Collaborative Scheduling and Synchronization of Distributable Real-Time Threads

Sherif F. Fahmy

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

Binoy Ravindran, Chair
Tamir A. Hegazy
Y. Thomas Hou
Paul E. Plassmann
Mark T. Jones
Subhash C. Sarin
E. Douglas Jensen

May 5, 2010
Blacksburg, Virginia

Keywords: Distributed Real-Time Scheduling, Software Transactional Memory,
Time/Utility Functions, Utility Accrual Scheduling

Collaborative Scheduling and Synchronization of Distributable Real-Time Threads

Sherif F. Fahmy

(ABSTRACT)

In this dissertation, we consider the problem of scheduling and synchronization of distributable real-time threads — Real-Time CORBA's first-class abstraction for programming real-time, multi-node sequential behaviors. Distributable real-time threads can be scheduled, broadly, using two paradigms: node independent scheduling, in which nodes independently construct thread schedules, based on node-level decomposition of distributable thread (or DT) scheduling parameters, and collaborative scheduling, in which nodes collaborate to construct system-wide thread schedules, which may or may not involve scheduling parameter decomposition.

While significant literature exists on node independent scheduling, little is known about collaborative scheduling and its concomitant tradeoffs. We design three collaborative scheduling algorithms, called ACUA, QBUA, and DQBUA. ACUA uses theory of consensus and QBUA uses theory of quorums for distributable thread schedule construction. DQBUA extends QBUA with lock-based, local and distributed concurrency control. The algorithms consider a model where distributable threads arrive arbitrarily, have time/utility function time constraints, access resources in an arbitrary way (e.g., arbitrary lock acquire/release order, arbitrary nestings), and are subject to arbitrary node crash failures and message losses.

We analytically establish several properties of the algorithms including probabilistic end-to-end termination time satisfactions, timeliness optimality during underloads, bounded exception handling time, and correctness of the algorithms in partially synchronous systems.

We implement distributable real-time threads in the Linux kernel as a first-class programming and scheduling abstraction. The resulting kernel, called ChronOS, provides application interfaces for creating and manipulating distributable threads, as well as kernel interfaces and mechanisms for scheduling them (using both independent and collaborative approaches). ChronOS also has failure detector mechanisms for detecting and recovering from distributable thread failures.

We implement the proposed scheduling algorithms and their competitors in ChronOS and compare their behavior. Our studies reveal that the collaborative scheduling algorithms are superior to independent scheduling algorithms for certain thread sets, in particular, when thread sections have significantly varying execution time. This variability, especially if the variability is not consistent among the threads, may cause each node to make conflicting decisions in the absence of global information. We observe that collaborative schedulers outperform independent schedulers (e.g., EDF augmented with PIP) in terms of accrued utility by as much as 75%.

We identify distributed dependencies as one of the major sources of overhead in collabora-

tive scheduling. In particular, the cost of distributed lock-based concurrency control (e.g., lock management, distributed deadlock detection/resolution) can significantly reduce the problem space for which collaborative scheduling is beneficial. To mitigate this, we consider the use of software transactional memory (or STM), an optimistic, non-blocking synchronization alternative to lock-based concurrency control which has been extensively studied in non real-time contexts. We consider distributable real-time threads with STM concurrency control, and develop techniques for analyzing and bounding their end-to-end response times on distributed single-processor and distributed multiprocessor systems. We also develop contention management techniques, a key component of STM, which are driven by threads' real-time scheduling parameters, and establish their tradeoffs against non-real-time contention managers.

Dedication

I dedicate this dissertation to my family, who were my constant companions through this long and tedious journey. In particular, I dedicate this dissertation to my father — may he rest in peace — whose last words encouraged me to finish my degree as soon as possible. This is for him.

Acknowledgments

This work would never have been possible without the help and support of many people. First, I would like to thank my advisor, Dr. Binoy Ravindran, for his help and guidance for the last half decade. His advise about both technical and non-technical topics have made me an infinitely better person. I would also like to extend a special thanks to Dr. E. D. Jensen. His valuable feedback on the technical aspects of the thesis, and his constant advise on how to develop better writing skills, were invaluable.

I also want to take this opportunity to thank all my colleagues in the real-time lab, for their valuable interaction. The fruitful discussions we had on using the LaTeX typesetting system, kernel programming on POSIX systems, and algorithm design were a very large part of my learning experience. I would particularly like to thank Piyush Garyali and Matthew Dellinger for their work on ChronOS, they created the platform that I built on for my work. I would also like to thank Piyush Garyali for helping me out with paperwork that my remote location made impossible for me to handle personally.

Last, but not least, I would like to thank my family for being there for me when I needed them the most. Without them, this page would probably have been the extent of my dissertation. Thanks for providing me with the environment that allowed me to go on to write the "Conclusions and Future Work" chapter.

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

ACPI Advanced Configuration and Power Interface, page 77

ACUA Asynchronous Consensus-based Utility Accrual scheduling algorithm, page iii

ADSL Asynchronous Digital Subscriber Line, page 104

API Application Programming Interface, page 79

CAS Compare-And-Swap, page 241

CCR Conditional Critical Regions, page 17

CDF Cumulative Distribution Function, page 34

CDMA Code Division Multiple Access, page 222

CM Contention Manager, page 221

CORBA Common Object Request Broker Architecture, page 11

CPU Central Processing Unit, page 74

CUA Consensus-based Utility Accrual scheduling algorithm, page 14

DASA Dependent Activity Scheduling Algorithm, page 29

DBE DASA Best Effort property, page 30

DNS Domain Name Service, page 38

DQBUA Distributed Quorum-Based Utility Accrual scheduling algorithm, page iii

DT Distributable Thread, page iii

FPU Floating Point Unit, page 76

G-EDF Global Earliest Deadline First, page 230

glibc GNU C Library, page 74

GNU GNU's *not* Unix, page 74

GPS Global Positioning System, page 22

gtid Global Thread ID, page 79

HPET High Precision Event Timer, page 77

HUA Handler-assured Utility Accrual scheduling algorithm, page 14

LBESA Locke's Best Effort Scheduling Algorithm, page 19

NIC Network Interface Card, page 56

NPTL Native POSIX Threading Library, page 78

NTP Network Time Protocol, page 78

PCP Priority Ceiling Protocol, page 18

PI Portable Interceptor, page 79

PID Process ID, page 87

PIP Priority Inheritance Protocol, page 18

PM Power Management, page 77

POSIX Portable Operating Systems Interface [for Unix], page 74

PUD Potential Utility Density, page 28

QBUA Quorum-Based Utility Accrual scheduling algorithm, page iii

QoS Quality of Service, page 7

RMS Rate Monotonic Scheduling, page 3

RTNS Real-time Network Simulator, page 205

SMP Symmetric Multi-Processing, page 74

SWETS System Wide Executable Thread Set, page 39

TMAR Thread Maintenance And Repair, page 14

TSC Time Stamp Counter, page 77

UDP User Datagram Protocol, page 83

UTC Universal Time, Coordinated, page 22

WCET Worst Case Execution Time, page 104

Chapter 1

Introduction

In distributed systems, action and information timeliness is often end-to-end—e.g., a causally dependent, multi-node, sensor to actuator sequential flow of execution in networked embedded systems that control physical processes. Such a causal flow of execution can be caused by a series of nested, remote method invocations. It can also be caused by a series of chained, publication and subscription events, caused due to topical data dependencies—e.g., publication of topic $\mathcal{A}$ depends on subscription to topic $\mathcal{B}$; publication of $\mathcal{B}$, in turn, depends on subscription to topic $\mathcal{C}$, and so on. Designers and users of distributed systems, networked embedded systems in particular, often need to dependably reason about — i.e., specify, manage, and predict — end-to-end timeliness.

Some emerging networked embedded systems are dynamic in the sense that they operate in environments with dynamically uncertain properties (e.g., [21]). These uncertainties include context-dependent activity execution times and arrival patterns (causing transient and sustained resource overloads), arbitrary activity resource access behaviors, and arbitrary node failures and message losses. Reasoning about end-to-end timeliness is a very difficult and unsolved problem in such dynamic uncertain systems. Despite the uncertainties, such applications desire the strongest possible assurances on end-to-end activity timeliness behavior.

Maintaining end-to-end properties (e.g., timeliness, connectivity) of a control or information flow requires a model of the flow's locus in space and time that can be reasoned about. Such a model facilitates reasoning about the contention for resources that occur along the flow's locus and resolving those contentions to optimize system-wide end-to-end timeliness. The *distributable thread* programming abstraction, which first appeared in the Alpha OS [29] and subsequently in Mach 3.0 [52] (a subset), MK7.3 [134], Real-Time CORBA 2.0 [105], and the emerging Distributed Real-Time Specification for Java (DRTSJ) [10], directly provides such a model as their first-class programming and scheduling abstraction. A distributable thread is a single thread of execution with a globally unique identity that transparently extends and retracts through local and remote objects.

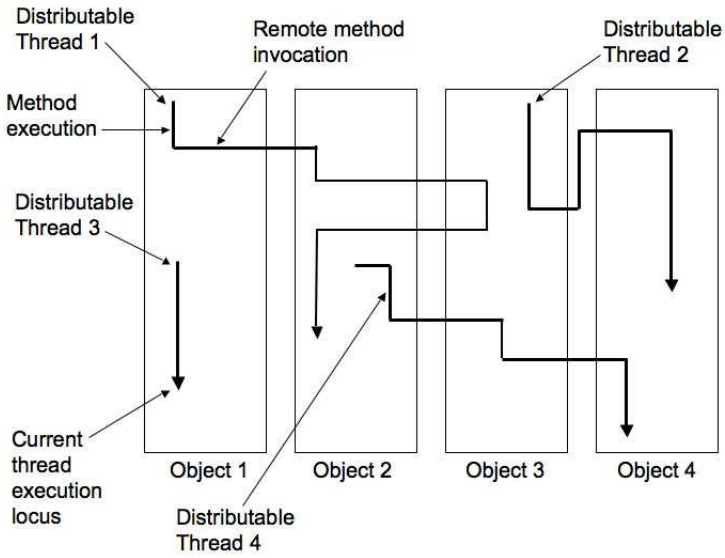

Figure 1.1: Four Distributable Threads

A distributable thread carries its execution context as it transits node boundaries, including its scheduling parameters (e.g., time constraints, execution time), identity, and security credentials. The propagated thread context is intended to be used by node schedulers for resolving all node-local resource contention among distributable threads such as that for node's physical (e.g., processor, I/O) and logical (e.g., locks) resources, according to a discipline that provides application specific, acceptably optimal, system-wide end-to-end timeliness. Figure 1.1 shows the execution of four distributable threads. We focus on distributable threads as our end-to-end control flow/programming/scheduling abstraction, and hereafter, refer to them as *threads*, except as necessary for clarity. When referring to a single-node thread, we explicitly distinguish them from distributable threads.

1.1 Timeliness Model

Application time constraints are expressed and handled in current real-time practice (e.g., Real-Time CORBA [105], POSIX [74], Ada 95 [11] and RTSJ [138]) using the *priority* artifact. Priorities have significant shortcomings, including the following: (1) Mapping application time constraints to priorities is generally not tractable. Doing so, results in significant loss of information, thereby making it difficult to dependably satisfy time constraints; (2) Priority assignments are not modular for expressing urgency, because they require global knowledge of all priority assignments, which is often difficult to obtain during system development (e.g., due to organizational boundaries); and (3) Urgency of an application activity is sometimes orthogonal to the activity's relative importance, but a priority cannot express both (urgency and importance). This causes difficulties in managing resources during

resource overloads, when completing activities that are more functionally important than those which are more urgent is often desirable.

Traditional real-time computing theory [80, 92] overcomes these shortcomings by providing application designers with the direct abstraction of time constraints (instead of mapping time constraints to priorities), and using that abstraction for resource management. However, that theory is fundamentally limited to the deadline time constraint. In conventional real-time theory, deadlines are mapped to fixed priorities in algorithms such as RMS [90] and DMA [90], or are directly used for scheduling and resource management in algorithms such as EDF [73]. Deadlines and deadline-based scheduling have the following drawbacks:

(1) A deadline is either (i) a binary-valued expression in the sense that it is either met or not met; or (ii) a linear-valued expression for the penalty of lateness—i.e., the penalty of being late per unit time is constant regardless of the activity's lateness. Thus, deadlines also cannot distinguish between urgency and importance — a serious limitation during overload situations (as explained previously);

(2) Classical deadline-based scheduling [73] suffers from the (counter-intuitive) *domino* effect during overloads [93], as deadline-based algorithms (during overloads) favor activities that have a high likelihood for missing their deadlines over those that have a low likelihood for doing so.

(3) Deadlines cannot express time constraints that are non-binary and non-linear in the sense that the utility attained for activity completion *varies* (e.g., increases, decreases) with the activity completion time.

Timeliness optimality criteria that can be expressed using deadlines thus fall into the following classes: (1) the hard real-time criterion of satisfying all deadlines; (2) criteria that are expressed using number of missed deadlines (e.g., minimize deadline misses, satisfy an upper bound on missed deadlines); and (3) criteria that are expressed using lateness (e.g., minimize maximum lateness, satisfy an upper bound on lateness).

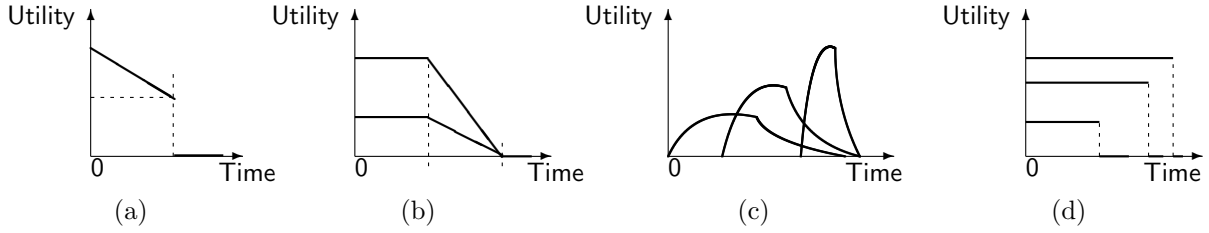

Figure 1.2: Example TUF Time Constraints. (a): MITRE Airborne Warning and Control System (AWACS) *association* TUF [30]; (b-c): GD/CMU Air defense *plot correlation*, *track maintenance*, & *missile control* TUFs [100]; (d): step TUFs.

The *time/utility function* (or TUF) timeliness model [75] overcomes these shortcomings. A TUF, which is a generalization of the deadline constraint, specifies the utility to the system

resulting from the completion of an activity (e.g., a thread) as a function of that activity's completion time. Figure 1.2 shows example time constraints from real applications specified using TUFs. Figures 1.2(a)–1.2(c) show some time constraints of two applications in the defense domain [30, 100]. Classical deadline is a binary-valued, downward "step" shaped TUF; Figure 1.2(d) shows examples.

When thread time constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing accrued thread utility—e.g., maximizing the total thread accrued utility. Such criteria are called *utility accrual* (or UA) criteria, and sequencing (scheduling, dispatching) algorithms that optimize UA criteria are called UA sequencing algorithms (e.g., [31, 93]).

UA algorithms that maximize total utility under downward step TUFs (e.g., [31, 93]) default to EDF during underloads, since EDF satisfies all deadlines during underloads. Consequently, they obtain the optimum total utility during underloads. During overloads, they inherently favor more important threads over less important ones (since more utility can be attained from the former), irrespective of thread urgency, and thus exhibit adaptive behavior and graceful timeliness degradation. This behavior of UA algorithms is called "best-effort" [93] in the sense that the algorithms strive their best to feasibly complete as many high importance threads — as specified by the application through TUFs — as possible.[1] Consequently, high importance threads that arrive at any time always have a very high likelihood for feasible completion (irrespective of their urgency). Note also that EDF's optimal timeliness behavior is a special-case of UA scheduling.

1.2 Distributed Scheduling

In this dissertation, we consider the problem of scheduling distributable threads in the presence of the previously mentioned uncertainties, focusing particularly on (arbitrary) node failures and message losses. Past efforts on thread scheduling (e.g., [29, 34, 114, 115]) can be broadly categorized into two classes: *independent node scheduling* and *collaborative scheduling*. In the independent scheduling approach (e.g., [29, 34, 115]), threads are scheduled at nodes using propagated thread scheduling parameters and without any interaction with other nodes (thereby not considering node failures during scheduling). Fault-management is separately addressed by *thread integrity protocols* [55] that run concurrent to thread execution. Thread integrity protocols employ failure detectors (abbreviated here as FDs), and use them to detect failures of the thread abstraction, and to deliver failure-exception notifications [29, 34]. In the collaborative scheduling approach (e.g., [114]), nodes explicitly cooperate to construct system-wide thread schedules, anticipating and detecting node failures using FDs.

[1] Note that the term "best effort" as used in the context of networks actually is intended to mean "least effort."

There are tradeoffs between the two approaches: Independent node scheduling avoids the overhead of inter-node communication, and is therefore message-efficient (from the thread scheduling standpoint). However, the approach poses theoretical difficulties in establishing end-to-end timing assurances, due to the complex (and concurrent) interaction between thread scheduling and thread fault-management mechanisms. This is overcome in collaborative scheduling, but the approach incurs message overhead costs. In [34, 115], upper bounds are established for such message costs.

FDs that are employed in both paradigms in past efforts have assumed a totally synchronous computational model—e.g., deterministically bounded message delay. While the synchronous model is easily adapted for real-time applications due to the presence of a notion of time, as pointed out in [68], this results in systems with low coverage. On the other hand, it is difficult to design real-time algorithms for the asynchronous model due to its total disregard for timing assumptions. Thus, there have been several (recent) attempts to reconcile these extremes. For example, in [4], Aguilera *et. al.* describe the design of a fast failure detector for synchronous systems and show how it can be used to solve the consensus problem for real-time systems. The algorithm achieves the optimal bound for both message and time complexity for synchronous systems.

In [69], Hermant and Widder describe the *Theta-model*, where only the ratio, Θ, between the fastest and slowest message in transit is known. This increases the coverage of algorithms (designed under this model) as less assumptions are made about the underlying system. While Θ is sufficient for proving the correctness of such algorithms, an upper bound on communication delay is needed to establish timeliness properties.

To summarize, there are different approaches on how to achieve end-to-end timeliness in distributed systems. These approaches can be either collaborative or independent and can be deployed on systems with different levels of "synchronicity." Thus, the central and fundamental question that we ask (and answer) in this dissertation is whether or not it is possible to design collaborative, distributable thread scheduling algorithms on partially synchronous systems. Further, what are the different ways to efficiently design such algorithms, so that their end-to-end timing properties in the presence of failures can be formally established. Furthermore, how do such algorithms perform with respect to each other and with respect to state-of-the-art independent schedulers? In addition, under what conditions their higher overhead is justifiable and what are the properties of applications (e.g., time constraints, execution times) that will benefit from them?

1.3 Synchronization

When distributable threads share resources, which are subject to mutual exclusion constraints, two types of synchronization problems arise. These include 1) local dependencies and 2) distributed dependencies. In the former, all locks are released before a thread makes

a remote invocation, which can cause threads to depend on another, at the node—i.e., a thread needs a lock which is held by another thread on the same node. In the latter, a thread makes a remote invocation while holding a lock, which can cause threads to depend on another remotely—i.e., a thread needs a lock which is held by a remote thread. Local dependencies can cause local deadlocks (e.g., threads $\mathcal{A}$ and $\mathcal{B}$ block on each other for locks held by the other on the same node), and distributed dependencies can cause distributed deadlocks (e.g., threads $\mathcal{A}$ and $\mathcal{B}$ remotely block on each other for locks held by the other).

Local dependencies and deadlocks are traditionally addressed using lock management and deadlock management protocols. In centralized real-time systems, this typically involves constructing dependency chains, which determines which thread is blocked on which other, and inflating the execution eligibility of lock-holding threads by the aggregate eligibility of those threads that they block. For example, Priority Inheritance Protocols [123] raise the priority of the lock holder with that of the blocked thread; TUF/UA schedulers (e.g., DASA [31], GUS [89], HUA [115]) increase the utility density of the lock holder with that of the blocked thread.[2] By doing so, the blocking times of the blocked threads are reduced.

Deadlocks are managed either by run-time detection (e.g., by cycle detection algorithms) and resolution (e.g., by aborting a deadlocked thread) such as in Priority Inheritance Protocols [123], DASA [31], GUS [89], and HUA [115], or by prevention/avoidance such as in Priority Ceiling Protocols [56] and Stack Resource Policy [14]. While detection/resolution requires little a-priori knowledge on threads' resource sharing behavior, avoidance/prevention requires significant a-priori knowledge (e.g., which threads require which locks, and in what order) [127]. Given the dissertation's highly dynamic model including the uncertainty on threads' lock acquire/release order, we rule out deadlock avoidance/prevention. The dissertation thus considers deadlock detection/resolution. This also implies a termination model for threads: when exceptions are raised on threads (e.g., due to deadlocks, time constraint violations, etc.), we assume that they can be aborted and the system safety can be ensured by executing application-specified thread exception handlers.

Lock management and deadlock management for distributed dependencies are significantly complex. Traditional methods for distributed deadlock management [20, 28, 36, 37, 42, 83, 84, 102, 116] break down for real-time systems [125]. In non real-time distributed systems, deadlocks are a stable condition—i.e., once a deadlock occurs, the system does not make further progress. Distributed deadlock detection algorithms such as edge chasing deadlock detection algorithms depend on this property for their correctness [125]. However, in real-time systems, this is not true. Under a termination model, once a deadlock occurs, it is automatically resolved when the thread with the earliest time constraint, which is deadlocked, violates its time constraint: the thread is aborted for handling the time constraint violation exception, which also resolves the deadlock. Thus, a large number of trade-offs must be considered in designing real-time distributed deadlock detection and resolution algorithms [125].

[2]Similar approaches are also considered in non real-time contexts. For e.g., in lottery scheduling [137], lotteries won by blocked threads are transferred to the lock holders to increase their execution eligibility.

Alternatives to lock-based concurrency control may be a possible solution to this problem. For example, non-lock based concurrency control solutions are not subject to deadlocks [57]. This is beneficial from two points of view. First, it reduces the complexity of the scheduling algorithm in terms of both its semantic complexity (by freeing it from dealing with deadlocks) and time complexity (by eliminating the need for deadlock detection and resolution, which is one of the major contributors to scheduling time complexity). Second, non-lock based concurrency control can significantly increase the semantic simplicity of the application code, thus improving programmer productivity [57]. After considering several alternatives to lock based concurrency control, are briefly overviewed in Section 2.2, we concluded that software transactional memory (STM) would be most suited for our purpose.

