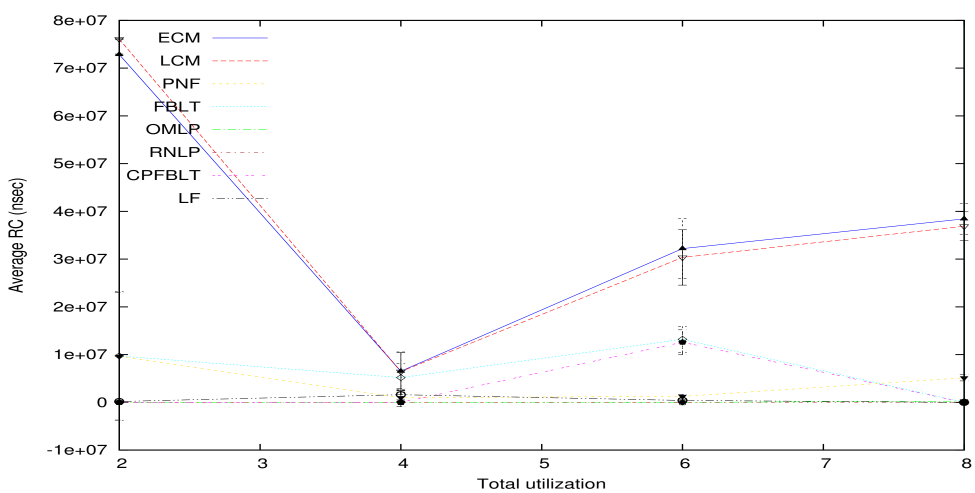


Figure C.246: Avg_RC for Tasksets 246, 516, 786 and 1056

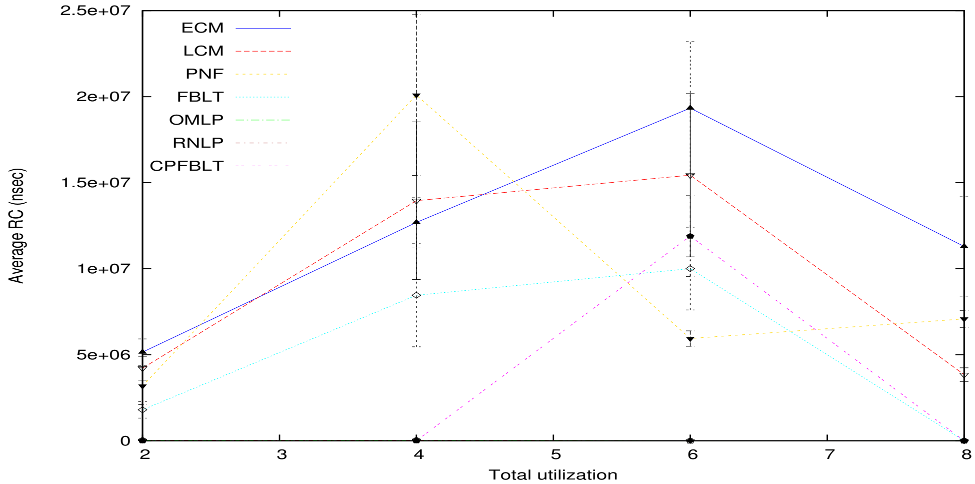


Figure C.247: Avg_RC for Tasksets 247, 517, 787 and 1057

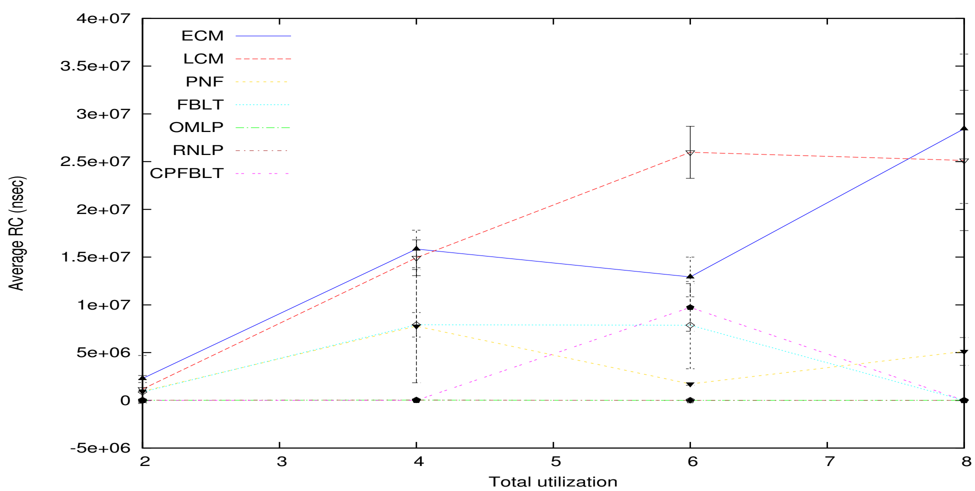


Figure C.248: Avg_RC for Tasksets 248, 518, 788 and 1058