Using STM as a concurrency control mechanism for distributed real-time systems raises several fundamental questions. These include, how to analyze and bound (end-to-end) response times of distributable threads with STM concurrency control on distributed single processor systems? How to do so on distributed multiprocessor systems? Is it possible to do real-time contention management (a key component of STM)? What are its tradeoffs against non-real-time contention management?

1.4 Research Contributions

Thus, the dissertation's problem space include:

1. Can timeliness assurances be provided for non-synchronous systems that are subject to crash failures?

2. What are the tradeoffs between collaborative scheduling and independent scheduling? and

3. Is it possible to reduce the overhead of collaborative scheduling to improve the timeliness properties of the algorithms?

We answer all these questions in the dissertation.

First, we design a collaborative scheduling algorithm called ACUA (*Asynchronous Consensus-based Utility accrual scheduling Algorithm*), using the theory of consensus [51, 111]. Doing so requires solving the consensus problem on a partially synchronous system. The partially synchronous system we target is one where both communication delay and communication failure are described stochastically [26]. On top of this system, we design an $\mathcal{S}$-class failure detector and use that to develop a consensus algorithm. We then use the consensus algorithm as the basis for ACUA. We analytically establish several properties of ACUA. These include probabilistic end-to-end termination time satisfactions, timeliness optimality during underloads and bounded exception handling time. We also show that ACUA can provide better timeliness, when compared to independent scheduling, for systems that can tolerate its (relatively) higher overhead.

To improve ACUA's overhead, we design another collaborative scheduling algorithm called QBUA (*Quorum-Based Utility Accrual scheduling*), using the theory of quorum [5, 106]. QBUA uses the partially synchronous model and QoS failure detectors described in [26]. We analytically establish several properties of QBUA, showing that it retains all the properties of ACUA while having a better message complexity that does not scale with the number of failures in the system. We extend QBUA with lock-based concurrency control. The resulting algorithm is called DQBUA (*Distributed Quorum-Based Utility Accrual scheduling algorithm*).

To understand how collaborative scheduling algorithms can be implemented and understand their empirical performance, we implemented distributable threads as a first-class programming and scheduling abstraction in the Linux kernel. We call the resulting kernel, ChronOS. Major features of ChronOS include application programming interfaces for creating and manipulating distributable threads, and kernel-level interfaces and mechanisms for implementing distributable thread schedulers, in both the independent and collaborative scheduling approaches. ChronOS also includes failure detectors for detecting and recovering from distributable thread failures, and a co-scheduling approach for optimizing distributable thread scheduling.

We implemented ACUA, QBUA, and DQBUA and their state-of-the-art competitors (in the independent scheduling approach) including, HUA [115], CUA [114], independent scheduling with EDF [73] augmented with the Priority Inheritance Protocol and independent scheduling with RMS [90] augmented with the Priority Inheritance Protocol in ChronOS.

We conducted extensive experimental studies using this implementation. Our studies reveal that collaborative algorithms can perform better than independent algorithms for thread sets that can a) tolerate their higher overhead and b) whose section execution times vary significantly, thus resulting in possibly conflicting scheduling decisions on different nodes. For such thread sets, the collaborative algorithms outperform their competitors. For example, ACUA has a maximum performance improvement over HUA in the range of about 6% to 39% depending on the thread set used, while QBUA has a maximum performance improvement over CUA in the range of about 5% to 21% depending on the thread set used. For the case of dependencies, DQBUA outperforms HUA by as much as 10% to 16% depending on the thread set used, outperforms independent RMS by as much as 18% to 27%, and independent EDF by as much as 42% to 75%.

Finally, we consider STM as an alternative, cost-effective concurrency control technique for distributable real-time threads. We developed techniques for analyzing and bounding their end-to-end response times on distributed single-processor systems scheduled using deadline schedulers. We extended these techniques for distributed multiprocessor systems scheduled using proportional schedulers.

We also developed contention management techniques, a key component of STM, which are driven by threads' real-time scheduling parameters, and establish their tradeoffs against non real-time contention managers. We show that real-time contention management techniques

can be used to tighten the retry bounds of transactions. Our results indicate that using a real-time contention manager allows systems to perform as well as using the non-real-time aggressive contention manager. This result holds even when the retry costs of the transactions using real-time contention managers are up to two times the retry overhead of the transactions using the aggressive contention manager.

To summarize, the dissertation's research contributions include:

1. We designed a class of collaborative scheduling algorithms (ACUA, QBUA, and DQBUA) for scheduling distributable real-time threads. The algorithms operate in partially synchronous systems where the communication delay and message loss are stochastically described, timeliness constraints are expressed using TUFs, crash failures are possible, overloads can occur, and threads can access resources arbitrarily. This class of schedulers performs best when the thread sections of distributable threads have significantly varying execution times as this leads to possible local minima in scheduling decisions that can only be overcome by collaboration. This is the first such result.

2. We implemented an extension of the Linux kernel, which we call ChronOS, that implements distributable threads as a first-class programming and scheduling abstraction, implements collaborative and independent schedulers at the kernel level, and adopts a co-scheduling approach to optimize collaborative scheduling. To the best of our knowledge, this is the only contemporary OS that supports distributable real-time threads as a first-class abstraction for programming and scheduling real-time, multi-node sequential behaviors.

3. We designed end-to-end response time analysis techniques under STM concurrency control for distributed single- and multiprocessor systems — the first such techniques. These techniques allow, for the first time, STM as a concurrency control technique in distributed real-time systems. We also develop real-time contention managers and show how they can be used to tighten the retry bounds of transactions.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows, in Chapter 2, we provide a brief review of the relevant literature. Chapter 3 presents ACUA and shows how it can provide better timeliness assurances than independent scheduling algorithms. In Chapter 4, we attempt to reduce the overhead of the consensus-based algorithm designed in Chapter 3 by using a quorum-based approach. In Chapter 5, the quorum-based approach is extended to deal with distributed dependencies.

The ChronOS kernel is described in Chapter 6 and the experiments conducted on this platform to evaluate the performance of the collaborative scheduling algorithms are presented in Chapters 7 and 8.

We present our case for using software transactional memory (or STM) in Chapter 9, provide

an overview of the requirements of incorporating STM into real-time systems and propose a set of problems we can solve to make STM in distributed real-time systems a reality. In Chapters 10 and 11, we present response time analysis for uniprocessor and multiprocess distributed systems programmed using STM respectively. Chapter 12, presents our work on real-time contention managers for STM. We conclude the dissertation in Chapter 13.

Chapter 2

Past and Related Work

As mentioned in Chapter 1, the main aim of this dissertation is to study the scheduling problem for real-time distributed systems. Specifically, the main aims of the research being conducted are; 1) Determine the possibility of providing timeliness assurances in non-synchronous systems that are subject to failures, 2) Investigate the properties of collaborative scheduling, 3) Identify the major performance bottlenecks in collaborative scheduling, and 4) Attempt to overcome these bottlenecks.

2.1 Distributed Scheduling

There have been a number of papers published in the literature addressing fault tolerant distributed real-time scheduling. In this chapter, we provide a brief overview of the publications most relevant to our current work. As mentioned in Chapter 1, there are two main approaches to scheduling threads on distributed real-time systems; collaborative and independent. Most past work has focused on independent scheduling due to its simplicity and low overhead.

2.1.1 Independent Scheduling

In independent scheduling, each node in a distributed system schedules the threads it hosts without recourse to communication with other nodes. Therefore, it was simple to extend the state of the art in single node scheduling to accommodate this model. A task making a remote invocation propagates its scheduling parameters to its destination node. The destination node then uses these propagated scheduling parameters to perform its own local scheduling.

Due to its simplicity and relatively low overhead, this approach has been incorporated into many real-time distributed programming standards. For example, Real-Time CORBA [105]

makes extensive use of this approach. However, even within the independent scheduling approach, there are a number of different factors to consider.

One of the most elusive factors (in terms of optimality) is the question of how to derive local scheduling parameters from global scheduling parameters. Specifically, since we are considering real-time systems, the question becomes how to derive local deadlines from global deadlines in order to ensure optimal system performance in terms of deadlines met. There are many different methods for decomposing global end-to-end deadlines in order to derive local deadlines. It is possible to use the end-to-end, or global, deadline to perform local scheduling, however, this approach may underestimate the urgency of components of end-to-end abstractions (since each component is given the urgency of the entire end-to-end abstraction). This may result in excessive delay to said components and thus to deadline misses.

Other approaches include dividing the end-to-end deadline equally among all its component sections, or dividing the end-to-end deadline in proportion to the execution times of each component. As mentioned before, the "best" method for decomposing end-to-end deadlines is elusive and depends on the application being considered and many other different heuristics. The same set of issues arise if we consider TUFs (see Chapter 1) as our timeliness abstraction. There have been a number of papers addressing this issue [76,77,86,118]. Note that deadline (or TUF) decomposition is essential for independent node scheduling, since it is this technique that allows an end-to-end scheduling problem to be broken down to a series of independent scheduling problems.

Deadline, or TUF, decomposition is not the only issue addressed by researchers in the distributed real-time field. Other important issues include synchronization protocols to ensure precedence constraints are met, scheduling algorithms, and the development of sufficiently tight schedulability analysis for the scheduling and synchronization protocols developed. A good example of previous work that investigates these issues for the independent scheduling paradigm is Sun's thesis [130].

Sun's work [130] is a classic example of an integrated distributed real-time system solution that uses independent scheduling. The author decomposes the problem of end-to-end scheduling into three separate sub-problems. Deriving priorities for tasks, deriving appropriate release times (the author uses the term execution synchronization) to ensure that precedence constraints are met, and then, finally, performing schedulability analysis on the tasks on each node independently using the derived release times. The response times of sub-tasks belonging to an end-to-end computation abstraction are then summed up to give an upper bound on the end-to-end response time. It should be noted that Sun's approach [130] involves "static release" of sub-tasks (i.e. the release times are computed offline and are enforced during run-time). This contrasts with the "dynamic release" approach where release times are not set offline, but tasks are released upon the receipt of an invocation call – through mechanisms such as RPC or RMI calls – at run-time. This approach is generally considered less flexible than the dynamic approach since its static release nature makes it

rather restricting for modern systems where dynamism is a primary feature. In addition, the statically derived release times are subject to pessimism.

Another work that has addressed the problem of end-to-end scheduling of programming abstractions in distributed real-time systems is [17]. In this paper, the authors consider the flow-shop version of the end-to-end scheduling problem, in which all tasks executed on different processors in the same order. The authors identify two tractable versions of this problem, and suggest a heuristic method for the general NP-hard instance of the problem. The paper, however, does not address fault tolerance or non-synchronous systems. Nor does it consider the behavior of systems during overloads or the possible benefits of collaborative scheduling. All of these factors are taken into consideration in this dissertation.

In [78], Kao *et. al.* present an approach to scheduling soft real-time systems using commercial off the shelf components. However, none of the algorithms used in the paper are specifically designed for scheduling soft real-time systems during overload and hence do not offer reasonable assurances in such scenarios.

Not all the papers on distributed real-time systems have focused on deadline (or TUF) decomposition, and distributed scheduling algorithms. A number of papers have addressed the important issue of developing the necessary schedulability analysis techniques to provide suitable timeliness assurances.

One of the first papers to address the development of scheduling analysis techniques for distributed real-time systems is [135]. In this seminal paper, Tindell *et. al.* [135] develop a method for analyzing distributed real-time systems that many later approaches have built on. The main difficult of schedulability analysis on distributed systems is the fact that the start times of some tasks will depend on the end times of others (since some tasks are invoked when their predecessor task completes and makes a remote invocation).

In order to solve this problem, Tindell *et. al.* [135] proposed the idea of initially setting the start times of a task to the earliest possible completion time of its predecessor task (assuming that each task executes on the processor uninterrupted by other tasks), and then iteratively adjusting these start times as the analysis yields more accurate completion times for tasks in the system. This iterative process is guaranteed to converge if the recurrence relations being iterated over are monotonic in their parameters.

Since Tindell *et. al.*'s seminal paper [135], several authors have attempted to refine the algorithms and analysis techniques to provide tighter bounds on response times in distributed systems. For example, in [62, 108, 109], the authors propose methods for using jitters and offsets to represent the programming models on a distributed system. The proposed methods allow the authors to develop response time analysis algorithms that can provide tighter bounds on the response times of tasks than previous algorithms. This trend is continued in [112], where the authors attempt to provide an even tighter bound on response times by eliminating the need for jitters. We make use of some of these results while developing our own schedulability analysis techniques for STM based distributed real-time systems in

Chapters 10 and 11.

Other previous studies that are relevant to our work include [6–8, 72] which describe schedulability analysis algorithms for stand alone systems programmed using lock-free concurrency control mechanisms. These papers have a relevance to our own STM schedulability analysis algorithms with the major difference being that we concentrate on distributed systems while the papers mentioned address stand alone systems only.

Most of the previous work on distributed real-time systems did not consider either fault tolerance or non-synchronous distributed systems. As previously mentioned, newly emerging distributed systems operate in domains where faults are possible and the system is not always synchronous. One of the proposed directions in this dissertation is to determine whether or not it is possible to provide some sort of timeliness assurances in such environments.

Despite the fact the few papers address fault tolerance, there have been some research addressing this issue. For example, the Alpha Kernel was built with fault tolerance in mind. Specifically, the concept of *thread maintenance and repair* (TMAR), was used to monitor the health of end-to-end threads in distributed systems and to recover in case of failure. Chapter 5 in [54] discusses the different algorithms and protocols that can be used to provide TMAR in the Alpha kernel.

More recently, Ravindran *et. al.* [115] develop HUA, an independent node scheduling algorithm for synchronous systems that uses propagated thread scheduling information to perform local scheduling. HUA uses TMAR for fault tolerance and is assumed to execute in a synchronous environment. The problem with independent scheduling is that it achieves its encouragingly low overhead by limiting the information available at each node. Specifically, each node has information about the tasks it hosts only. This lack of information about what is happening on other nodes may result in some decisions that are locally optimal but that compromise global optimality. In this dissertation, we study this issue and attempt to qualify the scenarios under which collaborative scheduling is preferable to independent scheduling and whether the significantly higher overhead of collaborative scheduling is justified. This issue is discussed in Chapters 3, 4 and 5.

2.1.2 Collaborative Scheduling

There have been some recent attempts at designing collaborative scheduling algorithms (e.g., [45, 47, 48, 114], etc). In collaborative scheduling algorithms, scheduling decisions are arrived at after collaboration among the nodes in the distributed system. This collaboration allows better timeliness assurances since nodes can now make informed scheduling decisions to ensure global optimality. However, these algorithm have higher overheads than independent scheduling algorithms and can only benefit systems that can tolerate their higher overhead. In [114], a scheduling algorithm for synchronous systems that uses the collaborative scheduling approach, CUA, is developed.

This algorithm uses fast consensus [4] to solve the scheduling problem. The other collaborative scheduling algorithms investigated in this dissertation (ACUA [48] , QBUA [45] and DQBUA [47]) are discussed in Chapters 3, 4 and 5 respectively. Another advantage of collaborative scheduling algorithms is that, since global state is shared, at least to some extent, it becomes possible to handle fault tolerance as an integral part of the scheduling algorithm instead of relying on orthogonal thread integrity protocols (such as TMAR). This offers the opportunity to provide better assurances about the behavior of the system in the presence of failures.

From our research in this domain, one fact has become quite obvious. Concurrency control contributes significantly to the complexity of real-time distributed systems. This complexity encompasses both the time complexity of the scheduling algorithms and the semantic complexity of writing correct, deadlock-free code using the standard method of locks and condition variables. We discuss this difficulty and make our case for using an alternative to locks, software transaction memory (or STM), in Chapter 9, however, in Sections 2.2 and 2.2.1 we review the literature on this topic.

2.2 Alternatives to Lock-Based Programming

Academia, and certain parts of industry, have realized the limitations of lock-based software, thus a number of proposed alternatives to lock-based software exist. The design of lock-free, wait-free or obstruction-free data structures is one such approach. The main problem with this approach is that it is limited to a small set of basic data structures, e.g., [8, 27, 64]. For example, to the best of our knowledge, there is no lock-free implementation of a red-black tree that does not use STM (this does not imply that it is impossible to do so, it is indeed possible, but merely indicates that the difficulty of designing such a complex data structure from basic principles has discouraged researchers from attempting it). Most of the literature on lock-free data structures concentrates on basics such as queues, stacks, and other simple data structures. It should be noted that lock-freedom, wait-freedom and obstruction-freedom are concepts and as such can encompass non lock-based solutions like STM. However, we use these terms in this context to refer to hand crafted code that allows concurrent access to a data structure without suffering from race conditions.

The discrete event model presented in [143, 144] provides an interesting alternative to thread based programming. While interesting and novel, it still remains to be seen whether programmers find the semantics of the model easier than the semantics of thread-based computing. In addition, the requirement of static analysis to determine a partial order on the events makes the system inapplicable to dynamic systems where little or no information is available a priori.

Transactional processing, the semantic ancestor of STM, has been around for a significant period of time and has proven its mettle as a method of providing concurrency control in

numerous commercial database products, in addition, it does not place any restriction on the dynamism of the system on which it is deployed. Unfortunately, the use of a distributed commit protocol, such as the two-phase commit protocol, increases the execution time of a transaction and can lead to deadline misses [60]. STM is a lighter-weight version of transactional processing, with no distributed commit protocol required in most cases. As such, it allows us to gain the benefits of transactional processing (i.e., fault tolerance and semantic simplicity), without incurring all its associated overhead.

We believe that STM is an attractive alternative to thread and lock-based distributed programming, since it eliminates many of the conceptual difficulties of lock-based concurrency control at the expense of a justifiable overhead that becomes less significant as the number of processors in the system scales.

2.2.1 Software Transactional Memory

Since the seminal papers about hardware and software transactional memory were published, renewed interest in the field has resulted in a large body of literature on the topic (e.g, see [18, 82, 98]). This body of work encompasses both purely software transactional memory systems and hybrid systems where software and hardware support for transactional memory are used in conjuncture to improve performance. Despite this large body of work, to the best of our knowledge, only three papers investigate STM for distributed systems [19, 67, 96].

We believe that distributed embedded systems stand to benefit significantly from STM. Such systems are most distinguished by their need to: 1) react to external events asynchronously and concurrently; 2) react to external events in a timely manner (i.e., real-time); and 3) cope with failures (e.g., processors, networks) – one of the raison d'être for building distributed systems. Thus, concurrency that is fundamentally intrinsic to distributed embedded systems naturally motivates the usage of STM. Their need to (concurrently) react timely to external events in the presence of failures is also a compelling reason – such behaviors are very complex to program, reason about, and obtain timing assurances using lock-based concurrency control mechanisms.

There has also been a dearth of work on real-time STM systems. Notable work on transactional memory and lock-free data structures in real-time systems include [6–8, 72, 97]. However, most of these works only consider uni-processor systems (with [72] being a notable exception). In this chapter, we propose to study the issues involved in implementing STM in distributed embedded real-time systems. Past work has shown that STM has lower throughput for systems with a small number of processors compared to fine-grain lock-based solutions but that this difference in performance is quickly reversed as the number of processors scales [63]. This, coupled with easier programming semantics of STM, makes it an attractive concurrency control mechanism for next generation embedded real-time systems with multi-core architectures and high distribution.

With STM, deadlocks are entirely or almost entirely precluded. This will immediately result in significant reductions in the cost of scheduling and resource management algorithms, as distributed dependencies are avoided and no expensive deadlock detection/resolution mechanisms are needed. Implementing higher level programming constructs, like, for example, Hoare's conditional critical regions (or CCR) [71], on top of STM [63], allows programmers to take advantage of the deadlock freedom and simple semantics of STM in their programs.

2.3 Summary

2.3.1 Distributed Scheduling

In this section, we briefly summarize the properties of various distributed real-time scheduling algorithms and specify the competitors we will be comparing against. We compare the algorithms based on three different criteria, namely 1) the ability to tolerate message loss, 2) the ability to tolerate node failures and 3) the ability to tolerate the presence of local minima. The result of this comparison is shown in Figure 2.1. The first two criteria are self explanatory, therefore we concentrate on describing the third.

As mentioned before, independent scheduling involves each node in the system making a decision based on the information it has locally. It cannot know the state of other nodes while reaching its scheduling decision. Therefore, it is possible for a particular node to make a locally optimal decision i.e., choose a specific thread that has the highest ability to accrue utility locally but that is far from being the optimum thread on a system-wide basis. Naturally, this scenario can only occur when the scheduling parameters of different sections of the same thread vary to such a degree that while an individual section may be the best local section to execute, the entire thread itself may accrue little utility thus resulting in the local node making a wrong decision. We call this scenario the "presence of local minima" and specify whether a distributed scheduling algorithm can handle them or not.

U>1 / U<=1	Case 2 RMS	Case 2 EDF	Case 2 LBESA	HUA/Case 2 DASA	CUA/ACUA	QBUA/DQBUA
Node failure	No / No	No / Yes	Yes / Yes	Yes / Yes	Yes / Yes	Yes / Yes
Message loss	No / No	No / No	No / No	No / No	Yes / Yes	Yes / Yes
Presence of local minima	No / No	No / Yes	No / Yes	No / Yes	Yes / Yes	Yes / Yes

Figure 2.1: Competitor Properties

In Figure 2.1, a 'yes' means that the algorithm attempts to maximize accrued utility under the circumstances defined and a 'no' means that the algorithm does not do so. As can be seen, the collaborative scheduling algorithms are the only ones that have a 'yes' under all scenarios.

Note that while Figure 2.1 specifies whether or not a particular algorithm attempts to maximize accrued utilities under certain scenarios, it does not specify the extent to which each algorithm is able to succeed in this goal – or whether the overhead involved in handling a particular scenario is justifiable. This issue in addressed in greater detail in our experimental section. Note also, that while RMS does not attempt to maximum accrued utility – it is not optimal during underloads and does not consider utility in its scheduling desicions during overload – our experimental evaluation in Chapter 8 shows that it outpeforms EDF scheduling during overloads since it does not suffer from the domino effect previously referred to.

Finally, when choosing a case two scheduling competitor for the non-dependency experiments, we performed a number of experiments to narrow down the field to the algorithms that perform best. The experiments we conduct to do this are described in Chapter 6.

2.3.2 Synchronization

In this section, we briefly summarize the synchronization mechanisms we tackled in this dissertation and provide a roadmap for understanding the synchronization algorithms used in the rest of the dissertation.

2.3.2.1 Locks

In this section, we describe the synchronization protocols that use locks. We also briefly describe the underlying structure that we use to implement locks in ChronOS, the kernel we derived from Linux to conduct our experiments.

As can be seen from Section 2.3.1, the competitor algorithms that we compare against are mostly independent schedulers. Therefore, most of the locking techniques that we compare against are also in that category. In particular, we consider the Priority Inheritance Protocol (PIP) [123] for all the non-TUF independent scheduling protocols we compare against (i.e., case two EDF and case two RMS). We do not consider the Priority Ceiling Protocol (PCP) [123] since it requires that the priority of each task in the system and the identify of the resources that it will lock are known a priori. This, particularly the latter condition of knowing in advance which locks will be held by which threads in the system, is antithetical to the dynamic model we consider in this dissertation.

One consequence of our choice of PIP rather than PCP for managing locks in the case two schedulers is that it is now possible for deadlocks to occur. PCP prevents this by preventing

nested chains of dependencies from occurring since the task that acquires the lock runs at a higher priority than the highest priority task that requires this lock, thus preventing any task that requires that lock from running until it has released the lock it holds [123].

Therefore, we augment the case two schedulers with deadlock detection mechanisms. Specifically, we check for cycles in the resource dependency graph when a system call requesting a lock is made by a particular thread. The implementation details for this functionality are described in Chapter 6. LBESA (LBESA is a best-effort UA scheduling algorithm described in [94]), listed in the third column of Figure 2.1, does not deal with dependencies and so is excluded from the lock-based experiments. DASA [31] uses the lock management scheme described in [31], a variant of which is used in DQBUA, our resource capable scheduling algorithm. The management scheme essentially involves scheduling all the dependencies of a task before it and using the entire dependency chain to compute the heuristic that determines the "importance" of each task.

All locks in this dissertation are built using the futex [39] construct provided by the Linux kernel. We build our own lock and unlock system calls using futexes. Our system calls make the acquiring and release of locks a scheduling event for case two schedulers and a distributed scheduling event for case three schedulers. We also add deadlock detection code to our system calls in order to prevent deadlock from occurring.

2.3.2.2 Non-Lock-based Synchronization

The non-lock-based synchronization mechanism that we use in this dissertation is STM. In Chapters 10 and 11 we describe analysis techniques for bounding the response times of distributed systems programmed using STM. Since, to the best of our knowledge, this is the first time that STM has been considered in a distributed real-time system, the experiments in those chapters compare the results of the analysis to the results obtained form simulations to see how tight the analysis is.

We also consider the effect of different contention management policies on STM in Chapter 12. Several existing contention management policies, listed in that chapter, are compared to the real-time aware contention managers we develop.

Chapter 3

Consensus-Based Collaborative Scheduling

3.1 Introduction

In this chapter, we consider the problem of scheduling threads in the presence of the uncertainties mentioned in Chapter 1, focusing particularly on (arbitrary) node failures and message losses. In the model we consider, communication delay and message losses are stochastically described as in [26]. The proposed algorithm is compared to previous distributable thread scheduling algorithms, HUA [115] and CUA [114].

In this chapter, we target partially synchronous systems, and consider the partially synchronous model in [26], where message delay and message loss are probabilistically described. For such a model, we design a collaborative thread scheduling algorithm called the *Asynchronous Consensus-based Utility accrual scheduling Algorithm* (or ACUA). We show that ACUA satisfies thread time constraints in the presence of crash failures and message losses, is early-deciding (i.e., its decision time is proportional to the actual number of crashes), and has a message and time complexity that compares favorably with other algorithms in its class. Furthermore, we show that ACUA has a better best-effort property — i.e., the affinity for feasibly completing as many high importance threads as possible, irrespective of thread urgency — than past thread scheduling algorithms [114, 115]. We also prove the exception handling properties of ACUA. To the best of our knowledge, this is the first collaborative thread scheduling algorithm designed under a partially synchronous model.

The rest of the chapter is organized as follows: We describe the system models and objectives in Section 3.2. In Section 3.3, we present ACUA. An analytical comparison of its performance to other scheduling algorithms are provided in Section 3.4. We conclude the chapter in Section 3.5.

3.2 Models and Objective

3.2.1 Models

Distributable Threads. Distributable threads execute in local and remote objects by location-independent invocations and returns. The portion of a thread executing an object operation is called a *thread segment*. Thus, a thread can be viewed as being composed of a concatenation of thread segments. A thread can also be viewed as being composed of a sequence of *sections*, where a section is a maximal length sequence of contiguous thread segments on a node.

We assume that execution time estimates of the sections of a thread are known when the thread arrives into the system and are described using TUFs (see our timeliness model). The sequence of remote invocations and returns made by a thread can typically be estimated by analyzing the thread code. The total number of sections of a thread is thus assumed to be known a-priori. The application is thus comprised of a set of threads, denoted $\mathbf{T} = \{T_1, T_2, \ldots\}$. The set of sections of a thread T_i is denoted as $[S_1^i, S_2^i, \ldots, S_k^i]$.

Timeliness Model. We consider the TUF timeliness model described in Chapter 1. Specifically, in our timeliness model, a thread T_i's TUF is denoted as $U_i(t)$. A classical deadline is unit-valued—i.e., $U_i(t) = \{1, 0\}$, since importance is not considered. A variation of downward step TUFs, where $U_i(t) = \{\{m\}, 0\}$, generalize classical deadlines. In this scenario, a thread accrues m utility if it completes before a certain instant of time and zero utility otherwise. The boundary between the point at which a thread accrues m utility and zero utility is the discontinuity point of the TUF. For this variation of downward step TUFs, this discontinuity point is referred to as the deadline of the TUF.

Note that $U_i(t) = \{\{m\}, 0\}$ is a special case of a general downward step TUF, which can be described as $U_i(t) = \{\{m\}, \{n\}\}$ (where n can be any number less than m, including zero and negative numbers). We focus on this special case, $U_i(t) = \{\{m\}, 0\}$, of downward step TUFs in this dissertation, and denote the maximum, constant utility of a TUF $U_i(t)$, simply as U_i. Each TUF has an initial time I_i, which is the earliest time for which the TUF is defined and is usually the arrival time of the thread whose timeliness constraints are defined by that particular TUF, and a termination time, X_i, when the definition of the TUF ceases to exist.

If an executing entity passes its termination time, it should be aborted immediately since its timeliness constraint would no longer exist. The termination time of a downward step TUF is application defined and can be placed at any point at or after the discontinuity point of the TUF. Thus, in general, the termination time of a TUF may not necessarily be equal to its deadline. In this dissertation, we focus on the special case of downward step TUFs where the termination time is equal to the deadline i.e., $U_i(t) = m, \forall t \in [I_i, X_i]$ and $U_i(t) = 0, \forall t \notin [I_i, X_i], \forall i$. In the rest of the dissertation, we use termination time and deadline interchangeably.

When a thread passes its deadline in our model, it is immediately aborted and its abort

handler, which may or may not have a timeliness constraints expressed as a TUF, is scheduled to bring the system to a safe state by releasing any locks held by the thread and undoing changes to the physical environment.

System Model. We consider a set of nodes $\Pi = \{1, \cdots, N\}$, with a logical communication channel between each pair of nodes. We assume that each node is equipped with two processors: a processor that executes thread sections on the node and a scheduling co-processor as in [29]. The dual processor assumption is used to reduce ACUA's scheduling overhead. The dual-processor assumption is also reasonable, given the current proliferation of multi-core/CPU chips. We assume that communication links are unreliable, i.e., messages can be lost with probability p, and communication delay is described by some probability distribution.

Bi-directional logical communication channels are assumed to exist between every pair of node. We assume that these basic communication channels may lose messages with probability p, and communication delay is described by some probability distribution.

On top of this basic communication channel, we consider a reliable communication protocol that delivers a message to its destination in probabilistically bounded time provided that the sender and receiver both remain correct, using the standard technique of sequence numbers and retransmissions. We assume that each node is equipped with two processors (a processor that executes thread sections on the node and a scheduling co-processor as in [29]), have access to GPS clocks that provides each node with a UTC time-source with high accuracy (e.g., [35,61,129]) or have clocks that are synchronized using the Network Time Protocol [101].

We also assume that each node is equipped with $N-1$ QoS failure detectors (FDs) [26] to monitor the status of all other nodes. On each node, i, these $N-1$ FDs output the nodes they suspect to the list $suspect_i$

Exceptions and Abort Model. Each section of a thread has an associated exception handler. We consider a termination model for thread failures including termination time violations and node failures. When such thread failures occur, the section exception handlers are triggered to restore the system to a safe state. The exception handlers may have time constraints expressed as TUFs.

A handler's TUF's initial time is the time of failure of the handler's thread. The handler's TUF's termination time is relative to its initial time. Thus, a handler's absolute and relative termination times are *not* the same. Each handler also has an execution time estimate. This estimate along with the handler's TUF are described by the handler's thread when the thread arrives at a node. A handler is marked as ready for execution when either its latest start time (see Section 3.3.1.3) expires, or it receives an explicit invocation from its successor.

Failure Model. The nodes in our system are subject to crash failures. Up to $f_{max} \leq n-1$ nodes can fail in our system. The actual number of failures in the system is denoted as $f \leq f_{max}$.

3.2.2 Scheduling Objectives

Our primary objective is to design a thread scheduling algorithm that will maximize the total utility accrued by all threads as much as possible. Further, the algorithm must provide assurances on the satisfaction of thread termination times in the presence of (up to f_{max}) crash failures. Moreover, the algorithm must exhibit the best-effort property.

3.3 Algorithm

3.3.1 Rationale

ACUA is a collaborative consensus-based scheduling algorithm. Being a collaborative algorithm, ACUA can construct schedules that result in higher system-wide accrued utility by avoiding locally optimal decisions that can compromise system-wide optimality ("local minimums"). It also allows ACUA to respond to node failures by eliminating threads that are affected by the failures, thus allowing the algorithm to gracefully degrade timeliness in the presence of failures.

In ACUA, when a thread arrives into the system, each node suggests a set of threads for rejection from the system based on local scheduling conditions. The nodes must then agree on a set of threads to reject from the system-wide schedule. We formulate this as a consensus problem with the following properties: (a) If a correct node decides on a reject set $rSet$, then some node proposed $rSet$; (b) Nodes do not decide on different reject sets (*Uniform agreement*); (c) Every correct node eventually decides (i.e. termination).

Since ACUA is a consensus-based algorithm, it can only run on systems on which the distributed consensus problem is solvable. In Section 3.3.1.1, we show that it is possible to design an S class FD, one of the Chandra-Toueg unreliable FDs, on the system model we consider and thus prove that consensus is solvable on that system [23]. Specifically, we show that it is possible to design a FD that provides the semantics of an S class FD *with very high probability* for the *duration of the consensus algorithm.*

Past work [114, 115] had considered the existence of a perfect FD (P class FD), since they considered a synchronous system model. In this work, we use the S class FD (which is weaker than a P class FD) because we consider partially synchronous systems. An S class FD has the following properties: 1) Completeness Property:- There is a time, T_D, after which a failed node is permanently suspected by all nodes; and 2) Accuracy Property:- There is some correct node that is never suspected by all other nodes.

In Section 3.3.1.3, we describe how end-to-end thread TUFs are decomposed in order to obtain the section TUFs necessary for local scheduling on each node.

3.3.1.1 Failure Detection

As mentioned in Section 3.2.1, each node is equipped with an aggregate FD consisting of $N-1$ QoS FDs. Assume that this aggregate FD is polled every δ time units in order to learn the state of the system.

From [26] we know that the probability, P_A, that the result of one of the QoS FDs is accurate when it is queried at random is $E(T_G)/E(T_{MR})$, where $E(T_G)$ is the average time that the FD's output remains correct and $E(T_{MR})$ is the average time between consecutive mistakes. We also know that $E(T_G) = E(T_{MR}) - E(T_M)$, where $E(T_M)$ is the average time it takes for the FD to correct an erroneous failure suspicion. Both $E(T_{MR})$ and $E(T_M)$ are input QoS values chosen when designing the FD, thus we can control P_A by choosing appropriate values for these two parameters.

To show that we can implement an S-class FD using the QoS FD in [26], we need to determine when the consensus algorithm needs the service of the FD. The consensus algorithm used in ACUA is the quorum-based algorithm in [104] which requires the service of the FD in line 5 only.

In the worst case, the algorithm takes N rounds (in each of the first $N-1$ rounds an erroneous suspicion of the round coordinator leads to the next round until round N is reached). Let Δ be the communication delay described by the probability density function $delay(t)$ and the cumulative distribution function $DELAY(t)$, the consensus algorithm will spend either Δ to receive the coordinator's estimate or T_D to detect the coordinator's failure.

In the worst case, the consensus algorithm will query the FD n times, where $n = \lfloor \frac{N \times T_D}{\delta} \rfloor$. We consider each of these queries to be an independent experiment with probability $p = 1 - P_A$ of resulting in an erroneous suspicion. Therefore, the probability that the FD monitoring a single node makes at least one erroneous suspicion during the execution of the algorithm is $P_{FDM} = 1 - bino(0, n, p)$, where $bino(x, n, p)$ is the binomial distribution with parameters n and p. Since there are $N-1$ FDs on each node, the probability that a given node erroneously suspects x nodes is given by $bino(x, N-1, P_{FDM})$ and the probability that a node suspects a majority of the nodes in the system is $\sum_{i=\frac{N-1}{2}+1}^{N-1} bino(i, N-1, P_{FDM})$. Using this analysis, we constructed a FD that suspected a majority of nodes with probability 1.5×10^{-110} for realistic settings. We believe this probability is too low to be of practical concern for the time scales we consider.

Since it is not practically possible for a node to erroneously suspect a majority of other nodes during the execution of the consensus algorithm, the set of nodes not suspected by all nodes in the system have to intersect in at least one node. That node is never suspected by any of the other nodes in the system, thus satisfying the accuracy property of an S class FD. In addition, the T_D detection time of our FD satisfies the completeness property of an S class FD. Therefore, we are able to implement an S class FD with very high probability on our system during the execution of our consensus algorithm.

3.3.1.2 Numerical Example

We now present a numerical example to show that the analysis in Section 3.3.1.1 can result in an S class FD. As in [26], we assume that the message delay is modeled by an exponential distribution with mean and variance of 0.02 seconds. We also assume that the probability of message loss is 0.01 and that there are 100 nodes in the system. In designing the FD, we chose $T_D = 1$ second, $E(T_M) = 0.5$ seconds and $E(T_{MR}) = 1$ month. We also assume that the consensus algorithm queries the FD every 10ms when it needs it. Therefore we can conclude that the consensus algorithm will make $n = 1 * 100/(10 \times 10^{-3}) = 10000$ queries to the FD.

$E(T_G) = E(T_{MR}) - E(T_G) = 2591999.5$. Therefore, $P_A \simeq 0.99999981$, and $p \approx 1.93 \times 10^{-7}$. As described in Section 3.3.1.1, the probability that the FD detector makes at least one mistake is $P_{FDM} = 1 - bino(0,n,p)$, $P_{FDM} \approx 9.64 \times 10^{-4}$. Using this value of P_{FDM}, we compute the probability that a node suspects a majority of other nodes in the system, $\sum_{i=\frac{N-1}{2}+1}^{N-1} bino(i, N-1, P_{FDM})$, to be $\approx 1.5 \times 10^{-110}$. We believe that this probability is too low to be of practical concern for the time scales we are considering. To get a perspective, note that the number of protons in the whole universe is a 24 digit number. Therefore, since it is not practically possible for a node to erroneously suspect a majority of other nodes during the execution of the consensus algorithm, the set of nodes not suspected by all nodes in the system have to intersect in at least one node. That node is never suspected by any of the other nodes in the system, thus satisfying the accuracy property of an S class FD. All calculations are performed using full precision, but values are displayed, in this section, to three significant figures for most outputs for ease of exposition.

3.3.1.3 TUF Decomposition

Thread time constraints are expressed using TUFs. The termination time of each section belonging to a thread needs to be derived from that thread's end-to-end termination time. This derivation should ensure that if all the section termination times are met, then the end-to-end termination time of the thread will also be met.

For the last section of a thread, we derive its termination time as the thread's termination time. The termination time of the other sections is the latest start time of the section's successor minus the communication delay. Thus the section termination times of a thread T_i, with k sections, is:

$$S_j^i.tt = \begin{cases} T_i.tt & j = k \\ S_{j+1}^i.tt - S_{j+1}^i.ex - D & 1 \leq j \leq k-1 \end{cases}$$

where $S_j^i.tt$ denotes section S_j^i's termination time, $T_i.tt$ denotes T_i's termination time, and $S_j^i.ex$ denotes the estimated execution time of section S_j^i. The communication delay, which we denote by D above, is a random variable Δ, as mentioned in Section 3.3.1.1. Therefore, the value of D can only be determined probabilistically. This implies that if each section

meets the termination times computed above, the whole thread will meet its termination time with a certain, high, probability. This is further explored in Section 3.4.

As mentioned in Section 3.2.1, each handler has a TUF that specifies its **relative** termination time, $S_j^h.X$. However, a handler's **absolute** termination time is relative to the time it is released, more specifically, the **absolute** termination time of a handler is equal to the sum of the **relative** termination time of the handler and the failure time t_f (which cannot be known a priori). In order to overcome this problem, we delay the execution of the handler as much as possible which allows us to delay the execution of the exception handlers as much as possible, thus leaving room for more important threads. Therefore, in the equations below we replace t_f with $S_k^i.tt$, the termination time of thread i's last section:

$$S_j^h.tt = \begin{cases} S_k^i.tt + S_j^h.X + T_D + t_a & j = k \\ S_{j+1}^h.tt + S_j^h.X + D & 1 \leq j \leq k-1 \end{cases}$$

where $S_j^h.tt$ denotes section handler S_j^h's termination time, $S_j^h.X$ denotes the relative termination time of section handler S_j^h, t_a is a correction factor corresponding to the execution time of the scheduling algorithm, and T_D is the time needed to detect a failure by our QoS FD. From this decomposition, we compute start times for each handler:

$$S_j^h.st = \{\ S_j^h.tt - S_j^h.ex \quad 1 \leq j \leq k$$

where $S_j^h.ex$ denotes the estimated execution time of section handler S_j^h. Thus, we assure the feasible execution of the exception handlers of failed sections, in order to revert the system to a safe state.

3.3.2 Procedural Description

Algorithm 1 shows the general structure of ACUA. Algorithm 1 is triggered when a thread arrives into the system or when a node fails. When ACUA is triggered, each node constructs a local schedule (line 5). In lines 6-14 each node suggests a set of threads for rejection based on the local schedule it constructs in line 5. In line 15, the nodes send the set of threads they suggest for rejection to all other nodes in the system. Each node then waits for a certain time period to collect the suggestions that other nodes send (lines 15-16). Using these suggestions, each node makes a decision about which set of threads should be rejected from the system (line 18). A consensus protocol is then started in order to reach agreement among the nodes about the set of threads that will be rejected, using the decision each node made in line 18 as input to the consensus protocol (line 19). After reaching agreement, the nodes remove the set of rejected threads from their waiting queue (line 20) and construct a new local schedule containing the remaining threads (line 21).

An important part of ACUA is how it selects a set of threads for rejection locally (lines 7-14). ACUA distinguishes between threads that become unschedulable due to local overloads, and

threads that become unschedulable in order to accommodate a newly arrived thread. This is necessary because a newly arrived thread can only be accepted into the system if *all* its future head nodes accept its sections. Thus, if some nodes reject other threads' sections in order to accommodate the arriving thread, and other nodes reject the sections of the arriving thread, the new thread should not be accepted into the system and the sections rejected to accommodate the new thread's sections on some nodes should be allowed to execute normally.

Algorithm 1: ACUA: ACUA on each node i

```
 1: input: σ_r^i; // σ_r^i: unordered ready queue;
 2: input: σ_p; // σ_p : previous schedule;
 3: output σ_i; // σ_i: schedule;
 4: Initialization: Σ_i = ∅; w_i = ∅;
 5: σ_i = ConstructSchedule(σ_r^i);
 6: if i is head node for newly arrived thread j then
 7:     σ_tmp = ConstructSchedule(σ_r^i − S_j^i);
 8:     if S_j^i ∉ σ_i then
 9:         rSet = 0 ∪ (σ_p − σ_i);
10:     else
11:         tmp = (σ_p − σ_tmp);
12:         rSet = 1 ∪ (σ_p − (σ_i − S_j^i) − tmp) ∪ ⊥ ∪ tmp;
13: else
14:     rSet = ∅ ∪ (σ_p − σ_i);
15: send(rSet_i, i, t) to all;
16: upon receive(rSet_j, j) until 2D do
17:     Σ_i = Σ_i ∪ rSet_j;
18: w_i=DetRejectSet(Σ_i);
19: w_i=UniformConsensus(w_i);
20: UpdateSectionSet(w_i, σ_r^i);
21: σ_i =ConstructSchedule(σ_r^i);
22: σ_p = σ_i;
23: return σ_i;
```

Lines 7-12 perform this function. If the section of the newly arrived thread is not part of the constructed schedule, it cannot be responsible for the elimination of other threads from the system. Thus the difference between the current schedule and the previous schedule is the set of threads that the node proposes for rejection (lines 8-9). On the other hand, if a section of the newly arrived thread is part of the schedule, we need to differentiate between two possible causes for rejecting threads: 1) overload conditions may render some threads unschedulable and 2) the newly arrived thread may render some threads unschedulable.

The former set can be determined by constructing a schedule without considering S_j^i (line 7) and then subtracting that set from the set of previously schedulable threads (line 11). On line 12 we place a separator, $\perp$, between the set of threads rendered unschedulable due to overload and the set of threads rendered unschedulable due to the acceptance of a section of the newly arrived thread. Note that nodes indicate whether they accept, reject, or are not responsible for the sections of a newly arrived thread by prepending 1, 0 and $\emptyset$ to their suggestions respectively.

Using this additional information, the problem mentioned above can be eliminated by only

eliminating threads rendered unschedulable by an arriving thread if ***all*** its future head nodes accept the thread. The details of this functionality is contained in the function *DetRejectSet*. Note that the timeout value on line 16 is a stochastic value, thus even if none of the nodes fail, there is a non-zero probability that some nodes do not receive the suggestions of all other nodes. This is further addressed in Section 3.4.