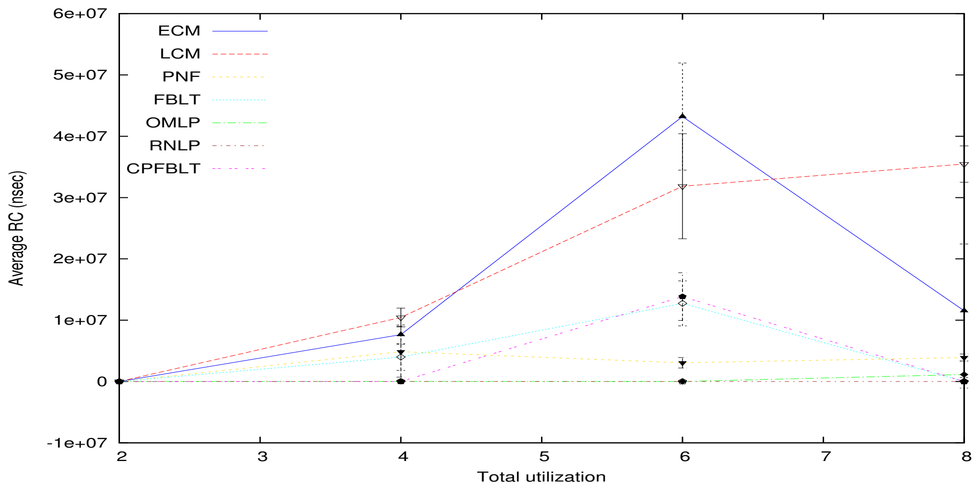


Figure C.249: Avg_RC for Tasksets 249, 519, 789 and 1059

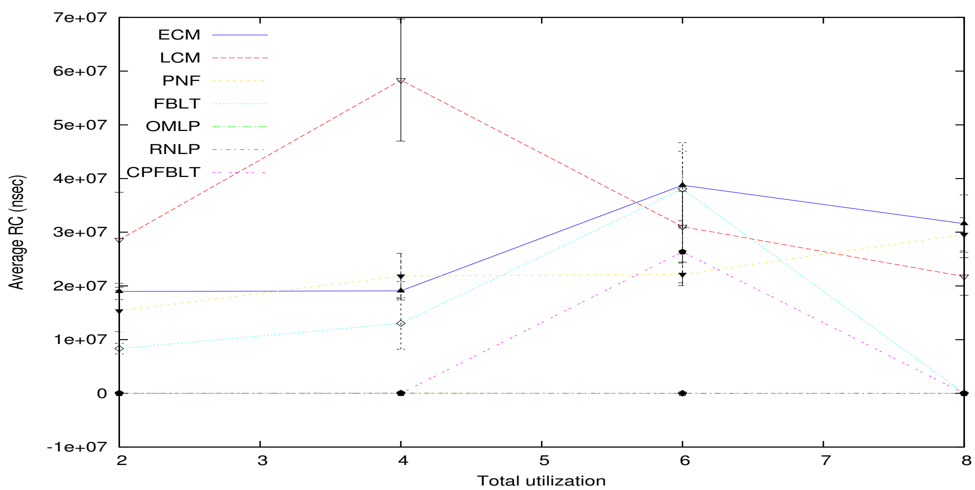


Figure C.250: Avg_RC for Tasksets 250, 520, 790 and 1060

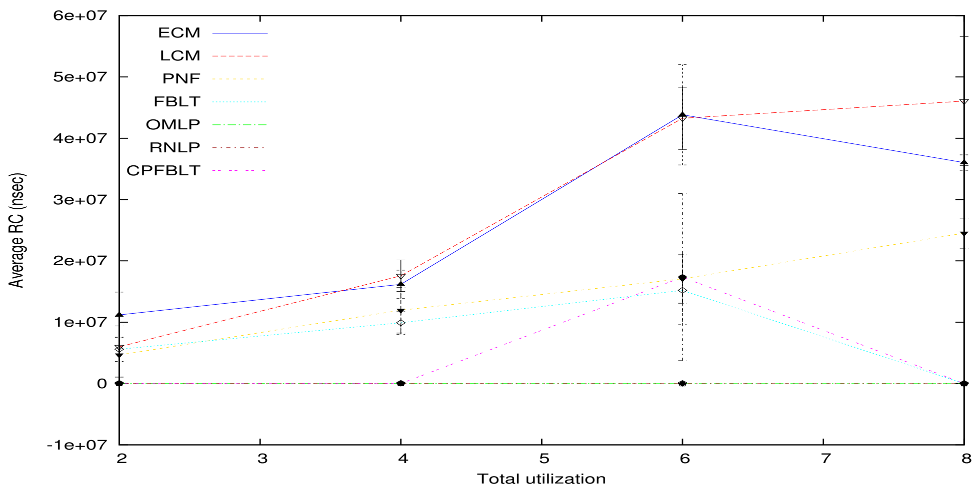


Figure C.251: Avg_RC for Tasksets 251, 521, 791 and 1061

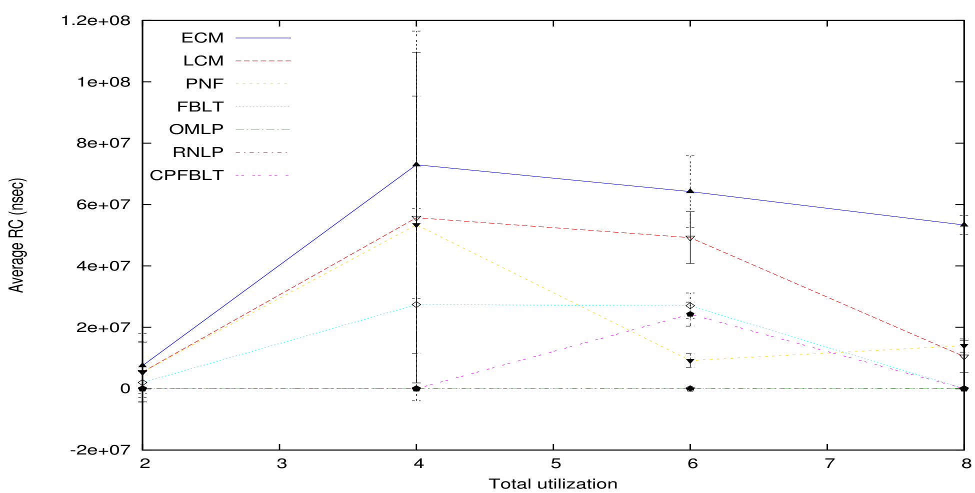