Algorithm 2: ACUA: DetRejectSet on node i

```
 1: input: Σ_i; // Σ_i: set of suggestions for rejection.
 2: output w_i; // w_i: rejection set output.
 3: accept=true;
 4: w_i = ∅;
 5: for each future head node, j, of newly arrived thread do
 6:     tmp_j=retrieve node j's entry from Σ_i;
 7:     if head(tmp_j)=0 then
 8:         accept=false;
 9: for each node j do
10:     rSet_j=retrieve node j's entry from Σ_i;
11:     rSet_j = rSet_j - first element in rSet_j;
12:     if j is a future head node then
13:         if accept=true then
14:             w_i=w_i ∪ elements before and after ⊥ in rSet_j;
15:         else
16:             w_i=w_i ∪ only elements after ⊥ in rSet_j;
17:     else
18:         w_i=w_i ∪ rSet_j;
19:     if node j is a head node for thread set Γ with a section on node i then
20:         if node i does not receive node j's suggestion then
21:             w_i=w_i ∪ Γ;
22: return w_i;
```

Algorithm 2 describes how nodes determine the set of threads to suggest for rejection from the system. The algorithm first checks whether the newly arrived thread has been accepted into the system by all future head nodes (lines 5-8). Lines 10 and 11 retrieve the suggestion of node j and remove the first element. Lines 12-16 determine which threads to consider for rejection based on the fact that threads rendered unschedulable by the newly arrived thread on some nodes should only be rejected if ***all*** head nodes accept the sections of the newly arrived thread. Line 18 adds the set of threads that non-head nodes suggest for rejection. Finally, lines 19-21 suggests threads for rejection if they have a section hosted on the current node and the current node does not receive any suggestions from one of the previous, current, or future head nodes of the threads. This is done because a node suspects those nodes it does not receive suggestions from to have failed, and thus suggests for elimination the threads that are hosted by them. The uniform consensus algorithm we use is described in [104].

We now turn our attention to the scheduling algorithm that nodes use to construct a local schedule. This algorithm is encapsulated by the function *ConstructSchedule* (see Algorithm 3). The algorithm takes a list of sections, and constructs a total order with each section's global Potential Utility Density (or PUD). The global PUD of a section is the ratio

of the utility of the thread that the section belongs to, to the sum of the remaining execution times of all the thread's sections. The algorithm examines each section in PUD-order, including them in the schedule, and testing for schedule feasibility. If infeasible, the inserted section is rejected, and the process is repeated until all sections are examined. Note that we construct a total order on *global* Potential Utility Density (PUD) in order to attempt to maximize system-wide accrued utility. This can be seen in line 8 of the algorithm, where the execution time of the whole thread, $T_i.ex$, is used instead of the execution time for each individual section, $S_i.ex$, when computing PUD. The algorithm for *UpdateSectionSet* involves a simple removal of the rejected threads from a node's ready queue.

Algorithm 3: ACUA: ConstructSchedule

```
1:  input: σ_r,σ_p,H; output σ;
2:  Initialization: t = t_cur;σ = ∅;HandlerIsMissed = fasle;
3:  for each S_i ∈ σ_p such that S_i ∉ σ_r do
4:      Insert(S_j^h,H,S_j^h.tt);
5:  σ = H;
6:  for each S_i ∈ σ_r do
7:      if S_{j-1}^i.tt + D + S_j^i.ex ≤ S_j^i.tt then
8:          S_i.PUD = min( U_i(t+T_i.ex)/T_i.ex , U_i^h(t+T_i.ex+T_i^h.ex)/(T_i.ex+T_i^h.ex) )
9:      else
10:         S_i.PUD = 0
11: σ_tmp=sortByPUD(σ_r);
12: for each S_i ∈ σ_tmp from head to tail do
13:     if S_i.PUD ≥ 0 then
14:         Insert(S_i,σ,S_i.tt);
15:         Insert(S_i^h,σ,S_i^h.tt);
16:         if Feasible(σ) = false then
17:             Remove(S_i,σ,S_i.tt);
18:             if S_i^h ∉ H then
19:                 Remove(S_i^h,σ,S_i^h.tt);
20:     else
21:         break;
22: σ_p = σ;
23: return σ;
```

3.4 Properties

We compare the best-effort properties of ACUA, CUA [114], and HUA [115]. In HUA (an *independent* node scheduling algorithm), thread sections are scheduled locally at each node they arrive at using their propagated scheduling parameters. The local scheduler is a modified version of DASA [31], which uses the heuristic of favoring tasks with a high utility to execution time ratio, i.e., high PUD, when constructing the schedule. These modifications allow HUA to manage the scheduling of exception handlers in case of thread failure.

In CUA (a *collaborative* scheduling algorithm), when a thread arrives, its sections are sent to all its future head nodes. Each node constructs its schedule locally according to a modified version of DASA. The nodes then cooperate with each other to reach agreement on a system-wide set of threads eligible for execution. Basically, this agreement step involves the elimination of any threads that have any of their sections missing from the global schedule.

We quantify the best-effort property by introducing the concept of DASA Best Effort (or DBE) property:

Definition 1. *Consider a distributed scheduling algorithm $\mathcal{A}$. **DBE** is defined as the property that $\mathcal{A}$ orders its threads in non-increasing order of global PUD while considering them for scheduling and schedules all feasible threads in the system in that order.*

Note that the ***DBE*** property is essential for any UA algorithm that attempts to maximize system-wide accrued utility by favoring tasks that offer the most utility for the least amount of execution time (which is the heuristic used by DASA [31]).

Lemma 1. *HUA, does not have the **DBE** property.*

Proof. The proof is by counterexample. Assume that a system has two nodes, n_1 and n_2, and two threads, T_1 and T_2. Assume that each thread has two sections, one hosted on each of the nodes. Let the sections be S_1^1 and S_1^2 for T_1 and S_2^1 and S_2^2 for T_2. Assume that both threads have end-to-end step-down TUFs, with the utility for T_1 being 5 and the utility of T_2 being 6. Also assume that both threads arrive at n_1 at t_0. Assume that the execution times of S_1^1, S_1^2, S_2^1 and S_2^2 are 2, 3, 3 and 1 time units respectively and that both threads have a relative termination time of 5.

The parameters above ensure that only one of the threads can be scheduled successfully. Therefore, an algorithm that has the ***DBE*** property would choose T_2 for execution since its global PUD, $\frac{6}{4} = 1.5$, is greater than the PUD of T_1, $\frac{5}{5} = 1$. Note that the ***DBE*** property will result in a system-wide accrued utility of 6 in this case.

In contrast, HUA computes the PUD of the *sections* of each thread hosted on each node when constructing its schedule [115]. Since the PUD of S_1^1, $\frac{5}{2} = 2.5$, is greater than the PUD of S_2^1, $\frac{6}{3} = 2$, the scheduler on n_1 will choose S_1^1 for scheduling first. By the time S_1^1 has finished execution, $t_0 + 3$, releasing S_2^1 for execution will mean that it will finish past the global termination time of T_2 $(T_0 + 5)$. Thus, only T_1 will execute with a resulting accrued utility of 5 for the system.

Thus HUA does not have the ***DBE*** property. □

Lemma 2. *CUA does not have the DBE property.*

Proof. CUA does not have the ***DBE*** property became it does not schedule all feasible threads in the system. For example, if two nodes host sections of two threads, T_1 and T_2, during

overloads, one node may schedule the section belonging to T_2 at the expense of that belonging to T_1 and the other may schedule the section belonging to T_1 at the expense of that belonging to T_2. Since CUA excludes threads from the system if they are missing any of their sections and both of the above threads have one of their sections missing, both threads will be excluded from the system. This is unnecessary since excluding one thread will render the other schedulable, thus the algorithm does not schedule all feasible threads and therefore does not have the ***DBE*** property. □

Theorem 3. *ACUA has the **DBE** property for threads that can be delayed $O(f\Delta+nk)$ (see Lemma 5) and are still schedulable.*

Proof. ACUA overcomes the issue mentioned in Theorem 1 because it uses the PUD of the entire thread when constructing local schedules on each node. Thus sections that are excluded are those with the least *system-wide* PUD. In other words, the threads in ACUA are considered in non-increasing order of global PUD for scheduling. In addition, ACUA overcomes the issue mentioned in Theorem 2 by preventing an arriving thread from eliminating other threads if at least one of the nodes that will be hosting a future head of the arriving thread does not accept that section for scheduling. The details of this procedure are explained in Algorithms 1 and 2. This allows ACUA to schedule all feasible threads. Thus all feasible threads that can tolerant the scheduling overhead of ACUA and still remain feasible will be scheduled in non-increasing order of global PUD. The theorem follows from Definition 1. □

Theorem 4. *ACUA can tolerate up to $f_{max} = n-1$ faulty processors.*

Proof. This follows directly from the fault tolerant property of the S class based consensus algorithm in [104] which we use in our work. □

Lemma 5. *ACUA has time complexity $O(f\Delta+nk)$.*

Proof. Lines 5 and 7 in Algorithm 1 have complexity $O(k^2)$ where k is the maximum number of sections in the ready queue of system nodes. Lines 8-15 have constant complexity, lines 16-17 have complexity 2Δ, line 18 has complexity $O(nk)$, line 19 has complexity $O((f+1)\Delta)$, line 20 has complexity $O(k)$ and line 21 has complexity $O(k^2)$. Therefore, the algorithm has actual complexity of $3k^2+2\Delta+nk+(f+1)\Delta+k$, which is asymptotically $O(f\Delta+nk)$ if we consider k a constant. □

This time complexity compares favorably with the time complexity of CUA, which is $O(D+df+nk)$ [114], asymptotically. However, the value of the time complexity of CUA is lower than that of ACUA since it makes the additional assumption of the existence of a fast FD [4]. In addition, Δ is a random variable, thus the timing guarantee for ACUA is stochastic in nature.

Lemma 6. *ACUA has message complexity $O(fn^2)$.*

Proof. Lines 16-17 in Algorithm 1 have message complexity n (one for each suggested rejection set sent by a node). Line 19 has message complexity $n^2(f+1)$ since each round has a message cost of n^2. The algorithm is early deciding so it will take $f+1$ rounds [104]. Therefore the actual message cost of the algorithm is $n+n^2(f+1)$, which is asymptotically $O(fn^2)$. □

Lemma 7. *The message size in ACUA is smaller than that in CUA for well behaved systems.*

Proof. The input to the consensus algorithm in ACUA is the set of rejected threads while the input to the consensus algorithm in CUA is the set of schedulable threads. Since the set of rejected threads should be smaller than the set of accepted threads in well behaved systems, we claim that the message size in ACUA is smaller than that in CUA. □

Lemma 8. *If each section of a thread meets its derived termination time (see Section 3.3.1.3), then under ACUA, the entire thread meets its termination time with high, computable probability,* p_{suc}.

Proof. Since the termination times derived for sections are a function of communication delay, and this communication delay is a random variable with CDF $DELAY(t)$, the fact that all sections meet their termination times implies that the whole thread will meet its global termination time only if none of the communication delays used in the derivation in Section 3.3.1.3 are violated during runtime.

Let D be the communication delay used in the derivation of section termination times. The probability that D is violated at runtime is $p = 1 - DELAY(D)$. For a thread with k sections, the probability that none of the section to section transitions incur a communication delay above D is $p_{suc} = bino(0,k,p)$. Thus, the probability that the thread meets its termination time is also $p_{suc} = bino(0,k,p)$. □

For the next set of theorems, we define a new property, which we shall refer to as the Handler Underload State, as follows:-

Definition 2 (Handler Underload State)**.** *A node with sections* $S_{1\ldots n}$ *and handlers* $H_{1\ldots n}$*, is considered in the Handler Underload State if* $\sum S_i.X/S_i.p <= 1 - \sum H_i.X/S_i.p$ *where* $S_i.X$ *and* $H_i.X$ *are the execution times of section* i *and its handler respectively, and* $S_i.p$ *is the deadline of the section. In other words, a node is in the Handler Underload State if the offered load of the sections it hosts is less than one minus the offered node of the failure handler of each of the sections it hosts.*

Note that for zero overhead failure handlers, or for non-zero but negligible handler overheads, Definition 2 is equivalent to the normal definition of underload.

Lemma 9. *If all nodes are in the Handler Underload State (Definition 2) and no nodes fail, then no threads will be suggested for rejection by ACUA with high, computable, probability* p_{norej}.

Proof. Since the nodes are all in the Handler Underload State (Definition 2) and no nodes fail, Algorithm 3 ensures that all sections will be accepted. Thus, the only source of thread rejection is if a node does not receive a suggestion from other nodes during the timeout value, D, (see Algorithm 1 in Section 3.3.2). This can occur due to one of two reasons; 1) the broadcast message (line 15), that indicates the start of the consensus algorithm, may not reach some nodes 2) the broadcast message reaches all nodes, but these nodes do not send their suggestions to other nodes in the system during the timeout value assigned to them.

The probability that a node does not receive a message within the timeout value from one of the other nodes is $p = 1 - DELAY(D)$. We consider the broadcast message to be a series of unicasts to all other nodes in the system. Therefore, the probability that the broadcast start of consensus message reaches all nodes is $P_{tmp} = bino(0,N,p)$ where $bino(x,n,p)$ is the binomial distribution with parameters n and p. If this message is received, a node waits for messages from all other nodes. The probability that none of these messages arrive after the timeout is $tmp = bino(0,N,p)$. Since there are N nodes, the probability that none of these nodes miss a message is $bino(N,N,tmp)$. Therefore the probability that no threads will be rejected is the product of the probability that the broadcast message reaches all nodes, and the probability that all nodes receive suggestions from all other nodes in response to this start of consensus message i.e. $p_{norej} = bino(N,N,tmp) \times P_{tmp}$. □

Theorem 10. *If all nodes are in the Handler Underload State (Definition 2), no nodes fail (i.e. $f = 0$), and threads can be delayed $O(f\Delta + nk)$ time units once and still be schedulable, ACUA meets all the thread termination times yielding optimal total utility with high, computable, probability, P_{alg}.*

Proof. By Lemma 9, no threads will be considered for rejection from a fault free system in the Handler Underload State (Definition 2) with probability p_{norej}. This means that all sections will be scheduled to meet their derived termination times by Algorithm 3. Thus, by Lemma 8, each thread, j, will meet its termination time with probability p^j_{suc}. Therefore, for a system with X threads, the probability that all threads meet their termination time is $P_{tmp} = \prod_{j=1}^{X} p^j_{suc}$. Given that the probability that all threads will be accepted is p_{norej}, $P_{alg} = P_{tmp} \times p_{norej}$.

ACUA takes $O(f\Delta + nk)$ time units to determine a newly arrived thread's schedulability. If this delay causes any of the thread's sections to miss their termination times, the thread will not be schedulable. We require that a thread suffer this delay *once* because we assume that there is a scheduling co-processor on each node. Thus, the delay will only be incurred by the newly arrived thread while other threads continue to execute uninterrupted on the other processor. □

Theorem 11. *ACUA is an early deciding algorithm that achieves consensus on the system-wide execution eligible thread set in a partially synchronous system with virtually certain probability.*

Proof. Since the consensus algorithm in [104], on which we base our algorithm, is early deciding so is our algorithm. In addition, we show in Section 3.3.1.1 that we can provide an S class FD with very high probability during the execution of our algorithm (with probability of error 1.50×10^{-110}), therefore the S class FD based consensus algorithm in [104] executes on our system with virtually certain probability. Since the input to the consensus algorithm is the set of threads to reject from the system, at its completion all nodes will agree on the set of threads to reject from their schedules and hence on the system-wide set of execution eligible threads. □

Theorem 12. *If $n-f$ nodes do not crash, are in the Handler Underload State (Definition 2), and all incoming threads can be delayed $O(f\Delta + nk)$ and still be schedulable, ACUA meets the execution time of all threads in its eligible execution thread set, Γ, with high computable probability, P_{alg}.*

Proof. By Theorem 11, ACUA achieves system-wide consensus on the set of schedulable threads. By Lemma 9, the probability that none of the threads hosted by the surviving nodes are rejected is, $p_{norej} = bino(N-f, N-f, tmp) \times tmp$ where $tmp = bino(0, N-f, p)$ and $p = 1 - DELAY(D)$. Thus all sections belonging to those threads will be scheduled to meet their derived termination times. By Lemma 8, this implies that each of these threads, j, will meet its termination time with probability p^j_{suc}. Therefore, for a system with an eligible thread set, Γ, the probability that all threads meet their termination times is $P_{tmp} = \prod_{j \in \Gamma} p^j_{suc}$. Thus, the probability that all the remaining threads are accepted is $P_{alg} = P_{tmp} \times p_{norej}$. □

Definition 3 (Section Failure)**.** *A section, S^i_j, is said to have failed when one or more of the previous head nodes of S^i_j's thread (other than S^i_j's node) has crashed.*

Lemma 13. *If a node hosting a section, S^i_j, of thread T_i fails at time t_f, every correct node will include handlers for thread T_i in H by time $t_f + T_D + t_a$, where t_a is an implementation-specific computed execution bound for ACUA calculated per the analysis in Theorem 5.*

Proof. Since the QoS FD we use detects a failed node in T_D time units [26], all nodes detect the failure of the failed node at time $t_f + T_D$. As a result, ACUA is triggered and excludes T_i from the system because nodes will not receive any suggestions from node j (see lines 19-21 of Algorithm 2). Consequently, Algorithm 3 will include the section handlers for this thread in H (see lines 3-4 of Algorithm 3). Execution of ACUA completes in time t_a and thus all handlers will be included in H by time $t_f + T_D + t_a$. □

Again it should be noted that t_a is a stochastic value and therefore the timeliness property above is probabilistic in nature. The probability that a particular value of t_a can be met is easy to obtain by considering the CDF of Δ when conducting the analysis in Theorem 5.

Lemma 14. *If a section S_i, where $i \neq k$, fails at time t_f (per Definition 3) and section S_{i+1} is correct, then under ACUA, its handler S^h_i will be released no earlier than S^h_{i+1}'s completion and no later than $S^h_{i+1}.tt + D + S^h_i.X - S^h_i.ex$.*

Proof. For $i \neq k$, a section's exception handler can be released due to one of two events; 1) its start time expires; or 2) an explicit invocation is made by the handler's successor.

For the first case, we know from the analysis in Section 3.3.1.3 that the start time of S_i^h is $S_{i+1}^h.tt + S_j^h.X + D - S_j^h.ex$. Thus, by definition, it satisfies the upper bound in the theorem. Also, since $S_j^h.X \geq S_j^h.ex$ (otherwise the handler would not be schedulable), $S_{i+1}^h.tt + S_j^h.X + D - S_j^h.ex > S_{i+1}^h.tt$, and this satisfies the lower bound of the theorem.

For the second case, an explicit message has arrived indicating the completion of S_{i+1}^h. Since the message was sent, this means that $S_{i+1}^h.tt$ has already passed, thus satisfying the theorem lower bound. Further, the message should have arrived D time units after S_{i+1}^h finishes execution (i.e., at $S_{i+1}^h.tt + D$), since $S_{i+1}^h.tt + D \leq S_{i+1}^h.tt + D + S_i^h.X - S_i^h.ex$ (as $S_i^h.X \geq S_i^h.ex$), thus satisfying the upper bound. □

Lemma 15. *If a section S_i fails (per Definition 3), then under ACUA, its handler S_i^h will complete no later than $S_i^h.tt$ (barring S_i^h's failure).*

Proof. If one or more of the previous head nodes of S_i's thread has crashed, it implies that S_i's thread was present in a system-wide schedulable set previously constructed. This means that S_i and its handler were previously determined to be feasible before $S_i.tt$ and $S_i^h.tt$, respectively (lines 13-19, Algorithm 3). When some previous head node of S_i's thread fails, ACUA will be triggered and will remove S_i from the pending queue. In addition, Algorithm 3 will include S_i^h in H and construct a feasible schedule containing S_i^h (lines 3-21). Since the schedule is feasible and S_i^h is inserted to meet $S_i^h.tt$ (line 4), then S_i^h will complete by time $S_i^h.tt$ □

Theorem 16. *When a thread fails, the thread's handlers will be executed in LIFO (last-in first-out) order. Furthermore, all (correct) handlers will complete in bounded time. For a thread with k sections, handler termination times $S_i^h.X$, which fails at time t_f, and (distributed) scheduler latency t_a, this bound is $T_i.X + \sum_i S_i^h.X + kD + T_D + t_a$.*

Proof. The LIFO property follows from Lemma 14. Since it is guaranteed that each handler, S_i^h, cannot begin before the termination time of handler S_{i+1}^h (the lower bound in Lemma 14), thus we guarantee LIFO execution of the handlers. Lemma 15 shows that all correct handlers complete in bounded time. Finally, if a thread fails at time t_f, all nodes will include handlers for this thread in their schedule by time $t_f + T_D + t_a$ (Lemma 13) and ACUA guarantees that all these sections will complete before their termination times (Lemma 15). Due to the LIFO nature of handler executions, the last handler to execute is the first exception handler, S_1^h. The termination time of this handler (from the equations in Section 3.3.1.3) is $T_i.X + \sum_i S_i^h.X + kD + T_D + t_a$. The theorem follows. □

3.5 Conclusions

We presented a best-effort utility accrual scheduling algorithm, ACUA, for scheduling distributable real-time threads in partially synchronous systems. We compared ACUA in terms of its best-effort property, and message and time complexity to two previous thread scheduling algorithms including CUA and HUA. We showed that ACUA has a better best-effort property during overloads than HUA and CUA, and has message and time complexities that are comparable to CUA (which is in its class). We also showed the exception handling properties of ACUA.

Both CUA and HUA have optimal best-effort properties during underloads. The two algorithms, as well as ACUA, achieve optimal total utility during underloads as defined in Definition 2 (for CUA and ACUA this property comes with the caveat that the threads should be able to tolerate the algorithm overheads). However, during overloads, Lemmas 1, 2 and Theorem 3 show that ACUA has better best-effort semantics. In addition, if we choose to implement ACUA on a totally synchronous system, it will have exactly the same message and time complexity as CUA (since we can now use the fast consensus algorithm in [4] instead of the quorum-based algorithm in [104]) and yet possess better best-effort properties than CUA during overloads. In addition, all ACUA's stochastic properties will become deterministic.

Chapter 4

Quorum-Based Collaborative Scheduling

4.1 Introduction

In this chapter, we consider the problem of scheduling threads in the presence of the uncertainties mentioned in Chapter 1, focusing particularly on (arbitrary) node failures and message losses. In the model we consider, communication delay and message losses are stochastically described as in [26]. The proposed algorithm is compared to previous distributable thread scheduling algorithms, HUA [115], CUA [114], and ACUA [48].

We present a collaborative scheduling algorithm called the *Quorum-Based Utility Accrual scheduling* (or QBUA) algorithm. The algorithm considers the partially synchronous model in [26], and uses a Quorum set of nodes for majority agreement on constructing system-wide thread schedules. We show that QBUA satisfies thread time constraints in the presence of node crash failures and message losses, has efficient message and time complexities that compare favorably with other algorithms in its class, and superior timeliness than past algorithms including CUA and HUA. We also show that the algorithm's lower overhead, in the presence of failure, enables it to allow more threads to benefit from its superior timeliness, than that allowed by past algorithms.

The rest of the chapter is organized as follows: We describe the system models and objectives in Section 4.2. In Section 4.3, we present QBUA. Its analytical properties are described in Section 4.4 respectively. We conclude the chapter in Section 4.5.

4.2 Models and Objective

4.2.1 Models

Distributable Threads. As in Chapter 3, we use the distributable thread abstraction as our programming model.

Timeliness Model. We employ the TUF timeliness model described in Chapter 3.

System Model. We consider a networked embedded system to consist of a set of client nodes $\Pi^c = \{1,2,\cdots,N\}$ and a set of server nodes $\Pi = \{1,2,\cdots,n\}$ (*server* and *client* are logical designations given to nodes to describe the algorithm's behavior). Bi-directional logical communication channels are assumed to exist between every client-server and client-client pair. We also assume that these basic communication channels may lose messages with probability p, and communication delay is described by some probability distribution.

On top of this basic communication channel, we consider a reliable communication protocol that delivers a message to its destination in probabilistically bounded time provided that the sender and receiver both remain correct, using the standard technique of sequence numbers and retransmissions. We assume that each node is equipped with two processors (a processor that executes thread sections on the node and a scheduling co-processor as in [29]), and have access to GPS clocks that provides each node with a UTC time-source with high accuracy (e.g., [35,61,129]) or have clocks that are synchronized using the Network Time Protocol [101].

We also assume that each node is equipped with $N-1$ QoS failure detectors (FDs) [26] to monitor the status of all other nodes. On each node, i, these $N-1$ FDs output the nodes they suspect to the list $suspect_i$

Exceptions and Abort Model. We employ the same exception and abort model described in Chapter 3.

Failure Model. Nodes are subject to crash failures. When a process crashes, it loses its state memory — i.e., there is no persistent storage. If a crashed client node recovers at a later time, we consider it a new node since it has already lost all of its former execution context. A client node is *correct* if it does not crash; it is *faulty* if it is not correct. In the case of a server crash, it may either recover or be replaced by a new server assuming the same server name (using DNS or DHT — e.g, [41] — technology). We model both cases as server recovery.

Since crashes are associated with memory loss, recovered servers start from their initial state. A server is *correct* if it does not fail; it is *faulty* if it is not correct. QBUA tolerates up to $N-1$ client failures and up to $f^s_{max} \leq n/3$ server failures. The actual number of failures is denoted as $f^s \leq f^s_{max}$ for servers and $f \leq f_{max}$ where $f_{max} \leq N-1$ for clients.

4.2.2 Scheduling Objectives

Our primary objective is to design a thread scheduling algorithm that will maximize the total utility accrued by all threads as much as possible. Further, the algorithm must provide assurances on the satisfaction of thread termination times in the presence of (up to f_{max}) crash failures. Moreover, the algorithm must exhibit the best-effort property.

4.3 Algorithm

4.3.1 Rationale

QBUA is a collaborative scheduling algorithm. Thus, QBUA can construct schedules that result in higher system-wide accrued utility by avoiding locally optimal decisions that can compromise system-wide optimality ("local minimums"). It also allows QBUA to respond to node failures by eliminating threads that are affected by the failures, thus allowing the algorithm to gracefully degrade timeliness in the presence of failures. There are two types of scheduling events that are handled by QBUA; a) local scheduling events and b) distributed scheduling events.

Local scheduling events are handled locally on a node without consulting other nodes. Examples of local scheduling events are section completion and section handler expiry events. For a full list of local scheduling events, please see Algorithm 9. Distributed scheduling events need the participation of all nodes in the system to handle them. In this work, only two distributed scheduling events exit; a) the arrival of a new thread into the system and b) failure of a node.

A node that detects a distributed scheduling event sends a START message to all other nodes requesting their scheduling information so that it can compute a System Wide Executable Thread Set (or SWETS). Nodes that receive this message, send their scheduling information to the requesting node and wait for schedule updates (which are sent to them when the requesting node computes a new system-wide schedule). This may lead to contention if several different nodes detect the same distributed scheduling event concurrently.

For example, when a node fails, many nodes may detect the failure concurrently. It is superfluous for all these nodes to start an instance of QBUA. In addition, events that occur in quick succession may trigger several instances of QBUA when only one instance can handle all of those events. To prevent this, we use a quorum system to arbitrate among the nodes wishing to run QBUA. In order to perform this arbitration, the quorum system examines the time-stamp of incoming events. If an instance of QBUA was granted permission to run *later* than an incoming event, there is no need to run another instance of QBUA since information about the incoming event will be available to the version of QBUA already running (i.e., the event will be handled by that instance of QBUA).

In order to perform this functionality, QBUA requires timeliness information for each of the sections it schedules. As mentioned in Section 4.2.1, end-to-end thread timeliness requirements are described using TUFs.

4.3.2 Procedural Description

We use the same method of TUF decomposition as mentioned in Chapter 3. As mentioned above, whenever a distributed scheduling event occurs, a node attempts to acquire permission from the quorum system to run a version of QBUA. After the quorum system has arbitrated among the nodes contending to execute QBUA, the node that acquires the "lock" executes Algorithm 4. In Algorithm 4, the node first broadcasts a start of algorithm message (line 1) and then waits $2T$ time units[1] for all nodes in the system to respond by sending their local scheduling information (line 2). After collecting this information, the node computes SWETS (line 3) using Algorithm 8. After computing SWETS, the node contacts affected nodes (i.e. nodes that will have sections added or removed from their schedule as a result of the scheduling event).

Algorithm 4: QBUA: Compute SWETS

```
1: Broadcast start of algorithm message, START;
2: Wait 2T collecting replies from other nodes;
3: Construct SWETS using information collected;
4: Multicast change of schedule to affected nodes;
5: return;
```

The algorithm that client nodes run when attempting to acquire a "lock" on running a version of QBUA, Algorithm 5, is loosely based on Chen's solution for FTME [25]. Upon the arrival of a distributed scheduling event, a node tries to acquire a "lock" on running QBUA (the try_1 part of the algorithm that starts on line 3). The first thing that the node does (lines 4-5) is check if it is currently running an instance of QBUA that is in its information collection phase (line 2 in Algorithm 4). If so, the new event that has occurred can simply be added to the information being collected by this version of QBUA. However, if no current instance of QBUA is being hosted by the node, or if the instance of QBUA being hosted has passed its information collection phase, then the event may have to spawn a new instance of QBUA (this starts at line 6 in the algorithm).

The first thing that Algorithm 5 does in this case is send a time-stamped request to the set of server nodes, $\mathbf{\Pi}$, in the system (lines 8-10). The time-stamp is used to inform the quorum nodes of the time at which the event was detected by the current node. Beginning at line 3, Algorithm 5 collects replies from the servers. Once a sufficient number of replies have arrived (line 14), Algorithm 5 checks whether its request has been accepted by a sufficient ($\lceil \frac{2n}{3} \rceil$ see Section 4.4) number of server nodes. If so, the node computes SWETS (lines 15-16).

[1] T is communication delay derived from the random variable describing the communication delay in the system.

On the other hand, if an insufficient number of server nodes support the request, two possibilities exist. The first possibility is that another node has been granted permission to run an instance of QBUA to handle this event. In this case, the current node does not need to perform any additional action and so releases the "lock" it has acquired on some servers (lines 17-21).

The second possibility is that the result of the contention to run QBUA at the servers was inconclusive due to differences in communication delay. For example, assume that we have 5 servers and three clients wishing to run QBUA and all three clients send their request to the servers at the same time, also assume different communication delay between each server and client. Due to these communication differences, the messages of the clients may arrive in such a pattern so that two servers support client 1, another 2 servers support client 2 and the last server supports client 3. This means that no client's request is supported by a sufficient — i.e., $\frac{2n}{3}$ — number of server nodes. In this case, the client node sends a YIELD message to servers that support it and an INQUIRE message to nodes that do not support it (line 22-28) and waits for more responses from the server nodes to resolve this conflict. Lines 30-35 release the "lock" on servers after the client node has computed SWETS, lines 36-38 are used to handle the periodic cleanup messages sent by the servers and lines 39-41 respond to the START of algorithm message (line 1, Algorithm 4).

Algorithm 6 is run by the servers, the function of this algorithm is to arbitrate among the nodes contending to run QBUA so as to minimize the number of concurrent executions of the algorithm. Since there may be more than one instance of QBUA running at any given time, the server nodes keep track of these instances using three arrays. The first array, $c_{owner}[]$, keeps track of which nodes are running instances of QBUA, the second, $t_{owner}[]$, stores the time at which a node in $c_{owner}[]$ sends a request to the servers (i.e., the time at which that node detects a certain scheduling event), and $t_{grant}[]$ keeps track of the time at which server nodes grant permission to client nodes to execute QBUA. Also, a waiting queue for each running instance of QBUA is kept in $R_{wait}[]$.

When a server receives a message from a client node, it first checks to see if this is a stale message (which may happen due to out of order delivery). A message from a client node, c_1, that has a time-stamp older than the last message received from c_1 has been delivered out of order and is ignored (line 7-8). Starting at line 9, the algorithm begins to examine the message it has received. If it is a REQUEST message, the server checks if the time-stamp of the event triggering the message is *less* than the time at which a client node was *granted* permission to run an instance of QBUA. If such an instance exists, a new instance of QBUA is not needed since the event will be handled by that previous instance of QBUA. Algorithm 6, inserts the incoming request into a waiting queue associated with that instance of QBUA and sends a message to the client (lines 10-13).

However, if no current instance of QBUA can handle the event, a client's request to start an instance of QBUA is granted (lines 14-18). If a client node sends a YIELD message, the server revokes the grant it issued to that client and selects another client from the waiting queue

Algorithm 5: QBUA: QBUA on client node i

```
1:  timestamp; // time stamp variable initially set to nil
2:  upon thread arrival or detection of a node failure:
3:    try_1:
4:      if a current version of QBUA is waiting for information from other nodes then
5:          Include information about event when computing SWETS;
6:      else
7:          timestamp ← GetTimeStamp;
8:          for all r_j ∈ Π do
9:              resp[j] ← (nil,nil);
10:             send (REQUEST, timestamp) to r_j;
11:         repeat
12:             wait until [received (RESPONSE, owner, t) from some r_j];
13:             if (c_1 ≠ owner or timestamp = t) then resp[j] ← (owner, t);
14:             if among resp[], at least m of them are not (nil, nil) then
15:                 if at least m elements in resp[] are (c_1, t) then
16:                     return Compute SWETS;
17:                 else if at least m elements in resp[] agree about a certain node then
18:                     for all r_k ∈ Π such that resp[k] ≠ (nil, nil) do
19:                         if resp[k].owner = c_1 then
20:                             send (RELEASE,timestamp) to r_k;
21:                     Skip rest of algorithm; //Event is already being handled
22:                 else
23:                     for all r_k ∈ Π such that resp[k] ≠ (nil, nil) do
24:                         if resp[k].owner = c_1 then
25:                             send (YIELD,timestamp) to r_k;
26:                         else
27:                             send (INQUIRE,timestamp) to r_k;
28:                         resp[k] ← (nil,nil);
29:         until forever ;
30:       exit_1:
31:         oldtimestamp ← timestamp;
32:         timestamp ← GetTimeStamp;
33:         for all r_k ∈ Π do
34:             send (RELEASE, oldtimestamp) to r_j;
35:             return;
36: upon receive (CHECK, t) from r_j
37:   if for all instances of QBUA running on this node, timestamp ≠ t then
38:       send (RELEASE, t) to r_j;
39: upon receive (START) from some client node
40:   Update RE_j^i for all sections;
41:   send σ_j and RE_j^i's to requesting node;
```

for that event (lines 21-31). This part of the algorithm can only be triggered if the result of the first round of contention to run QBUA is inconclusive (as discussed when describing Algorithm 5). Recall that this inconclusive contention is caused by different communication delays that allow different requests to arrive at different severs in different orders. However, all client requests for a particular instance of QBUA are queued in $R_{wait}[]$, therefore, when a client sends a YIELD message, servers are able to choose the highest priority request (which we define as the request with the earliest time-stamp and use node id as a tie breaker). Thus, we guarantee that this contention will be resolved in the second round of the algorithm. Lines 32-34 show servers' response to INQUIRE messages and lines 35-39 show the clean up procedures to remove stale messages. As can be seen on line 36, when a node that is currently running an instance of QBUA fails, HandleFailure($c_{owner}[i]$,$c_{owner}[]$,$t_{owner}[]$,$t_{grant}[]$,$R_{wait}[]$) is called to handle this failure. Algorithm 7 shows the details of this function. If the waiting queue corresponding to this instance of QBUA, $R_{wait}[i]$, is empty, then there are no other nodes that have detected the event that triggered QBUA on $c_{owner}[i]$ and so the system is cleared of this instance of QBUA (lines 1-3). Otherwise, there are other nodes that have detected the event that triggered QBUA on $c_{owner}[i]$, or another concurrent event, and therefore the failure of $c_{owner}[i]$ results in selecting another node from the waiting queue $R_{wait}[i]$ to run QBUA to handle this event (lines 4-9).

Algorithm 8 is used by a client node to compute SWETS once it has received information from all other nodes in the system (line 2 in Algorithm 4). It performs two basic functions, first, it computes a system wide order on threads by computing their global Potential Utility Density (PUD). It then attempts to insert the remaining sections of each thread, in non-increasing order of global PUD, into the scheduling queues of all nodes in the system. After the insertion of each thread, the schedule is checked for feasibility. If it is not feasible, the thread is removed from SWETS (after scheduling the appropriate exception handler if necessary).

First we need to define the global PUD of a thread. Assume that a thread, T_i, has k sections denoted $\{S_1^i, S_2^i, \cdots, S_k^i\}$. We define the global remaining execution time, GE_i, of the thread to be the sum of the remaining execution times of each of the thread's sections. Let $\{RE_1^i, RE_2^i, \cdots, RE_k^i\}$ be the set of remaining execution times of T_i's sections, then $GE_i = \sum_{j=1}^{k} RE_j^i$. Assuming that we are using step-down TUFs, and T_i's TUF is $U_i(t)$, then its global PUD can be computed as $T_i.PUD = U_i(t_{curr} + GE_i)/GE_i$, where U is the utility of the thread and t_{curr} is the current time. Using global PUD, we can establish a system wide order on the threads in non-increasing order of "return on investment". Thus allowing us to consider them for scheduling in an order that attempts to maximize accrued utility [31].

In Algorithm 8, each node, j, sends the node running QBUA its current local schedule σ_j^p. Using these schedules, the node can determine the set of threads, Γ, that are currently in the system. Both these variables are inputs to the scheduling algorithm (lines 1 and 2 in Algorithm 8). In lines 3-6, the algorithm computes the global PUD of each thread in Γ.

Before we schedule the threads, we need to ensure that the exception handlers of any thread

Algorithm 6: QBUA: QBUA on server node i

```
1:  c_owner[]; Array of nodes holding lock to run QBUA
2:  t_owner[]; t_owner[i] contains time-stamp of event that triggered QBUA for node in c_owner[i]
3:  t_grant[]; t_grant[i] contains time at which node in c_owner[i] was granted lock to run QBUA
4:  R_wait[]; R_wait[i] is waiting queue for instance of QBUA being run by c_owner[i];
5:  upon receive (tag, t)
6:     CurrentTime ← GetTimeStamp;
7:     if (c1, t') appears in (c_owner[],t_owner[]) or R_wait[] then
8:        if t < t' then Skip rest of algo; //This is an old message
9:     if tag = REQUEST then
10:       if ∃ t_grant ∈ t_grant[] such that t ≤ t_grant then
11:          send (RESPONSE, c, t_grant) to c1; //where c ← c_owner[i], such that t_grant[i] = t_grant;
12:          Enqueue (c1, t) in R_wait[i], such that t_grant[i] = t_grant;
13:          Skip rest of algorithm;
14:       else
15:          AddElement(c_owner[],c1);
16:          AddElement(t_owner[],t);
17:          AddElement(t_grant[],CurrentTime);
18:          send (RESPONSE, c1, t) to c1;
19:    else if tag = RELEASE then
20:       Delete entry corresponding to c1, t from c_owner[], t_owner[], t_grant[], and R_wait[];
21:    else if tag = YIELD then
22:       if (c1, t) ∈ (c_owner[], t_owner[]) then
23:          For i, such that (c1, t) = (c_owner[i], t_owner[i])
24:          Enqueue (c1, t) in R_wait[i];
25:          (c_wait,t_wait) ← top of R_wait[i];
26:          c_owner[i] ← c_wait; t_owner[i] ← t_wait;
27:          t_grant[i] ← CurrentTime;
28:          send (RESPONSE, c_wait, t_wait) to c_wait;
29:       if c1 ∉ c_owner[] then
30:          (c,t_p) ← (c_owner[i],t_owner[i]), for min i such that t ≤ t_grant[i];
31:          send (RESPONSE, c, t_p) to c1;
32:    else if tag = INQUIRE then
33:       (c,t_p) ← (c_owner[i],t_owner[i]), for min i such that t ≤ t_grant[i];
34:       send (RESPONSE, c, t_p) to c1;
35: upon suspect that c_owner[i] has failed:
36:    HandleFailure(c_owner[i],c_owner[],t_owner[],t_grant[],R_wait[]);
37: periodically:
38:    ∀ c_owner ∈ c_owner[]:
39:       send (CHECK, t_owner) to c_owner; //NB. t_owner is the entry in t_owner[] that corresponds to c_owner.
```

Algorithm 7: QBUA: HandleFailure($c_{owner}[i]$,$c_{owner}[]$,$t_{owner}[]$,$t_{grant}[]$,$R_{wait}[]$)

```
1: if R_wait[i] is empty then
2:    remove c_owner[i]'s entry from c_owner[], t_owner[], t_grant[];
3:    Delete R_wait[i];
4: else
5:    CurrentTime ← GetTimeStamp;
6:    (c_wait,t_wait) ← top of R_wait[i];
7:    c_owner[i] ← c_wait; t_owner[i] ← t_wait;
8:    t_grant[i] ← CurrentTime;
9:    send (RESPONSE, c_wait, t_wait) to c_wait;
```

Algorithm 8: QBUA: ConstructSchedule

```
1:  input: Γ; //Set of threads in the system
2:  input: σ_j^p, H_j ← nil; //σ_j^p: Previous schedule of node j, H_j: set of handlers scheduled
3:  for each T_i ∈ Γ do
4:      if for some section S_j^i belonging to T_i, t_curr + S_j^i.ex > S_j^i.tt then
5:          T_i.PUD ← 0;
6:      else T_i.PUD ← U_i(t_curr+GE_i) / GE_i;
7:  for each task el ∈ σ_j^p do
8:      if el is an exception handler for section S_j^i then Insert(el, H_j, el.tt);
9:  σ_j ← H_j;
10: σ_temp ← sortByPUD(Γ);
11: for each T_i ∈ σ_temp do
12:     T_i.stop ←false;
13:     if did not receive σ_j from node hosting one of T_i's sections S_j^i then
14:         T_i.stop ←true;
15:     for each remaining section, S_j^i, belonging to T_i do
16:         if T_i.PUD > 0 and T_i.stop ≠true then
17:             Insert(S_j^i, σ_j, S_j^i.tt);
18:             if S_j^h ∉ σ_j^p then Insert(S_j^h, σ_j, S_j^h.tt);
19:             if isFeasible(σ_j)=false then
20:                 T_i.stop ←true;
21:                 Remove(S_k^i, σ_k, S_k^i.tt) for 1 ≤ k ≤ j;
22:                 if S_j^i ∉ σ_j^p then Remove(S_j^h, σ_j, S_j^h.tt);
23: for each j ∈ N do
24:     if σ_j ≠ σ_j^p then Mark node j as being affected by current scheduling event;
```

that has already been accepted into the system can execute to completion before its termination time. We do this by inserting the handlers of sections that were part of each node's previous schedule into that node's current schedule (lines 7-9). Since these handlers were part of σ_j^p, and QBUA always maintains the feasibility of a schedule as an algorithm invariant, we are sure that these handlers will execute to completion before their termination times.

In line 10, we sort the threads in the system in non-increasing order of PUD and consider them for scheduling in that order (lines 11-21). In lines 13-14 we mark as failed any thread that has a section hosted on a node that does not participate in the algorithm. If the thread can contribute non-zero utility to the system and the thread has not been rejected from the system, we insert its sections into the scheduling queue of their corresponding node (line 17).

After inserting the section into its corresponding ready queue (at a position reflecting its termination time), we check to see whether this section's handler had been included in the previous schedule of the node. If so, we do not insert the handler into the schedule since this has been already taken care of by lines 7-8. Otherwise, the handler is inserted into its corresponding ready queue (line 18). Once the section, and its handler, have been inserted into the ready queue, we check the feasibility of the schedule (line 19). If the schedule is infeasible, we remove the thread's sections from the schedule (line 21). However, we first check to see whether the section's handler was part of a previous schedule before we remove it (line 22). We perform this check before removing the handler because if the handler was part of a previous schedule, then its section has failed and we should keep its exception handler for clean up purposes. Finally, if the schedule of any node has changed, these nodes are marked to have been affected by the current instance of QBUA (lines 23-24). It is to these nodes that the current node needs to multicast the changes that have occurred (line 4, Algorithm 4). In order to test the feasibility of a schedule, we need to check if all the sections in the schedule can complete before their derived termination times.

We use a function *isFeasible* in Algorithm 8 to determine the feasibility of the schedules we construct. *isFeasible* is a linear time complexity function that iterates over the ready queue and determines whether or not the sections can complete before their termination times.

QBUA's dispatcher is shown in Algorithm 9. Only two scheduling events result in collaborative scheduling, viz: the arrival of a thread into the system, and the failure of a node, all other scheduling events are handled locally. Since we are talking about a partially synchronous system, the FD we use to detect node failures can make mistakes. Thus, QBUA may be started due to an erroneous detection of failure. The this can be reduced by designing a QoS FD [26] with appropriate QoS parameters.

Algorithm 9: QBUA: Event Dispatcher on each node i

```
1  Data: schedevent, current schedule σ_p;
2  switch schedevent do
3      case invocation arrives for S_j^i
4          mark segment S_j^i ready;
5      case segment S_j^i completes
6          remove S_j^i from σ_r,σ_p;
7          remove S_j^h from H;
8          set RE_j^i to zero;
9      case S_j^h ∈ H and S_i^h.st expires
10         mark handler S_j^h ready;
11     case downstream handler S_{j+1}^h completes
12         mark handler S_j^h ready;
13     case handler S_j^h completes
14         remove S_j^h from σ_p,H;
15         notify scheduler for S_{j-1}^h;
16     case new thread, T_i, arrives
17         if origin node, send segments S_j^i to all;
18         pass event to QBUA;
19     case node failure detected
20         pass event to QBUA;
21 execute first ready segment in σ_p;
```

4.4 Properties

We now turn our attention to proving some theoretical results for QBUA. Below, T is the communication delay, and Γ is the set of threads in the system.