Figure C.252: Avg_RC for Tasksets 252, 522, 792 and 1062

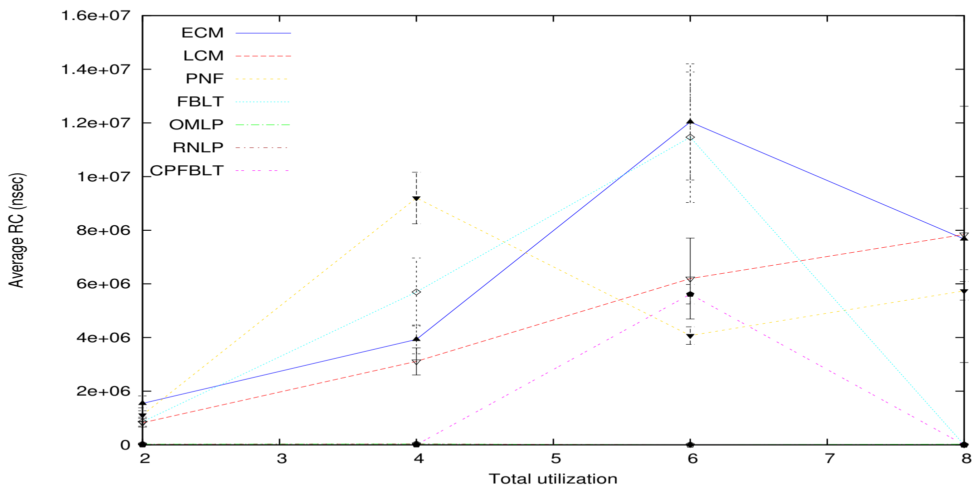


Figure C.253: Avg_RC for Tasksets 253, 523, 793 and 1063

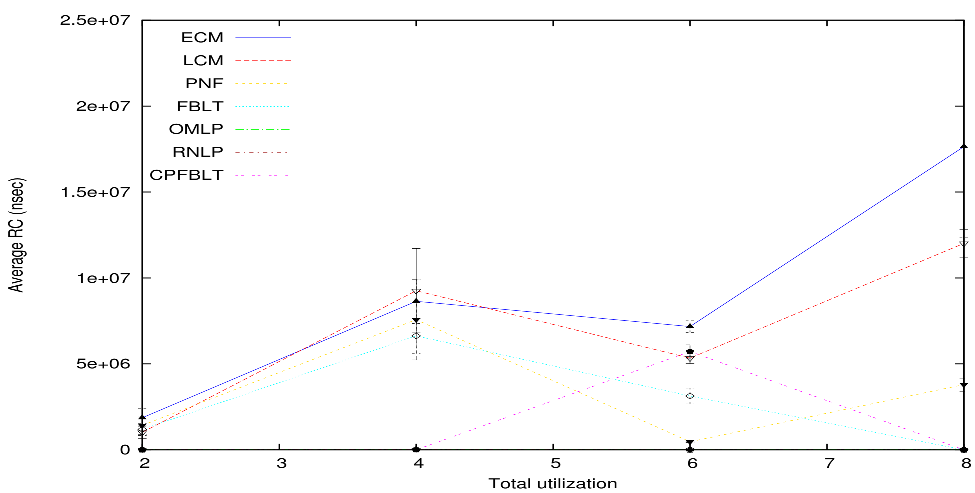


Figure C.254: Avg_RC for Tasksets 254, 524, 794 and 1064

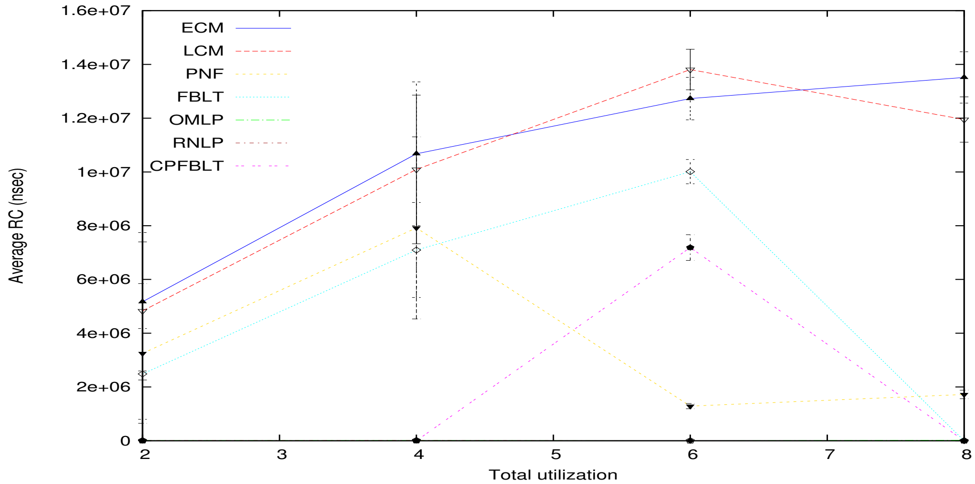