Lemma 17. *A node determines whether or not it needs to run an instance of QBUA at most* $4T$ *time units after it detects a distributed scheduling event, with high, computable probability,* P_{lock}.

Proof. When a distributed scheduling event is detected by a node, it contacts the quorum system to determine whether or not to start an instance of QBUA (see Section 4.3.2). T time units is used to contact the quorum nodes, and another T time units is taken for the reply of the quorum nodes to reach the requesting node. After these two communication steps, two outcomes are possible.

One possibility is that a quorum ($\frac{2n}{3}$) of servers receive a particular node's request first and so grant that node permission to run an instance of QBUA (lines 15-23, Algorithm 5). In this case, only $2T$ time units are necessary to come to a decision.

The second possibility is that none of the client nodes receive permission from a quorum of server nodes, and therefore the result of the first round of contention is inconclusive (see Section 4.3.2 for an example). In this case, all nodes send YIELD messages to the server nodes to relinquish the "lock" they where granted to run an instance of QBUA (lines 24-30,

Algorithm 5). As discussed in Section 4.3.2, the above scenario is caused by differences in communication delays between different client-server pairs. However, by the time that the YIELD messages reach the server nodes ($3T$), all client requests must have reached the server nodes and will be present in their waiting queue (line 12, Algorithm 6) so the server nodes can now make a decision about which node gets to run QBUA (lines 22-31, Algorithm 6) by selecting the earliest request in its waiting queue (ties are broken using client ID). Therefore, the contention is resolved in $4T$ messages delays, one T for each of the REQUEST, RESPONSE, YIELD and RESPONSE messages communicated between client server pairs.

Thus, the whole process of using the quorum system to determine whether or not to run an instance of QBUA takes $4T$ time units in the worst case. Since each of the communication delays, T, are random variables with CDF $DELAY(t)$. The probability that a communication round will take more than T time units is $p = 1 - DELAY(T)$. Since there are four communication rounds, the probability that none of these rounds take more than T time units is $P = bino(0,4,p)$, where $bino(x,n,p)$ is the binomial distribution with parameters n and p. Thus the probability that a node determines whether or not it needs to run a version of QBUA after $4T$ is also $P_{lock} = bino(0,4,p)$. □

Lemma 18. *Once a node is granted permission to run an instance of QBUA, it takes* $O(T + N + |\Gamma|\log(|\Gamma|))$ *time units to compute a new schedule, with high, computable, probability,* P_{SWETS}.

Proof. Once a node is granted permission to run an instance of QBUA it executes Algorithm 4. This algorithm has three communication steps, one to broadcast the START message, another to receive the replies from other nodes in the system and one to multicast any changes to affected nodes. Thus the algorithm takes a total of $3T$ time units for its communication with other nodes.

In addition to these communication steps, Algorithm 4 also takes time to actually compute SWETS (line 3). Algorithm 8 is the algorithm that is used to compute SWETS. In this algorithm, lines 3-7 take $|\Gamma|k$ time units for threads with k sections each. The for loop on lines 8-10 will take wN time units to examine the w sections in the scheduling queue of each of the N nodes in the system. Line 12 takes $O(|\Gamma|\log(|\Gamma|))$ time units to sort the threads in non-increasing order of global PUD using quick sort. The two nested loops on lines 13-26 take $|\Gamma|k^2w$ in the worst case, since there are $|\Gamma|$ threads each with k sections to insert into scheduling queues and each queue needs to be tested for feasibility after the insertion of a section using the linear time function *isFeasible* in $O(w)$ time and removing all previously accepted sections, line 24, can take at most O(k) time. Finally, lines 27-29 determines which nodes need to be notified of changes in $O(N)$ time.

Thus the total time complexity of the algorithm is $3T + |\Gamma|k + wN + O(|\Gamma|\log(|\Gamma|)) + |\Gamma|k^2w + O(N)$. If we consider the number of sections in a thread, k, and the number of sections in the waiting queue of a node, w, to be constants, then the asymptotic time complexity of the

algorithm is $O(T+N+|\Gamma|\log(|\Gamma|))$.

There are three communication rounds in this procedure. However, the first two of these communication rounds depend on timeouts (line 2, Algorithm 4), therefore it is only the third that is probabilistic in nature. Therefore, the probability that SWETS is computed in the time derived above is equal to the probability that the nodes receive the multicast message sent on line 4 of Algorithm 4 within T time units. Since the communication delay has CDF $DELAY(t)$, the probability that T is not violated during runtime, and thus that the time bound above is respected, is $P_{SWETS} = DELAY(T)$. □

Theorem 19. *A distributed scheduling event is handled at most $O(T+N+|\Gamma|\log(|\Gamma|)+T_D)$ time units after it occurs, with high, computable, probability, P_{hand}.*

Proof. There are two possible distributed scheduling events: 1) the arrival of a new thread into the system and 2) the failure of a node.

In case of the arrival of a new thread, the root node of that thread immediately attempts to acquire a "lock" on running an instance of QBUA. By Lemma 17, the node takes $4T$ time units to acquire a lock and by Lemma 18, it takes the algorithm $O(T+N+|\Gamma|\log(|\Gamma|))$ to compute SWETS. Therefore, in the case of the arrival of a thread the event is handled $O(T+N+|\Gamma|\log(|\Gamma|)+4T)$=$O(T+N+|\Gamma|\log(|\Gamma|))$ time units after it occurs. Note that $O(T+N+|\Gamma|\log(|\Gamma|))$ is $O(T+N+|\Gamma|\log(|\Gamma|)+T_D)$.

In case of a node failure, some node will detect this failure after T_D time units. That node then attempts to acquire a lock from the quorum system to run an instance of QBUA. By Lemmas 17 and 18, this takes $O(T+N+|\Gamma|\log(|\Gamma|))$ time units. Thus the event is handled $O(T+N+|\Gamma|\log(|\Gamma|)+T_D)$ time units after it occurs.

In both these cases, the result relies on Lemmas 17 and 18, so the probability that events are handled within the time frame mentioned above is $P_{hand} = P_{SWETS} \times P_{lock}$. □

Lemma 20. *The worst case message complexity of the algorithm is $O(n+N)$.*

Proof. The actual message cost of the algorithm is $5n+3N$. The $5n$ component of the message complexity comes from the quorum based arbitration system used in the algorithm. The $5n$ comes from n messages for REQUEST, RESPONSE, YIELD/INQUIRE, RESPONSE and RELEASE respectively. After a node has acquire a "lock", it broadcasts a start message (line 1, Algorithm 4) this takes N messages. The nodes then reply to the current node (line 2, Algorithm 4) using another N messages. Finally, the current node multicasts its results to affected nodes (line 4, Algorithm 4) using another N messages (because in the worst case all nodes in the system may be affected). Thus the actual message complexity of the algorithm is $5n+3N$ which is asymptotically $O(n+N)$. □

Lemma 21. *If all nodes are in the Handler Underload State (Definition 2) and no nodes fail, then no threads will be suggested for rejection by QBUA with high, computable, probability p_{norej}.*

Proof. Since the nodes are all in the Handler Underload State (Definition 2) and no nodes fail, Algorithm 8 ensures that all sections will be accepted for scheduling in the system. Therefore, the only source of thread rejection is if a node does not receive a suggestion from other nodes during the timeout value, $2T$, (see line 2 in Algorithm 4). This can occur due to one of two reasons; 1) the broadcast message (line 1, Algorithm 4), that indicates the start of the algorithm, may not reach some nodes 2) the broadcast message reaches all nodes, but these nodes do not send their suggestions to the node running QBUA during the timeout value assigned to them.

The probability that a node does not receive a message within the timeout value from one of the other nodes is $p = 1 - DELAY(T)$. We consider the broadcast message to be a series of unicasts to all other nodes in the system. Therefore, the probability that the broadcast START message reaches all nodes is $P_{tmp} = bino(0,N,p)$ where $bino(x,n,p)$ is the binomial distribution with parameters n and p. If the START message is received, each node sends its schedule to the node that sent the START message. The probability that none of these messages violate the timeout is $tmp = bino(0,N,p)$. As mentioned before, if none of the nodes miss a message, no threads will be rejected, thus the probability that no threads will be rejected is the product of the probability that the broadcast message reaches all nodes, and the probability that all nodes send their schedule before the timeout expires. Therefore, $p_{norej} = tmp \times P_{tmp}$. □

Lemma 22. *If each section of a thread meets its derived termination time, then under QBUA, the entire thread meets its termination time with high, computable probability, p_{suc}.*

Proof. Since the termination times derived for sections are a function of communication delay and this communication delay is a random variable with CDF $DELAY(t)$ the fact that all sections meet their termination times implies that the whole thread will meet its global termination time only if none of the communication delays used in the derivation are violated during runtime.

Let T be the communication delay used in the derivation of section termination times. The probability that T is violated during runtime is $p = 1 - DELAY(T)$. For a thread with k sections, the probability that none of the section to section transitions incur a communication delay above T is $p_{suc} = bino(0,k,p)$. Therefore, the probability that the thread meets its termination time is also $p_{suc} = bino(0,k,p)$. □

Theorem 23. *If all nodes are in the Handler Underload State (Definition 2), no nodes fail (i.e. $f = 0$) and each thread can be delayed $O(T + N + |\Gamma|\log(|\Gamma|))$ time units once and still be schedulable, QBUA meets all the thread termination times yielding optimal total utility with high, computable, probability, P_{alg}.*

Proof. By Lemma 21, no threads will be considered for rejection from a fault free system in the Handler Underload State (Definition 2) with probability p_{norej}. This means that all sections will be scheduled to meet their derived termination times by Algorithm 8.

By Lemma 22, this implies that each thread, j, will meet its termination time with probability p^j_{suc}. Therefore, for a system with $X = |\Gamma|$ threads, the probability that all threads meet their termination time is $P_{tmp} = \prod_{j=1}^{X} p^j_{suc}$. Given that the probability that all threads will be accepted is p_{norej}, $P_{alg} = P_{tmp} \times p_{norej}$.

We make the requirement that a thread tolerate a delay of $O(T+N+|\Gamma|\log(|\Gamma|))$ time units and still be schedulable because QBUA takes $O(T+N+|\Gamma|\log(|\Gamma|))$ time units to reach its decision about the schedulability of a newly arrived thread. Thus if this delay causes any of the thread's sections to miss their deadlines, the thread will not be schedulable. We only require that the thread suffer this delay *once* because we assume that there is a scheduling coprocessor on each node, thus the delay will only be incurred by the newly arrived thread while other threads continue to execute uninterrupted on the other processor. □

Theorem 24. *If $N-f$ nodes do not crash, are in the Handler Underload State (Definition 2), and all incoming threads can be delayed $O(T+N+|\Gamma|\log(|\Gamma|))$ and still be schedulable, then QBUA meets the execution time of all threads in its eligible execution thread set, Γ, with high computable probability, P_{alg}.*

Proof. As in Lemma 21, no thread in the eligible thread set Γ will be rejected if nodes receive the broadcast START message and respond to that message on time. The probability of these two events is $bino(0,N-f,p)$ where $p = 1 - DELAY(T)$. Therefore, the probability that none of the threads in Γ are rejected is $P_{norej} = bino(0,N-f,p) \times bino(0,N-f,p)$. This means that all the sections belonging to those threads will be scheduled to meet their derived termination times. By Lemma 22, this implies that each of these threads, T_j, will meet their termination times with probability p^j_{suc}. Therefore, for a system with an eligible thread set, Γ, the probability that all threads meet their termination times if their sections meet their termination times is $P_{tmp} = \prod_{j\in\Gamma} p^j_{suc}$. The probability that all the remaining threads are execute to completion is thus $P_{alg} = P_{tmp} \times p_{norej}$. □

Lemma 25. *QBUA has a quorum threshold, m, (see Algorithm 5) of $\lceil \frac{2n}{3} \rceil$ and can tolerate $f^s = \frac{n}{3}$ faulty servers.*

Proof. Our algorithm considers a memoryless crash recovery model for the quorum nodes. This means that a quorum node that crashes and then recovers losses all its state information and starts from scratch. What this implies is that for our algorithm to tolerate such failures, the threshold m should be large enough such that there is at least one correct server in the intersection of any two quorums.

Assume that f is the maximum number of faulty servers in the system (i.e. servers that may fail at some time in the future), then the above requirement can be expresses as $2m - n > f^s$. On the other hand, m cannot be too large since some servers will fail and choosing too large a value of m may mean that client nodes may wait indefinitely for responses from servers that have failed. The requirement translates to $m \leq n - f^s$. Combining the two we get, $f^s = \frac{n}{3}$ and m can be set to $\lceil \frac{2n}{3} \rceil$. □

Definition 4 (Section Failure)**.** *A section, S_j^i, is said to have failed when one or more of the previous head nodes of S_j^i's thread (other than S_j^i's node) has crashed.*

Lemma 26. *If a node hosting a section, S_j^i, of thread T_i fails (per Definition 4) at time t_f, every correct node will include handlers for thread T_i in its schedule by time $t_f + T_D + t_a$, where t_a is an implementation-specific computed execution bound for QBUA calculated per the analysis in Theorem 19, with high, computable, probability, P_{hand}*

Proof. Since the QoS FD we use in this work detects a failed node in T_D time units [26], all nodes in the system will detect the failure of the node at time $t_f + T_D$. As a result, the QBUA algorithm will be triggered and will exclude T_i from the system because node j will not send its schedule (lines 15-16 Algorithm 8). Consequently, Algorithm 8 will include the section handlers for this thread in H. Execution of QBUA completes in time t_a and thus all handlers will be included in H by time $t_f + T_D + t_a$.

Of all these timing terms, only t_a is stochastic. From Theorem 19, we know that t_a will be obeyed with probability P_{hand}, therefore, the time bound derived above is also obeyed with probability P_{hand}. □

Lemma 27. *If a section S_i, where $i \neq k$, fails (per Definition 4) at time t_f and section S_{i+1} is correct, then under QBUA, its handler S_i^h will be released no earlier than S_{i+1}^h's completion and no later than $S_{i+1}^h.tt + T + S_i^h.X - S_i^h.ex$.*

Proof. For $i \neq k$, a section's exception handler can be released due to one of two events; 1) its start time expires (lines 9-10 in Algorithm 9); or 2) an explicit invocation is made by the handler's successor (lines 11-12 in Algorithm 9).

In the first case, we know from the analysis in Section 4.3.2 that the start time of S_i^h is $S_{i+1}^h.tt + S_j^h.X + T - S_j^h.ex$. Thus, by definition, it satisfies the upper bound in the theorem. Also, since $S_j^h.X \geq S_j^h.ex$ (otherwise the handler would not be schedulable), $S_{i+1}^h.tt + S_j^h.X + T - S_j^h.ex > S_{i+1}^h.tt$, and this satisfies the lower bound of the theorem.

In the second case, an explicit message has arrived indicating the completion of S_{i+1}^h. Since the message was sent, this indicates that $S_{i+1}^h.tt$ has already passed, thus satisfying the lower bound of the theorem. In addition, the message should have arrived T time units after S_{i+1}^h finishes execution (i.e at $S_{i+1}^h.tt + T$), since $S_{i+1}^h.tt + T \leq S_{i+1}^h.tt + T + S_i^h.X - S_i^h.ex$ (remember that $S_i^h.X \geq S_i^h.ex$), then the upper bound is satisfied. □

Lemma 28. *If a section S_i fails (per Definition 4), then under QBUA, its handler S_i^h will complete no later than $S_i^h.tt$ (barring S_i^h's failure).*

Proof. If one or more of the previous head nodes of S_i's thread has crashed, it implies that S_i's thread was present in a system wide schedulable set previously constructed. This implies that

S_i and its handler were previously determined to be feasible before $S_i.tt$ and $S_i^h.tt$ respectively (lines 18-26 of Algorithm 8).

When some previous head node of S_i's thread fails, QBUA will be triggered and will remove S_i from the pending queue. In addition, Algorithm 8 will include S_i^h in H and construct a feasible schedule containing S_i^h (lines 8-11 and lines 18-26). Since the schedule is feasible and S_i^h is inserted to meet $S_i^h.tt$ (line 10), then S_i^h will complete by time $S_i^h.tt$. □

We now state QBUA's bounded clean-up property.

Theorem 29. *In the event of a failure of a thread, the thread's handlers will be executed in LIFO (last-in first-out) order. Furthermore, all (correct) handlers will complete in bounded time. For a thread with k sections, handler termination times $S_i^h.X$, which fails at time t_f, and (distributed) scheduler latency t_a, this bound is $T_i.X + \sum_i S_i^h.X + kT + T_D + t_a$, with high computable probability P_{exep}.*

Proof. The LIFO property follows from Lemma 27. Since it is guaranteed that each handler, S_i^h, cannot begin before the termination time of handler S_{i+1}^h (the lower bound in Lemma 27), then we guarantee LIFO execution of the handlers.

The fact that all correct handlers complete in bounded time is shown in Lemma 28, where each correct handler is shown to complete before its termination time.

Finally, if a thread fails at time t_f, all nodes will include handlers for this thread in their schedule by time $t_f + T_D + t_a$ (Lemma 26) with probability P_{hand} and QBUA guarantees that all these sections will complete before their termination times (Lemma 28). Due to the LIFO nature of handler executions, the last handler to execute is the first exception handler, S_1^h. The termination time of this handler (from the equations in Section 4.3.2) is $T_i.X + \sum_i S_i^h.X + kT + T_D + t_a$ (which is basically the sum of the relative termination times of all the exception handlers, plus the termination time of the last section, which is used as an estimate for the worst case failure time of the threads per the discussion in Section 4.3.2, k communication delays T to notify handlers in LIFO order, T_D to detect the failure after it occurs and t_a for QBUA to execute).

Since Lemma 28 guarantees that all handlers will finish before their derived termination times, the only stochastic part of the theorem is the probability that QBUA will include the handlers of all the section in time $t_f + T_D + t_a$. From Lemma 26, we know this probability is P_{hand}, thus $P_{exep} = P_{hand}$. □

Lemma 30. *QBUA has the DBE property for threads that can survive the scheduling overhead of the algorithm — i.e., threads that can be delayed $O(T + N + |\Gamma| \log(|\Gamma|))$ (see Theorem 19) and still be schedulable.*

Proof. The DBE property requires all threads to be ordered in non-decreasing order of global PUD, and this is accomplished in lines 3-7 and line 12 of Algorithm 8. In addition, the DBE

property requires that all feasible threads be scheduled in non-decreasing order of PUD, and this is accomplished in lines 13-26 of Algorithm 8. Thus QBUA has the DBE property for all threads that can withstand the $O(T+N+|\Gamma|\log(|\Gamma|))$ overhead of the algorithm and still remain schedulable. □

Lemma 31. *HUA [115], does not have the **DBE** property.*

Lemma 32. *CUA [114] does not have the DBE property.*

Lemma 33. *ACUA has the **DBE** property for threads that can survive the scheduling overhead of the algorithm — i.e., threads that can be delayed $O(fT+Nk)$ and still be schedulable.*

The proof for Lemmas 31, 32, and 33 can be found in [48].

Theorem 34. *QBUA has a better best-effort property than HUA and CUA and a similar best-effort property to ACUA.*

Proof. The proof follows directly from Lemmas 31, 32, 33 and 30. In particular, HUA and CUA do not have the DBE property while QBUA does, and both QBUA and ACUA have the DBE property but for threads that can survive their, different, scheduling overheads. □

Lemma 35. *The message overhead of QBUA is better than the message overhead of ACUA and scales better with the number of node failures.*

Proof. The message complexity of ACUA is $O(fN^2)$ which is clearly asymptotically more expensive than the $O(n+N)$ message complexity of QBUA. In addition, since the message complexity of ACUA is a linear function of f, the number of failed nodes, and the message complexity of QBUA does not depend on f —i.e., is not affected by the number of node failures— the message overhead of QBUA scales better in the presence of failure. □

Lemma 36. *The time overhead of QBUA is asymptotically similar to the time overhead of ACUA and scales better with the number of node failures. In addition, when the number of threads in the system is fixed, the time complexity of QBUA is asymptotically better than that of ACUA and scales better in the presence of failure.*

Proof. The time complexities of the two algorithms, $O(T+N+|\Gamma|\log(|\Gamma|))$ for QBUA and $O(fT+kN)$ for ACUA, are asymptotically similar, but since the time complexity of ACUA is a function of f and QBUA's is not, QBUA's time complexity scales better in the presence of failure.

When the number of threads in the system is fixed, the term $|\Gamma|\log(|\Gamma|)$ in the time complexity of QBUA becomes a constant and thus its asymptotic time complexity becomes $O(T+N)$ which is better than the time complexity of ACUA, $O(fT+kN)$. Further, since the time complexity of QBUA is not a function of f and the time complexity of ACUA is, QBUA's complexity scales better in the presence of failure. □

Theorem 37. *QBUA has lower overhead than ACUA and its overhead scales better with the number of node failures.*

Proof. The proof follows directly from Lemmas 35 and 36. □

In should be noted that in our computation of time complexity of algorithms, we do not take into account the effect of message overhead. However, the message complexity affects the utilization of the communication channel and the queue delay at nodes and hence has a direct impact on communication delay (which appears as a term in both time complexities mentioned above).

Theorem 38. *QBUA limits thrashing by reducing the number of instances of QBUA spawned by concurrent distributed scheduling event.*

Proof. Thrashing occurs when concurrent distributed events spawn, superfluous, separate instances of QBUA. QBUA prevents this by having nodes wishing to run an instance of QBUA contact a quorum system to gain permission for doing so (see Section 4.3.2). In lines 9-13 of Algorithm 6, the quorum system does not spawn a new instance of QBUA if there is an instance of QBUA already running that was granted permission to start *after* the timestamp of the arriving scheduling event. This occurs because the instance of QBUA that started after the scheduling event occurred will have information about that event and will thus handle it. This reduces thrashing by prevent superfluous concurrent instances from running at the same time. □

4.5 Conclusions

We presented a collaborative, quorum-based thread scheduling algorithm, QBUA, for unreliable distributed real-time systems. The collaborative approach employed allows QBUA to outperform non-collaborative algorithms during overloads.

This occurs because the collaborative approach allows QBUA to take into account global information when constructing the schedule and thus avoid making locally optimal decisions that can compromise global optimality. The collaborative approach also allows the algorithm to take into account node failures and to attempt to maximize timeliness in the presence of these failures.

QBUA is designed for a partially synchronous system where message loss and communication delay is stochastically described. Thus QBUA has better coverage than algorithms designed for fully synchronous systems. The performance and properties of QBUA were analytically established. Experimental comparison of QBUA against other algorithms is performed in Chapter 7.