Figure C.255: Avg_RC for Tasksets 255, 525, 795 and 1065

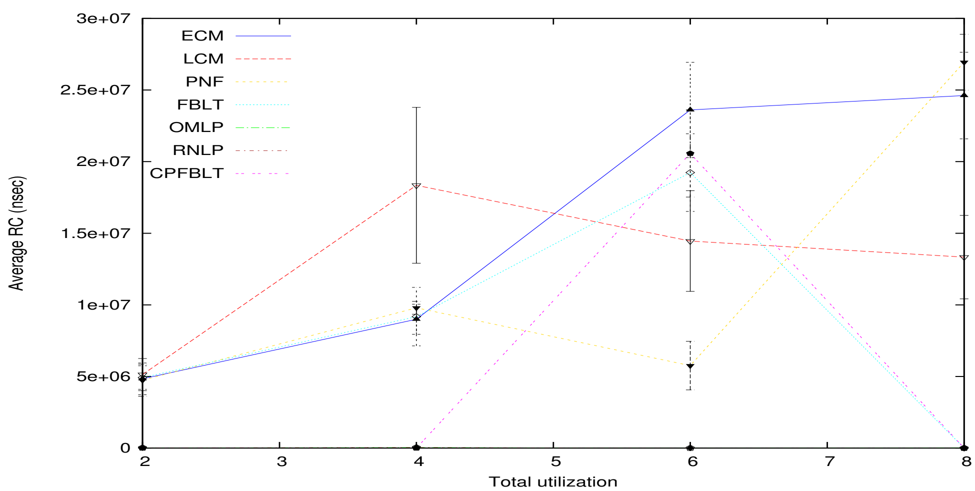


Figure C.256: Avg_RC for Tasksets 256, 526, 796 and 1066

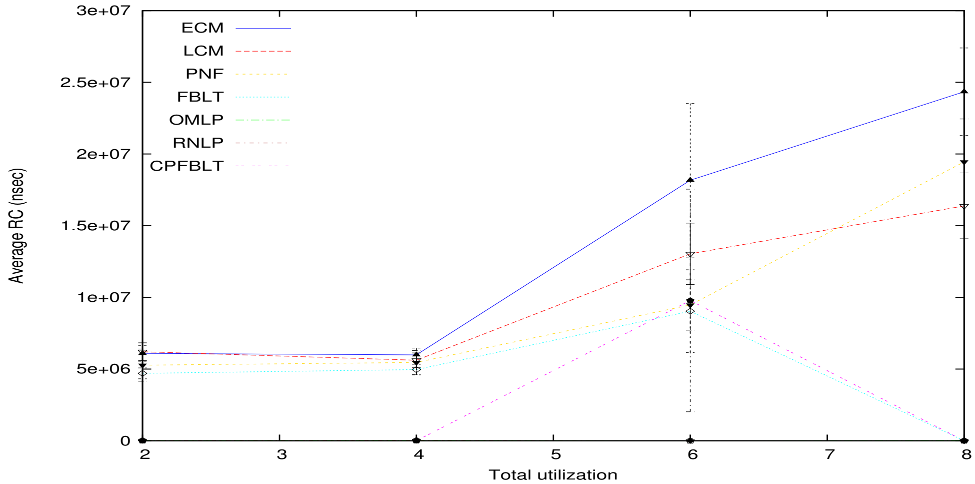


Figure C.257: Avg_RC for Tasksets 257, 527, 797 and 1067

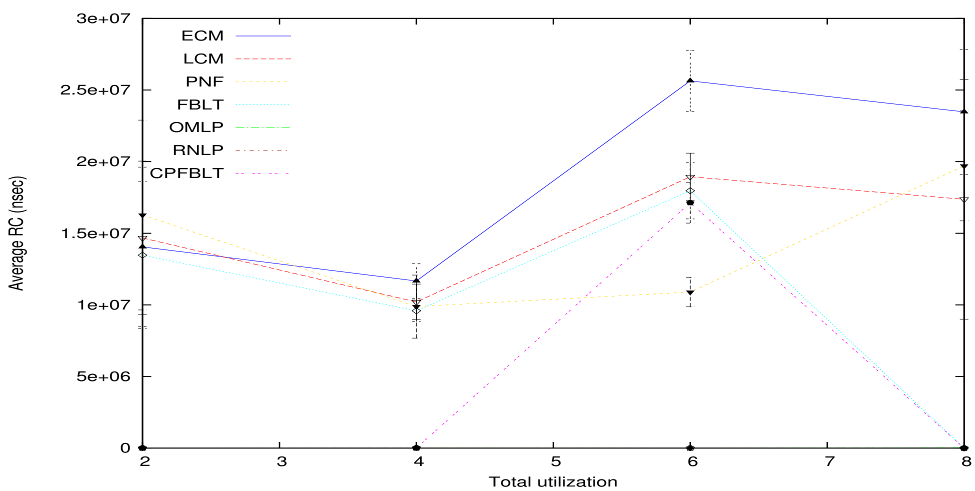


Figure C.258: Avg_RC for Tasksets 258, 528, 798 and 1068

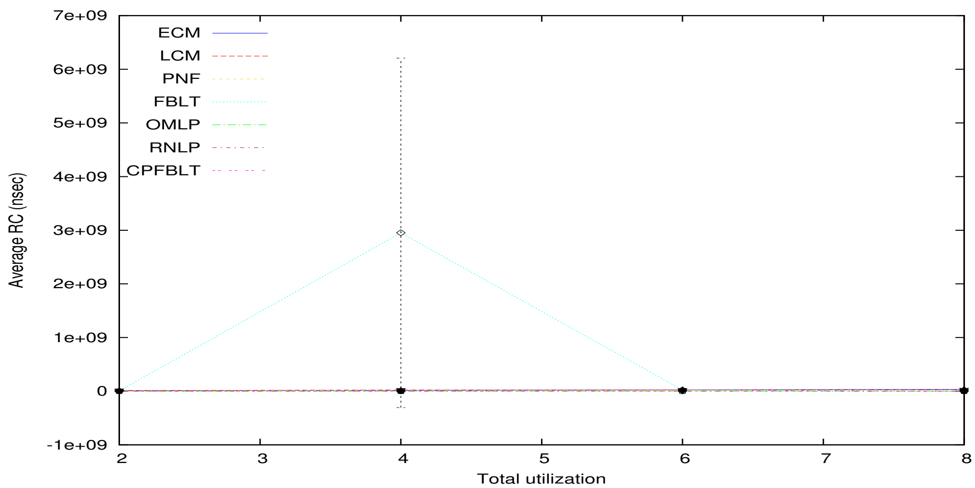


Figure C.259: Avg_RC for Tasksets 259, 529, 799 and 1069

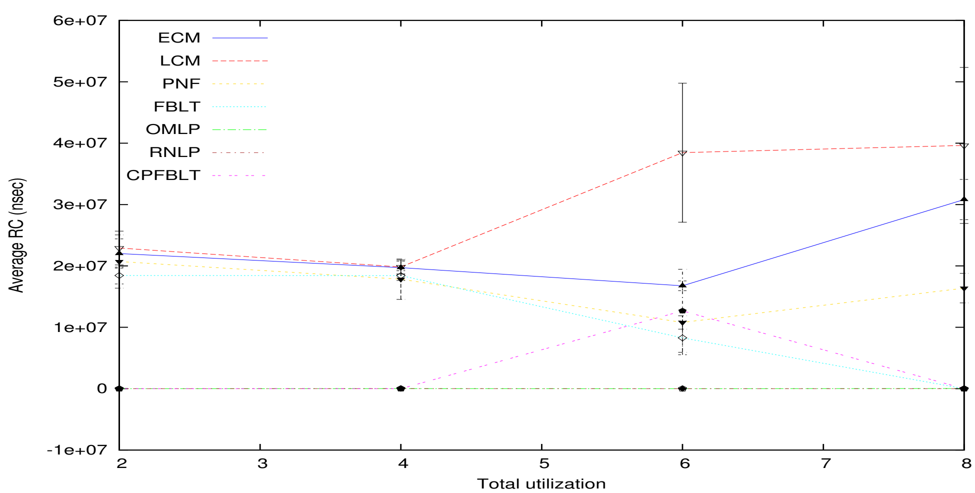


Figure C.260: Avg_RC for Tasksets 260, 530, 800 and 1070

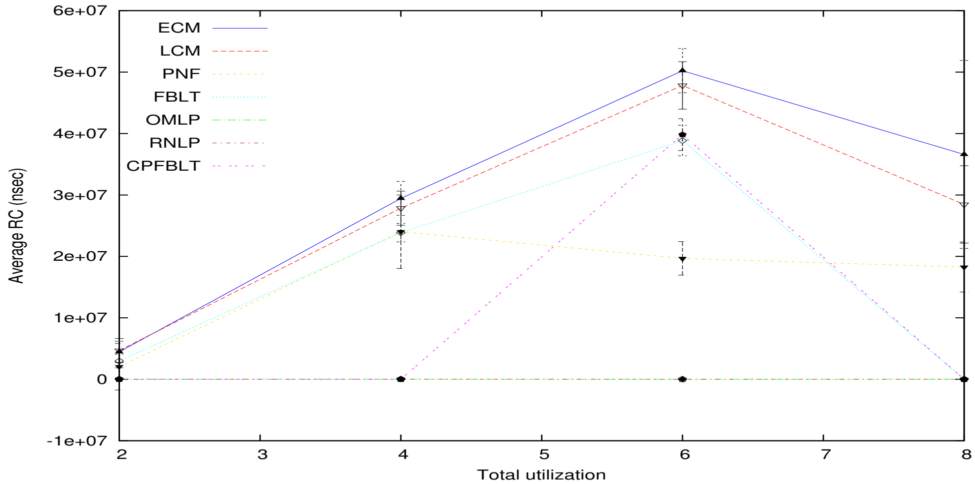