Chapter 5

Quorum-Based Collaborative Scheduling with Dependencies

5.1 Introduction

In this chapter, we consider the problem of scheduling dependent threads in the presence of uncertainties mentioned in Chapter 1. We design a collaborative thread scheduling algorithm, DQBUA, that can handle distributed dependencies. To the best of our knowledge, this is the first collaborative scheduling algorithm to consider distributed dependencies. We compare DQBUA to several other distributed scheduling algorithms empirically in Chapter 8. This chapter is confined to the algorithm description and theoretical analysis.

5.2 Models and Objective

Since this is essentially an extension the algorithm described in Chapter 4 (QBUA), all of the models used there are the same of this algorithm. However, we introduce, below, the resource model which is not used by QBUA.

Resource Model Threads can access serially reusable non-CPU resources (e.g., disks, NICs) located at their nodes during their execution. We consider the single resource model where only one instance of each resource exists in the system. Resources can be shared under mutual exclusion constraints. A thread may request multiple resources during its lifetime but can only have one outstanding request at any given instance of time. Threads explicitly release all granted requests before the end of their execution.

All resource request/release pairs are assumed to be confined within one node. Thus, a node cannot lock a resource on one node and release it on another. However, it is possible for

a thread to lock a resource on a node and then make a remote invocation to another node (carrying the lock with it). Since resource reqeust/release pairs are confined to one node, the lock is released when the thread's node returns back to the node on which the resource was acquired.

Resources are assumed to access their resources in an arbitrary order — i.e., which resources are needed by which threads is not known a priori. Consequently we employ deadlock detection and resolution methods instead of prevention and avoidance techniques. Deadlock resolution is performed by aborting one of the deadlocked threads by executing its exception handler.

Scheduling Objectives. Our primary objective is to design a thread scheduling algorithm that will maximize the total utility accrued by all threads as much as possible in the presence of dependencies. Further, the algorithm must provide assurances on the satisfaction of thread termination times in the presence of (up to f_{max}) crash failures. Moreover, the algorithm must bound the time threads remain in deadlock.

5.3 Algorithm Rationale

In [44], we develop QBUA, a scheduling algorithm for distributable real-time threads in partially synchronous systems. In this work, we extend QBUA by adding resource dependencies and precedence constraints handling capabilities, we call the resulting algorithm DQBUA. As in [31], precedence constraints can be programmed as resource dependencies and are handled the same way.

As in QBUA, when a node detects a distributed scheduling event (the failure of a node, the arrival of a new thread into the system or a resource request) it contacts a quorum system requesting permission to run an instance of DQBUA (in order to construct a global schedule). All other scheduling events, such as resource releases and section completion, are dealt with locally, see Algorithm 10. Once permission is granted, it broadcasts a start of algorithm message to all other nodes requesting their scheduling information. Nodes that receive this message reply by sending their scheduling information. When all nodes have sent their scheduling information to the requesting node, it computes a system-wide schedule, which we call a System Wide Executable Thread Set (or SWETS), and multicasts any updates to nodes whose schedule has been affected.

The purpose of the quorum system is to arbitrate among nodes that detect a distributed scheduling event concurrently. This arbitration reduces thrashing by minimizing the number of instances of DQBUA that are started to handle the same or concurrent scheduling events. Due to space limitations, we do not reproduce the details of the quorum arbitration algorithm, see [44] for details.

While computing a system-wide schedule, threads are ordered in non-increasing order of their

Algorithm 10: DQBUA: Event Dispatcher on each node i

```
1  Data: schedevent, current schedule σ_p;
2  switch schedevent do
3      case invocation arrives for S_j^i
4          mark segment S_j^i ready;
5      case segment S_j^i completes
6          remove S_j^i from σ_p;
7          remove S_j^h from H;
8          set RE_j^i to zero;
9      case S_j^h ∈ H and S_i^h.st expires
10         mark handler S_j^h ready;
11     case downstream handler S_{j+1}^h completes
12         mark handler S_j^h ready;
13     case handler S_j^h completes
14         remove S_j^h from σ_p,H;
15         notify scheduler for S_{j-1}^h;
16     case new thread, T_i, arrives
17         if origin node, send segments S_j^i to all;
18         pass event to DQBUA;
19     case node failure detected
20         pass event to DQBUA;
21     case S_j^i requests a resource
22         pass event to DQBUA;
23     case S_j^i releases a resource
24         free resource;
25 execute first ready segment in σ_p;
```

global Potential Utility Density (PUD) (which we define as the ratio of a thread's utility to its remaining execution time), the threads are then considered for scheduling in that order. Favoring high global PUD threads allows us to select threads for scheduling that result in the most increase in system utility for the least effort. This heuristic attempts to maximize total accrued utility [31].

Both local and distributed resource dependencies are possible, therefore both local and distributed deadlock can occur. By considering resource requests as distributed scheduling events, DQBUA detects and resolves both local and distributed deadlock in a timely manner. In addition, contention for resources is resolved using their global PUD.

5.4 Algorithm Description

Once the arbitration phase of the algorithm is complete and a node has been granted permission to run an instance of DQBUA, that node runs the algorithm depicted in Algorithm 11. In Algorithm 11, the node first broadcasts a start of algorithm message (line 1) and then waits $2T$ time units for all nodes in the system to respond by sending their local scheduling information (line 2). After collecting this information, the node computes SWETS (line 3) using Algorithm 13. After computing SWETS, the node contacts affected nodes (i.e. nodes that will have sections added or removed from their schedule).

Algorithm 11: DQBUA: Compute SWETS

```
1: Broadcast start of algorithm message, START;
2: Wait 2T collecting replies from other nodes;
3: Construct SWETS using information collected;
4: Multicast change of schedule to affected nodes;
5: return;
```

Algorithm 13 is used by a client node to compute SWETS once it has received information from all other nodes in the system (line 3 in Algorithm 11). It performs two basic functions, first, it computes a system wide order on threads by computing their global PUD. It then attempts to insert the remaining sections of each thread, in non-increasing order of global PUD, into the scheduling queues of all nodes in the system. After the insertion of each thread, the schedule is checked for feasibility. If it is not feasible, then the thread is removed from SWETS (after scheduling the appropriate exception handler if necessary).

First we need to define the global PUD of a thread. Assume that a thread, T_i, has k sections denoted $\{S_1^i, S_2^i, \cdots, S_k^i\}$. We define the global remaining execution time, GE_i, of the thread to be the sum of the remaining execution times of each of the thread's sections. Let $\{RE_1^i, RE_2^i, \cdots, RE_k^i\}$ be the set of remaining execution times of T_i's sections, then $GE_i = \sum_{j=1}^{k} RE_j^i$. Assuming that we are using step-down TUFs, and T_i's TUF is $U_i(t)$, then its global PUD can be computed as: $T_i.PUD = U_i(t_{curr} + GE_i)/GE_i$, where U is the utility of the thread and t_{curr} is the current time. Using global PUD, we can establish a system wide order on the

threads in non-increasing order of "return on investment". Thus we consider the threads for scheduling in an order that is designed to maximize accrued utility [31].

In the absence of dependencies, the above computation can be used to represent the utility that would be accrued if a thread where to execute immediately. However, since we consider dependencies, the utility of a thread can only be accrued if all the threads it depends on either complete their execution or are aborted first. Therefore when a section requests a resource, we compute its dependency list by following the chain of resource requests and ownership. Since a resource request is a distributed scheduling event, the node that gets permission to run an instance of DQBUA (after arbitration by the quorum system) will be sent all the information necessary for it to compute the dependency chain.

Once the dependency list has been computed, we compute the PUD of the current thread by using a least effort heuristic —i.e., while examining the threads in the dependency list to compute PUD, if it is faster to abort them than to continue execution, then the threads are aborted and vice versa. Thus we compute the PUD of a thread if it is executed as soon as possible. A similar heuristic is used in [31] but for a single processor, in contrast to the distributed system we consider in this work. Note that this heuristic minimizes the amount of time a high utility thread waits for a resource, at the expense of having to possibly re-execute threads that have been aborted (see [31] for details).

Algorithm 12: DQBUA: computePUD

```
1:  Input: T_i, Dep(i,k), j; //j is node where resource request occurred
2:  Util ← 0; Time ← 0; Seen ← ∅;
3:  for each Dep(i,k) do
4:      for each S ∈ Dep(i,k) do
5:          if S.ID ∉ Seen then
6:              Seen ← Seen ∪ S.ID;
7:              //Γ1: sections from S until last visit to j
8:              S.Rem ← Σ_{k∈Γ1} RE_k^{S.ID};
9:              //Γ2: all downstream sections
10:             S.Abort ← Σ_{k∈Γ2} S_k^h.ex;
11:             if S.Abort > S.Rem then
12:                 Time ← Time + S.Rem;
13:                 Util ← Util + U_T(t_curr + S.Rem)
14:             else Time ← Time + S.Abort;
15: Time ← Time + GE_i;
16: Util ← Util + U_i(t_curr + GE_i);
17: T_i.PUD = Util/Time;
18: return T_i.PUD;
```

Since we are computing the PUD of the whole thread, we need to consider the dependencies of all sections belonging to the thread. Therefore, Algorithm 12 considers the dependency list, $Dep(i,k)$, of each of the k sections of thread T_i while computing the PUD. While computing the global PUD of a thread, we take into account the utility of the threads that it depends on. The reason that we do this is that in order to schedule a thread, we need to schedule its dependencies first, so when the thread completes, its dependencies will also have completed thus accruing the utility of both the dependencies and the thread itself. Thus we compute

the utility of completing the current thread as the sum of the utility of the current thread and the threads it depends on (lines 13 and 16). We measure the potential utility of a thread and its dependents as the ratio of the utility they can accrue and the time taken for them to accrue this utility (line 17).

We assume that each thread in the system has a globally unique ID, and that each of the thread's sections, S_i, store this global ID in the variable, $S_i.ID$. Since a thread has multiple sections, each of these sections may be dependent on a number of different sections. It is possible that two sections of a thread are dependent on two sections of another thread. In this case, we should only consider the utility of the dependent thread once (since the utility of this thread will only be accrued once when it completes execution). Therefore, in line 5, we check whether a section in a dependency chain, $Dep(i,k)$, belongs to a thread that has been handled before (because another of its sections is in the dependency list of a different section belonging to the current thread). Only threads that have not been considered before are used to compute the current thread's global PUD.

Note that when computing the time remaining for a section, S, to release a resource (line 8), we consider the remaining time for sections starting from S until the last section belonging to S's thread arrives at the current node j. The reason for this is that we do not know when the resource will be released by section S, however since we assume that all resource request/release pairs occur on the same node (see Section 5.2), the latest time at which the resource will be released is when the last section belonging to S's thread to visit node j terminates.

Similarly, when we compute the abort time for section S (line 10), we only consider the abort times of downstream sections. The reason for this is that DQBUA ensures LIFO execution of abort handlers and therefore the current section's abort handler will only execute after the handlers of its downstream sections have terminated. Note that neither the abort time computed in line 10 nor the remaining time to release the resource computed in line 8 are actual times at which those events will occur (interference by other threads ensures that this is not the case), rather the values are merely used as an indication of the amount of work necessary to abort a section to release its resources or to complete a section to release its resources respectively. Since we consider a heuristic of performing the least amount of work, we choose the scenario that takes the least amount of time (line 13).

When computing the global PUD of a section, we need to have up-to-date information about threads that have a section in $Dep(i,k)$. Since resource requests are distributed scheduling events, this information will be received when all nodes in the system send their scheduling information to the node constructing the schedule in response to its broadcast start of algorithm message (lines 1-2 in Algorithm 11).

We now turn our attention to the method used to check schedule feasibility. For a schedule to be feasible, all the sections it contains should complete their execution before their assigned termination time. Since we are considering threads with end-to-end termination times, the termination time of each section needs to be derived from its thread's end-to-end termination

time. This derivation should ensure that if all the section termination times are met, then the end-to-end termination time of the thread will also be met. For the last section in a thread, we derive its termination time as simply the termination time of the entire thread. The termination time of the other sections is the latest start time of the section's successor minus the communication delay. Thus the section termination times of a thread T_i, with k sections, is:

$$S_j^i.tt = \begin{cases} T_i.tt & j = k \\ S_{j+1}^i.tt - S_{j+1}^i.ex - T & 1 \leq j \leq k-1 \end{cases}$$

where $S_j^i.tt$ denotes section S_j^i's termination time, $T_i.tt$ denotes T_i's termination time, and $S_j^i.ex$ denotes the estimated execution time of section S_j^i. The communication delay, which we denote by T above, is a random variable Δ. Therefore, the value of T can only be determined probabilistically. This implies that if each section meets the termination times computed above, the whole thread will meet its termination time with a certain, high, probability. In addition, each section's handler has a **relative** termination time, $S_j^h.X$. However, a handler's **absolute** termination time is relative to the time it is released, more specifically, the **absolute** termination time of a handler is equal to the sum of the **relative** termination time of the handler and the failure time t_f (which cannot be known a priori). To overcome this problem, we delay the execution of the handler as much as possible, thus leaving room for more important threads. We compute the handler termination times as follows:

$$S_j^h.tt = \begin{cases} S_k^i.tt + S_j^h.X + T_D + t_a & j = k \\ S_{j+1}^h.tt + S_j^h.X + T & 1 \leq j \leq k-1 \end{cases}$$

where $S_j^h.tt$ denotes section handler S_j^h's termination time, $S_j^h.X$ denotes the relative termination time of section handler S_j^h, $S_k^i.tt$ is the termination time of thread i's last section, t_a is a correction factor corresponding to the execution time of the scheduling algorithm, and T_D is the time needed to detect a failure by our QoS FD [26]. From this, we compute latest start times for each handler: $S_j^h.st = S_j^h.tt - S_j^h.ex$ for $1 \leq j \leq k$, where $S_j^h.ex$ denotes the estimated execution time of section handler S_j^h. Using these derived termination times, we can check whether a schedule is feasible or not.

We use a function *isFeasible* to determine the feasibility of the schedules we construct. *isFeasible* is a linear time complexity function that iterates over the ready queue and determines whether or not the sections can complete before their termination times.

In Algorithm 13, each node, j, sends the node running DQBUA its current local schedule σ_j^p. Using these schedules, the node can determine the set of threads, Γ, that are currently in the system. Both these variables are inputs to the scheduling algorithm (lines 1 and 2 in Algorithm 13). In lines 3-8, the algorithm DQBUA computes the global PUD of each thread in Γ. The global PUD is computed by first checking whether all sections in a thread can complete execution before their termination time if they were executed immediately. If this is not the case, the thread is assigned a PUD of zero since it cannot possibly accrue any

Algorithm 13: DQBUA: ConstructSchedule

```
1:  input: Γ; //Set of threads in the system
2:  input: σ_j^p, H_j ← nil; //σ_j^p: Previous schedule of node j, H_j: set of handlers scheduled
3:  for each T_i ∈ Γ do
4:      if for some section S_j^i ∈ T_i, t_curr + S_j^i.ex > S_j^i.tt then T_i.PUD ← 0;
5:      else
6:          Compute Dep(i, j), resolving deadlock if necessary;
7:          T_i.PUD ← ComputePUD(T_i, Dep(i, j));
8:  for each task el ∈ σ_j^p do
9:      if el is an exception handler for section S_j^i then Insert(el, H_j, el.tt);
10: σ_j ← H_j;
11: σ_temp ← sortByPUD(Γ);
12: for each T_i ∈ σ_temp do
13:     T_i.stop ← false;
14:     if do not receive σ_j from node hosting S_j^i ∈ T_i then
15:         T_i.stop ← true;
16:     if T_i.PUD > 0 and T_i.stop ≠ true then
17:         insertByEDF(T_i, Dep(i, j));
18: for each j ∈ N do
19:     if σ_j ≠ σ_j^p then Mark node j as being affected;
```

utility to the system (lines 4). Otherwise, we compute the dependency chain for the thread's sections and call Algorithm 12 to compute the global PUD of the thread (lines 6-7). In line 6, we check for cycles to detect any deadlock that may exist. If a cycle is found, it is broken by aborting the thread with the least PUD by executing its exception handler.

Before we schedule the threads, we need to ensure that the exception handlers of any thread that has already been accepted into the system can execute to completion before its termination time. We do this by inserting the handlers of sections that were part of each node's previous schedule into that node's current schedule (lines 8-9). Since these handlers were part of σ_j^p, and DQBUA always maintains the feasibility of a schedule as an algorithm invariant, we are sure that these handlers will meet their termination times.

In line 11, we sort the threads in the system in non-increasing order of PUD and consider them for scheduling in that order (lines 12-17). In lines 14-15 we mark as failed any thread that has a section hosted on a node that does not participate in the algorithm. If a thread can contribute non-zero utility to the system and the thread has not been rejected from the system, then we insert its sections, and their dependencies, into the scheduling queue of the node responsible for them in non-decreasing order of termination time by calling Algorithm 14 (lines 16-17).

When Algorithm 14 is invoked, a copy is made of the current schedule so that any changes that result in an infeasible schedule can be undone (line 2). We then consider each of the remaining sections of the thread being considered, if the section does not already belong to the current schedule (because it was part of the dependency chain of a previous thread), the section and its handler are inserted into the current schedule (lines 5-7).

Algorithm 14: DQBUA: insertByEDF

```
1:  input: σ_j^p, σ_j;
2:  σ_j^tmp ← σ_j; // make a copy of the schedule
3:  for each remaining section, S_j^i, belonging to T_i do
4:      if S_j^i ∉ σ_j^tmp then
5:          Insert(S_j^i,σ_j^tmp,S_j^i.tt);
6:          TT_cur ← S_j^i.tt;
7:          if S_j^h ∉ σ_j^p then Insert(S_j^h,σ_j^tmp,S_j^h.tt);
8:          for ∀S_n^k ∈ Dep(i, j) do
9:              if S_n^k ∈ σ_n^tmp then
10:                 if S_n^k is an abortion handler then
11:                     Remove all sections belonging to S_n^k's thread;
12:                 TT ← lookUp(S_n^k, σ_n^tmp);
13:                 if TT < TT_cur then
14:                     TT_cur ← TT;
15:                     Continue;
16:                 else
17:                     Remove(S_n^k, σ_n^tmp, TT);
18:                     Insert(S_n^k, σ_n^tmp, TT_cur);
19:                     δ ←TT − TT_cur;
20:                     for all predecessors, S_l^x, of S_n^k do
21:                         //If S_n^k is an abortion handler, S_l^xs are also abortion handlers.
22:                         //Otherwise, S_l^xs are normal sections
23:                         TT ← lookUp(S_l^x,σ_l^tmp); γ ← δ;
24:                         if S_n^k.tt − TT < δ then
25:                             γ ← δ − (S_n^k.tt − TT) ;
26:                         Remove(S_l^x, σ_l^tmp, TT);
27:                         Insert(S_l^x, σ_l^tmp, TT − γ);
28:             else
29:                 TT_cur ← min(TT_cur, S_n^k.tt);
30:                 Insert(S_n^k, σ_n^tmp, TT_cur);
31:                 if S_n^k is not an abortion handler then
32:                     if S_n^h ∉ σ_n^p then Insert(S_n^h,σ_n^tmp,S_n^h.tt);
33: if isFeasible(σ_j^tmp 's)=true then
34:     σ_j ← σ_j^tmp for all j ;
35: return σ_j for all j;
```

We then consider the dependencies of that section (lines 8-32). Although sections are considered for scheduling in non-increasing order of global PUD, they are inserted into the schedule in non-decreasing termination time order. Thus during underloads (see Definition 2, when no threads are rejected, the resulting schedule is basically a deadline ordered list. So during underloads, our scheduling algorithm defaults to Earliest Deadline First (EDF) scheduling, which is an optimal real-time scheduling algorithm [91] that accrues 100% utility during underloads. Note that if a section, S_n^k, in the dependency chain, $Dep(i, j)$, needs to be aborted in order to reduce the blocking time of a thread, then all the sections belonging to S_n^k's thread need to be aborted as well (lines 10-11).

In order to ensure that the order of the dependencies is maintained, if the termination time of a section is greater than the termination time of a section that depends on it, its termination time is moved up to the termination time of the section that depends on it (lines 17 and 27). In addition, all the predecessors of that current section have their termination time adjusted to reflect this new value (lines 20-27).

5.5 Algorithm Properties

We now turn our attention to proving some theoretical results for the algorithm. Below, T is the communication delay, Γ is the set of threads in the system and k is the maximum number of sections in a thread.

Lemma 39. *A node determines whether or not it needs to run an instance of DQBUA at most* $4T$ *time units after it detects a distributed scheduling event, with high, computable probability,* P_{lock}.

Proof. This follows from Lemma 1 in [44]. □

Lemma 40. *Once a node is granted permission to run an instance of DQBUA, it takes* $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ *time units to compute a new schedule, with high, computable, probability,* P_{SWETS}.

Proof. Once a node is granted permission to run an instance of DQBUA it executes Algorithm 11. This algorithm has three communication steps, one to broadcast the START message, another to receive the replies from other nodes in the system and one to multicast any changes to affected nodes. Thus the algorithm takes a total of $3T$ time units for its communication with other nodes. In addition to these communication steps, Algorithm 11 also takes time to actually compute SWETS (line 3). Algorithm 13 is the algorithm that is used to compute SWETS. In this algorithm, lines 3-7 iterate $|\Gamma|$ times, and the function computePUD, invoked in line 7, takes $|\Gamma|k^2$ time in the worst case. Therefore the time complexity of lines 3-7 is $|\Gamma|^2 k^2$.

Lines 8-9 take $O(|\Gamma|k)$ time in the worst case, line 11 sorts the threads in $O(|\Gamma|\log(|\Gamma|))$ time. The for loop in lines 12-17 iterates $|\Gamma|$ times in the worst case. The body of this loop calls Algorithm 14 in line 17. The time complexity of Algorithm 14 is dominated by three nested loops in lines 3-32. The outer loop iterates k times in the worst case, the middle loop (starting at line 8) iterates $O(|\Gamma|k)$ times in the worst case, and the inner most loop (starting at line 20) iterates k times in the worst case. The time complexity of the body of the inner loop is dominated by the time complexity of the peek, insert and remove operations (lines 23, 26 and 27 respectively). Using self-balancing binary search trees to represent the queues, all these operations can be performed in $O(\log(|\Gamma|k))$ time. Therefore the nested loop structure, and thus Algorithm 14, has a time complexity of $O(|\Gamma|k^3\log(|\Gamma|k))$. Therefore the time complexity of lines 12-17 is $O(|\Gamma|^2k^3\log(|\Gamma|k))$.