Figure C.261: Avg_RC for Tasksets 261, 531, 801 and 1071

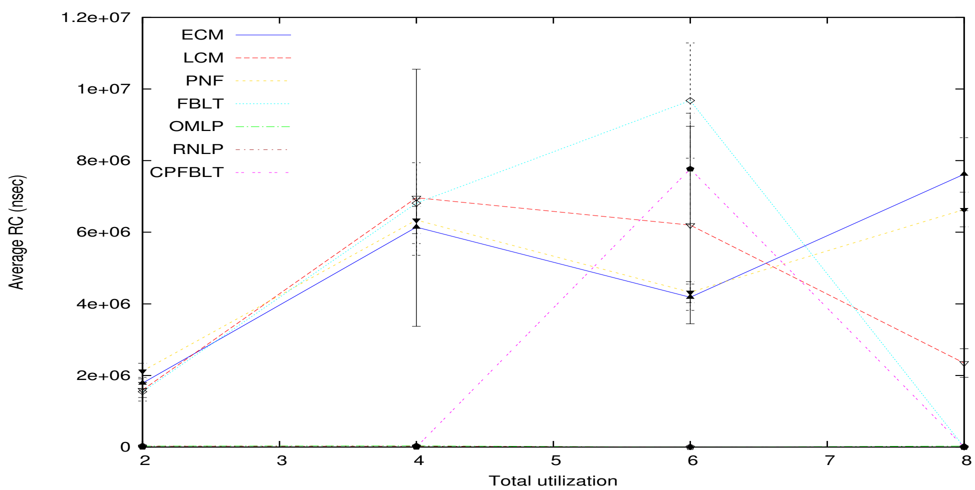


Figure C.262: Avg_RC for Tasksets 262, 532, 802 and 1072

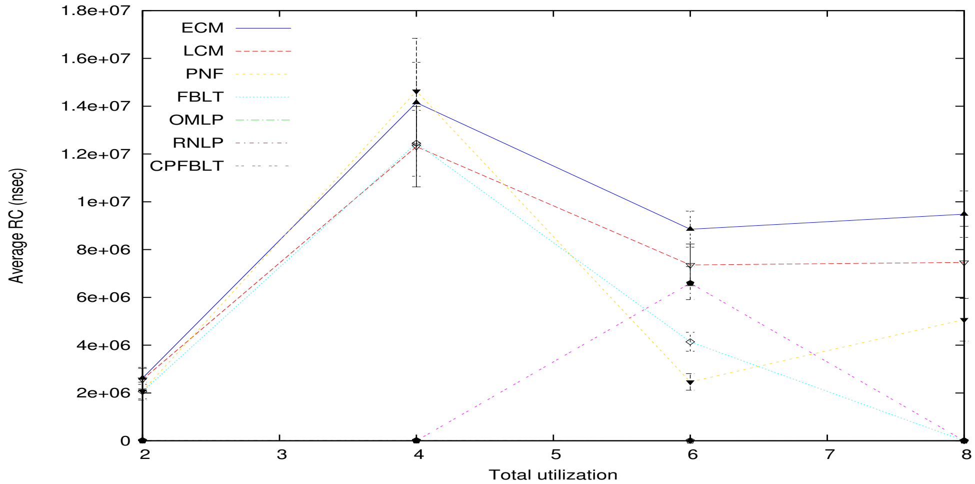


Figure C.263: Avg_RC for Tasksets 263, 533, 803 and 1073

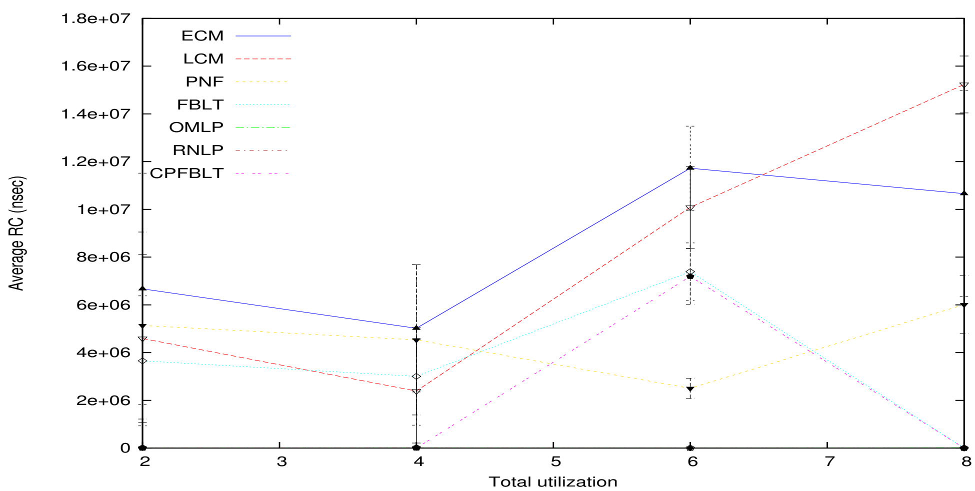


Figure C.264: Avg_RC for Tasksets 264, 534, 804 and 1074

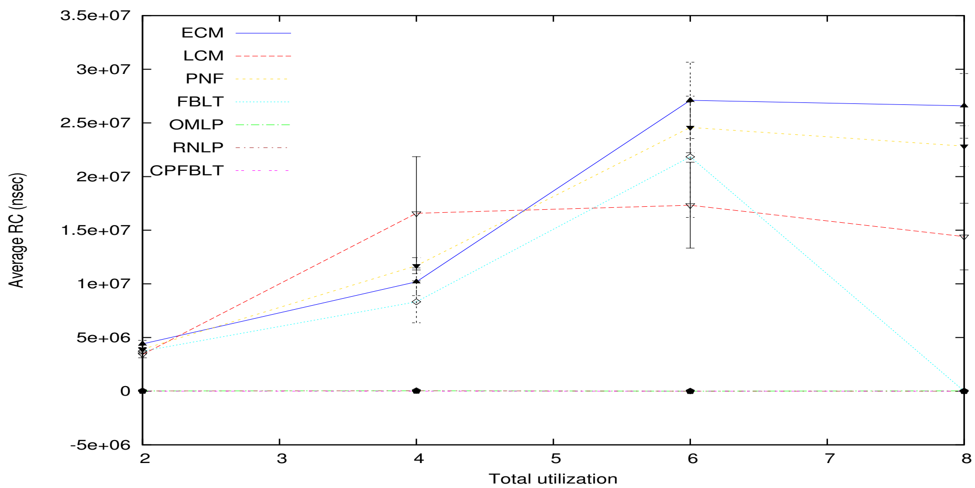


Figure C.265: Avg_RC for Tasksets 265, 535, 805 and 1075

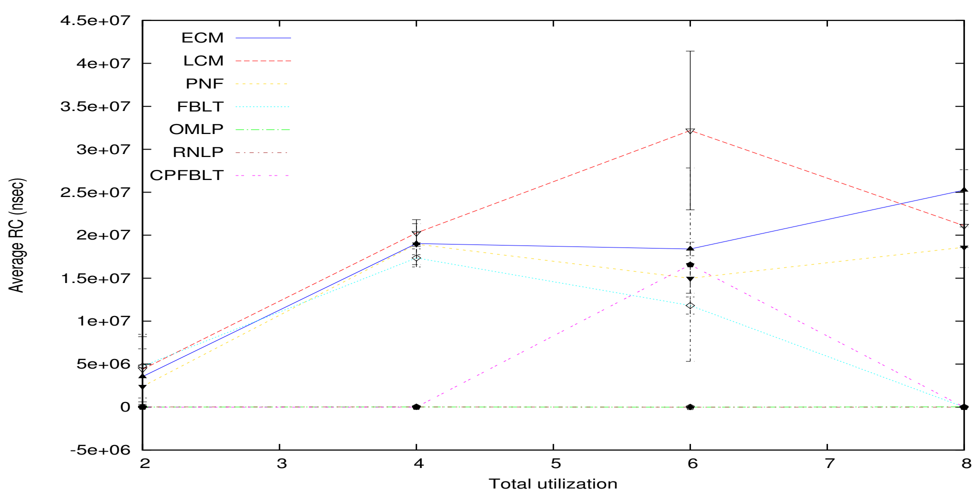


Figure C.266: Avg_RC for Tasksets 266, 536, 806 and 1076

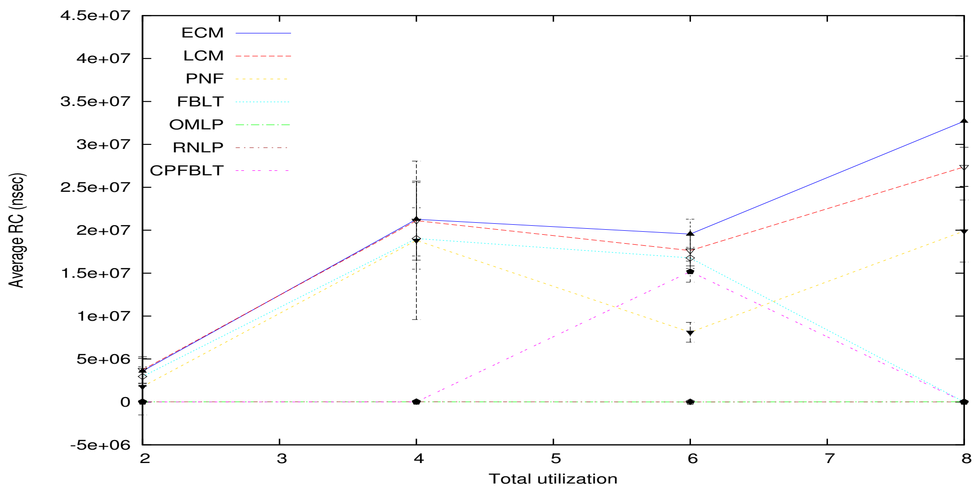


Figure C.267: Avg_RC for Tasksets 267, 537, 807 and 1077

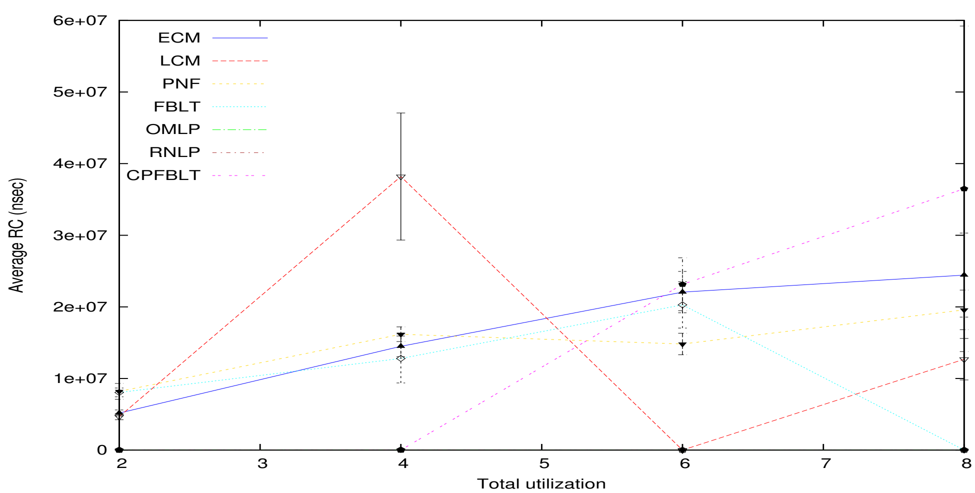


Figure C.268: Avg_RC for Tasksets 268, 538, 808 and 1078

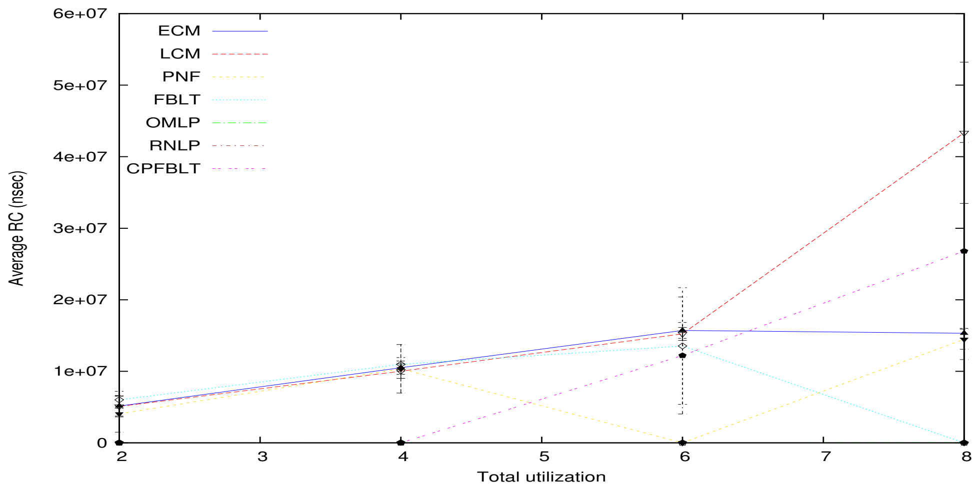


Figure C.269: Avg_RC for Tasksets 269, 539, 809 and 1079

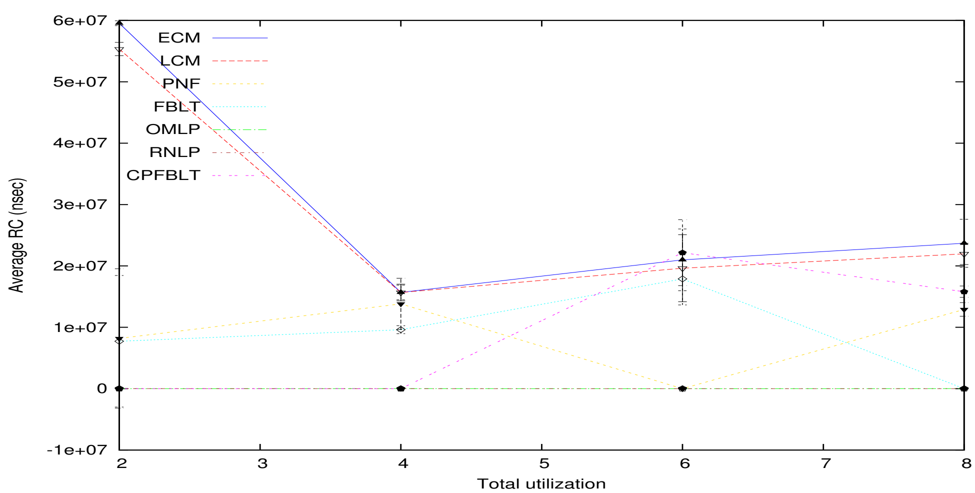


Figure C.270: Avg_RC for Tasksets 270, 540, 810 and 1080