Thus the complexity of the algorithm is $O(|\Gamma|^2k^2+|\Gamma|k+|\Gamma|\log(|\Gamma|)+|\Gamma|^2k^3\log(|\Gamma|k))$, which is asymptotically $O(|\Gamma|^2k^3\log(|\Gamma|k))$. Adding the communication delay to this computational complexity we get $O(|\Gamma|^2k^3\log(|\Gamma|k)+3T)$, which is asymptotically $O(|\Gamma|^2k^3\log(|\Gamma|k)+T)$. There are three communication rounds in this procedure. However, the first two of these communication rounds depend on timeouts (line 2, Algorithm 11), therefore it is only the third that is probabilistic in nature. Therefore, the probability that SWETS is computed in the time derived above is equal to the probability that the nodes receive the multicast message sent on line 4 of Algorithm 11 within T time units. Since the communication delay has CDF $DELAY(t)$, the probability that T is not violated during runtime, and thus that the time bound above is respected, is $P_{SWETS} = DELAY(T)$. □

Theorem 41. *A distributed scheduling event is handled at most $O(|\Gamma|^2k^3\log(|\Gamma|k)+T+T_D)$ time units after it occurs, with high, computable, probability, P_{hand}.*

Proof. There are three possible distributed scheduling events: 1) the arrival of a new thread into the system, 2) a resource request and 3) the failure of a node.

In case of the arrival of a new thread or a resource request, the head node of the thread immediately attempts to acquire a "lock" on running an instance of DQBUA. By Lemma 39, the node takes $4T$ time units to acquire a lock and by Lemma 40, it takes the algorithm $O(|\Gamma|^2k^3\log(|\Gamma|k)+T)$ to compute SWETS. Therefore, in the case of the arrival of a thread or a resource request the event is handled $O(|\Gamma|^2k^3\log(|\Gamma|k)+T+4T)$=$O(|\Gamma|^2k^3\log(|\Gamma|k)+T)$ time units after it occurs. Note that $O(|\Gamma|^2k^3\log(|\Gamma|k)+T)$ is $O(|\Gamma|^2k^3\log(|\Gamma|k)+T+T_D)$.

In case of a node failure, some node will detect this failure after T_D time units. That node then attempts to acquire a lock from the quorum system to run an instance of DQBUA. By Lemmas 39 and 40, this takes $O(|\Gamma|^2k^3\log(|\Gamma|k)+T)$ time units. Thus the event is handled $O(|\Gamma|^2k^3\log(|\Gamma|k)+T+T_D)$ time units after it occurs.

In both these cases, the result relies on Lemmas 39 and 40, so the probability that events are handled within the time frame mentioned above is $P_{hand} = P_{SWETS} \times P_{lock}$. □

Lemma 42. *If all nodes are in the Handler Underload State (Definition 2) and no nodes fail,*

then no threads will be suggested for rejection by DQBUA with high, computable, probability p_{norej}.

Proof. Since the nodes are all in the Handler Underload State (Definition 2) and no nodes fail, Algorithm 13 ensures that all sections will be accepted for scheduling in the system. Therefore, the only source of thread rejection is if a node does not receive a suggestion from other nodes during the timeout value, $2T$, (see line 2 in Algorithm 11). This can occur due to one of two reasons; 1) the broadcast message (line 1, Algorithm 11), that indicates the start of the algorithm, may not reach some nodes 2) the broadcast message reaches all nodes, but these nodes do not send their suggestions to the node running DQBUA during the timeout value assigned to them.

The probability that a node does not receive a message within the timeout value from one of the other nodes is $p = 1 - DELAY(T)$. We consider the broadcast message to be a series of unicasts to all other nodes in the system. Therefore, the probability that the broadcast START message reaches all nodes is $P_{tmp} = bino(0,N,p)$ where $bino(x,n,p)$ is the binomial distribution with parameters n and p. If the START message is received, each node sends its schedule to the node that sent the START message. The probability that none of these messages violate the timeout is $tmp = bino(0,N,p)$. As mentioned before, if none of the nodes miss a message, no threads will be rejected, thus the probability that no threads will be rejected is the product of the probability that the broadcast message reaches all nodes, and the probability that all nodes send their schedule before the timeout expires. Therefore, $p_{norej} = tmp \times P_{tmp}$. □

Lemma 43. *If each section of a thread meets its derived termination time, then under DQBUA, the entire thread meets its termination time with high, computable probability,* p_{suc}.

Proof. Since the termination times derived for sections are a function of communication delay and this communication delay is a random variable with CDF $DELAY(t)$ the fact that all sections meet their termination times implies that the whole thread will meet its global termination time only if none of the communication delays used in the derivation are violated during runtime.

Let T be the communication delay used in the derivation of section termination times. The probability that T is violated during runtime is $p = 1 - DELAY(T)$. For a thread with k sections, the probability that none of the section to section transitions incur a communication delay above T is $p_{suc} = bino(0,k,p)$. Therefore, the probability that the thread meets its termination time is also $p_{suc} = bino(0,k,p)$. □

Theorem 44. *If all nodes are in the Handler Underload State (Definition 2), no nodes fail (i.e.* $f = 0$*) and each thread can be delayed* $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ *time units once and still be schedulable, DQBUA meets all the thread termination times yielding optimal total utility with high, computable, probability,* P_{alg}.

Proof. By Lemma 42, no threads will be considered for rejection from a fault free system in the Handler Underload State (Definition 2) with probability p_{norej}. This means that all sections will be scheduled to meet their derived termination times by Algorithm 13.

By Lemma 43, this implies that each thread, j, will meet its termination time with probability p^j_{suc}. Therefore, for a system with $X = |\Gamma|$ threads, the probability that all threads meet their termination time is $P_{tmp} = \prod_{j=1}^{X} p^j_{suc}$. Given that the probability that all threads will be accepted is p_{norej}, $P_{alg} = P_{tmp} \times p_{norej}$.

We make the requirement that a thread tolerate a delay of $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units and still be schedulable because DQBUA takes $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units to reach its decision about the schedulability of a newly arrived thread. Thus if this delay causes any of the thread's sections to miss their deadlines, the thread will not be schedulable. We only require that the thread suffer this delay *once* because we assume that there is a scheduling coprocessor on each node, thus the delay will only be incurred by the newly arrived thread while other threads continue to execute uninterrupted on the other processor. □

Theorem 45. *If $N-f$ nodes do not crash, are in the Handler Underload State (Definition 2), and all incoming threads can be delayed $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ and still be schedulable, then DQBUA meets the termination time of all threads in its eligible execution thread set, Γ, with high computable probability, P_{alg}.*

Proof. As in Lemma 42, no thread in the eligible thread set Γ will be rejected if nodes receive the broadcast START message and respond to that message on time. The probability of these two events is $bino(0, N-f, p)$ where $p = 1 - DELAY(T)$. Therefore, the probability that none of the threads in Γ are rejected is $P_{norej} = bino(0, N-f, p) \times bino(0, N-f, p)$. This means that all the sections belonging to those threads will be scheduled to meet their derived termination times. By Lemma 43, this implies that each of these threads, T_j, will meet their termination times with probability p^j_{suc}. Therefore, for a system with an eligible thread set, Γ, the probability that all threads meet their termination times if their sections meet their termination times is $P_{tmp} = \prod_{j \in \Gamma} p^j_{suc}$. The probability that all the remaining threads are execute to completion is thus $P_{alg} = P_{tmp} \times p_{norej}$. The reason for tolerating $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ delay is the same as in Theorem 44. □

Definition 5 (Section Failure)**.** *A section, S^i_j, is said to have failed when one or more of the previous head nodes of S^i_j's thread (other than S^i_j's node) has crashed.*

Lemma 46. *If a node hosting a section, S^i_j, of thread T_i fails (per Definition 5) at time t_f, every correct node will include handlers for thread T_i in its schedule by time $t_f + T_D + t_a$, where t_a is an implementation-specific computed execution bound for DQBUA calculated per the analysis in Theorem 41, with high, computable, probability, P_{hand}*

Proof. Since the QoS FD we use in this work detects a failed node in T_D time units [26], all nodes in the system will detect the failure of the node at time $t_f + T_D$. As a result, the

DQBUA algorithm will be triggered and will exclude T_i from the system because node j will not send its schedule (lines 14-15 Algorithm 13). Consequently, Algorithm 13 will include the section handlers for this thread in the schedule. Execution of DQBUA completes in time t_a and thus all handlers will be included in the schedule by time $t_f + T_D + t_a$.

Of all these timing terms, only t_a is stochastic. From Theorem 41, we know that t_a will be obeyed with probability P_{hand}, therefore, the time bound derived above is also obeyed with probability P_{hand}. □

Lemma 47. *If a section S_i, where $i \neq k$, fails (per Definition 5) at time t_f and section S_{i+1} is correct, then under DQBUA, its handler S_i^h will be released no earlier than S_{i+1}^h's completion and no later than $S_{i+1}^h.tt + T + S_i^h.X - S_i^h.ex$.*

Proof. For $i \neq k$, a section's exception handler can be released due to one of two events; 1) its start time expires; or 2) an explicit invocation is made by the handler's successor.

In the first case, we know from the analysis in Section 5.4 that the start time of S_i^h is $S_{i+1}^h.tt + S_j^h.X + T - S_j^h.ex$. Thus, by definition, it satisfies the upper bound in the theorem. Also, since $S_j^h.X \geq S_j^h.ex$ (otherwise the handler would not be schedulable), $S_{i+1}^h.tt + S_j^h.X + T - S_j^h.ex > S_{i+1}^h.tt$, and this satisfies the lower bound of the theorem.

In the second case, an explicit message has arrived indicating the completion of S_{i+1}^h. Since the message was sent, this indicates that $S_{i+1}^h.tt$ has already passed, thus satisfying the lower bound of the theorem. In addition, the message should have arrived T time units after S_{i+1}^h finishes execution (i.e at $S_{i+1}^h.tt + T$), since $S_{i+1}^h.tt + T \leq S_{i+1}^h.tt + T + S_i^h.X - S_i^h.ex$ (remember that $S_i^h.X \geq S_i^h.ex$), then the upper bound is satisfied. □

An interesting thing about the property above is that it is not probabilistic in nature. At first sight, it would seem that the property is stochastic due to the probabilistic communication delay used in the second case mentioned in the proof. One would expect the upper bound in the property to be respected only probabilistically in the second case. However, if the upper bound is not met in the second case (i.e. the stochastic communication delay causes the notification of the completion of handler S_{i+1}^h to arrive after the upper bound in the theorem), then the first case kicks in and starts the handler before the upper bound expires anyway. Therefore this result is deterministic in nature.

Lemma 48. *If a section S_i fails (per Definition 5), then under DQBUA, its handler S_i^h will complete no later than $S_i^h.tt$ (barring S_i^h's failure).*

Proof. If one or more of the previous head nodes of S_i's thread has crashed, it implies that S_i's thread was present in a system wide schedulable set previously constructed. This implies that S_i and its handler were previously determined to be feasible before $S_i.tt$ and $S_i^h.tt$ respectively (lines 5-7 of Algorithm 14).

When some previous head node of S_i's thread fails, DQBUA will be triggered and will remove S_i from the pending queue. In addition, Algorithm 13 will include S_i^h in H and construct a feasible schedule containing S_i^h (lines 8-9 and line 10). Since the schedule is feasible and S_i^h is inserted to meet $S_i^h.tt$ (line 7, Algorithm 14), then S_i^h will complete by time $S_i^h.tt$. □

Note that the termination times mentioned in the proofs above may be modified to reflect dependencies (lines 20-27 in Algorithm 14). We now state DQBUA's bounded clean-up property.

Theorem 49. *In the event of a failure of a thread, the thread's handlers will be executed in LIFO (last-in first-out) order. Furthermore, all (correct) handlers will complete in bounded time. For a thread with k sections, handler termination times $S_i^h.X$, which fails at time t_f, and (distributed) scheduler latency t_a, this bound is $T_i.X + \sum_i S_i^h.X + kT + T_D + t_a$, with high computable probability P_{exep}.*

Proof. The LIFO property follows from Lemma 47. Since it is guaranteed that each handler, S_i^h, cannot begin before the termination time of handler S_{i+1}^h (the lower bound in Lemma 47), then we guarantee LIFO execution of the handlers.

The fact that all correct handlers complete in bounded time is shown in Lemma 48, where each correct handler is shown to complete before its termination time.

Finally, if a thread fails at time t_f, all nodes will include handlers for this thread in their schedule by time $t_f + T_D + t_a$ (Lemma 46) with probability P_{hand} and DQBUA guarantees that all these sections will complete before their termination times (Lemma 48). Due to the LIFO nature of handler executions, the last handler to execute is the first exception handler, S_1^h. The termination time of this handler (from the equations in Section 5.4) is $T_i.X + \sum_i S_i^h.X + kT + T_D + t_a$ (which is basically the sum of the relative termination times of all the exception handlers, plus the termination time of the last section, which is used as an estimate for the worst case failure time of the threads per the discussion in Section 5.4, k communication delays T to notify handlers in LIFO order, T_D to detect the failure after it occurs and t_a for DQBUA to execute).

Since Lemma 48 guarantees that all handlers will finish before their derived termination times, the only stochastic part of the theorem is the probability that DQBUA will include the handlers of all the section in time $t_f + T_D + t_a$. From Lemma 46, we know this probability is P_{hand}, thus $P_{exep} = P_{hand}$. □

Theorem 50. *A deadlock is resolved in at most $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units by terminating the thread that can contribute the least amount of utility to the system.*

Proof. A resource request is a distributed scheduling event. Therefore, when the resource request that causes a wait-for cycle to form occurs, it is handled in at most $O(|\Gamma|^2 k^3 \log(|\Gamma|k) + T)$ time units (see Theorem 41). While computing the dependency chain in Algorithm 13,

the cycle will be detected and broken by terminating the thread in the cycle with the lowest PUD. The theorem follows. □

Theorem 51. *Resource contention is resolved in order of thread PUD.*

Proof. Threads are ordered according to their PUD in Algorithm 13. Therefore if more than one thread is waiting for a particular resource, threads with higher PUD will be considered before threads with lower PUD. □

Theorem 52. *DQBUA limits thrashing by reducing the number of instances of DQBUA spawned by concurrent distributed scheduling event.*

Proof. This follows from the proof of Theorem 22 of [44]. □

5.6 Conclusions

We presented an algorithm, DQBUA, for scheduling dependent distributable threads in a partially synchronous system. We showed that it accrues optimal utility during underloads and attempts to maximize the accrued utility during overloads. We compare the behavior of DQBUA with several other distributed scheduling algorithms in Chapter 8.

Chapter 6

The ChronOS Distributed Real-Time Linux Kernel

6.1 Introduction

When searching for an appropriate testbed to implement our scheduling algorithms for experimental evaluation, we were faced with a problem – the only platforms that supported distributable threads were either middleware (e.g., [87, 142]) or operating systems that were no longer extant [32]. While middleware can provide a quick way to investigate certain scheduling algorithms, the natural home of schedulers is in the OS kernel. Therefore, in order to create a platform to test our scheduling algorithms, we derived a new kernel, which we call ***ChronOS***, from the Linux kernel.

ChronOS provides distributable threads as a first-class kernel scheduling construct, allows time constraints to be specified using TUFs, implements UA and other real-time scheduling algorithms at the kernel level, and supports the use of a co-scheduling approach to optimize collaborative scheduling. To the best of our knowledge, the ChronOS kernel is the only kernel that allows an extant operating system to provide such functionality.

The rest of this chapter outlines the modifications we made to the stock Linux kernel to implement ChronOS. Our additions are divided into three main parts – 1) Adding deadline and TUF scheduling to the Linux kernel, 2) designing a distributable thread implementation using the existing threading capabilities of GNU/Linux and 3) designing the collaborative scheduling algorithms.

6.2 Local Scheduling

In this section, we describe the method used to implement deadline and TUF schedulers in the Linux kernel. It is on top of this framework that the collaborative scheduling algorithms are designed. There have been a number of attempts to bring real-time extensions to Linux [33]. Two different approaches for including real-time in Linux have emerged. The first, typified by Ingo Molnar's fully preemptible kernel [103] patch, attempts to reduce kernel latencies by several different techniques – this approach is discussed in greater detail later. The second approach, used by real-time applications that require the highest degree of precision and lowest latencies, involve running a minimal real-time kernel on the bare hardware to manage real-time tasks and running the Linux kernel as a low priority thread on top of this minimal kernel. The latter approach can guarantee deterministic microsecond accuracy timeliness. Unfortunately, it is not represented in the Open Source community (one proprietary example that uses this approach is QNX [2]).

In this dissertation, we use the former approach, specifically using Ingo Molnar's PREEMPT patch to minimize kernel latencies. This series of patches has been released under the GPL [53] and is thus ideally suited for academic research. We now turn our attention to briefly describing how the PREEMPT patch attempts to minimize kernel latencies. For further information on real-time extensions to the Linux kernel (such as locking memory pages to prevent page fault latencies) please see [15].

6.2.1 The PREEMPT Patch

The PREEMPT patch attempts to reduce kernel latencies using two different approaches. First, interrupt handlers are moved to high priority kernel threads instead of being part of the monolithic kernel. Since interrupt handlers are moved to threads, they are less likely to block the kernel for a long time processing interrupts. Once this change is made, the largest source of latencies in the kernel becomes critical sections protected using spinlocks. These locks disable preemption and therefore can result in unpredictable latencies.

The PREEMPT patch replaces most of these spinlocks with real-time mutexes, essentially the traditional kernel semaphores with priority inheritance added. This makes critical sections protected by these locks preemptible, thus minimizing latencies. However, there are certain parts of the kernel that really need to disable preemption while executing – one such example is the scheduler itself since it is very hard to imagine scenarios where preempting the scheduler would be a good idea. The PREEMPT patch allows the locks guarding those parts of the kernel to retain the semantics of traditional spinlocks.

Naturally, going through the code to determine which spinlocks can be converted to real-time mutexes and which locks should retain the original semantics of spinlocks is a time-consuming and error-prone process. The original PREEMPT patch retained the original

spinlock semantics for about 90 locks. Since then, several different versions of the PREEMPT patch have been produced that attempt to reduce this number and to further reduce kernel latencies.

The PREEMPT patch has become very popular in the Linux audio community and is distributed in the repositories of most popular Linux distributions. It is on this platform that we build the rest of our work. We now turn our attention to the approach we use to incorporate deadline and TUF schedulers into the Linux kernel.

6.2.2 Real-time Schedulers

In addition to the PREEMPT patch described in Section 6.2.1, Linux also provides scheduling algorithms for soft real-time applications. Most real-time operating systems provide only one type of scheduling – fixed priority scheduling. In fixed priority scheduling, each task is assigned a fixed priority and the scheduling algorithm favors high priority tasks.

Linux provides two different scheduling algorithms for soft real-time applications, SCHED_FIFO and SCHED_RR. In SCHED_FIFO, a longstanding POSIX-specified feature [74], processes are given the CPU for as long as they want subject only to the arrival of a higher priority task. Fixed priority scheduling maps perfectly to this scheduling algorithm. SCHED_RR is a simple extension to SCHED_FIFO that schedules processes of the same priority in a round robin fashion while favoring high priority tasks. In the rest of this dissertation, we shall concentrate on the SCHED_FIFO scheduling algorithm since it is the base of our scheduling extensions.

SCHED_FIFO: The real-time scheduler in Linux maintains a per processor ready queue for real-time tasks. In the rest of this section, we will consider the uniprocessor case for ease of exposition and since it is this mode we use in our work[1].

In Linux, the real-time ready queue of a processor is represented as a multi-level queue. In this architecture, depicted in Figure 6.1, each level of the queue represents one priority level. For each level, there is a linked list representing the runnable tasks at that priority. The SCHED_FIFO algorithm simply selects the head of the ready queue with the highest priority.

While GNU/Linux systems (in this case, the GNU C library, glibc, and the Linux kernel) attempt to conform to the POSIX [74] standard as much as possible, it is a well known fact that 100% conformance to the standard does not currently exist on GNU/Linux systems. To clarify the scheduling semantics of the system we use, we specify the behavior of SCHED_FIFO tasks on GNU/Linux systems and highlight the minor differences that exist when compared to the POSIX standard. The following scheduling rules apply to SCHED_FIFO tasks on the GNU/Linux systems we use:-

[1]Actually, we use SMP kernels on dual-core machines in our experiments, but we use CPU affinity to relegate the real-time tasks to one core and use the other core to implement the collaborative scheduler and to run the DT API.

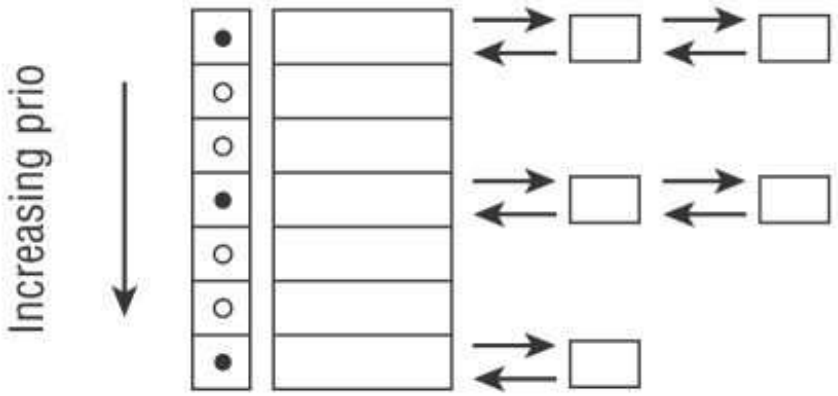

Figure 6.1: RT Ready Queue in Linux [99], used under fair use

- A process that has been preempted by a higher priority process remains at the top of the list for its priority level and resumes execution as soon as higher priority processes finish execution.
- When a SCHED_FIFO process becomes runnable, it is placed at the end of the list for its priority.
- A call to **sched_setscheduler()** or **sched_setparam()** will put the SCHED_FIFO task at the start of the list if it is runnable, thus, possibly, preempting the current running process if it has the same priority (not that the POSIX.1-2001 standard specifies that the task should be placed at the *end* of the list).
- A process calling **sched_yield()** is placed at the end of the list.

It should be noted that the scheduling behavior of GNU/Linux systems, when dealing with FIFO tasks, differs in the third point only from the POSIX standard.

In order to speed up the scheduling function, there is a bitmap, **active.bitmap**, with as many entries as there are priority levels. If a particular priority level has some tasks queued on its ready queue, the corresponding bitmap entry is set to one. When the ready queue at a certain priority level becomes empty, the corresponding bitmap entry is set to zero. This way, the scheduler can efficiently schedule a task by examining the bitmap to determine the highest priority ready queue that contains tasks and then dispatching the top of that queue. This scheduler has $O(1)$ complexity.

Our Schedulers: In order to cause minimal disruption to the scheduling framework of the Linux kernel, we chose to implement deadline and TUF schedulers as sub-types of the SCHED_FIFO scheduling class. We therefore extended the **task_struct** struct, which is used

in the kernel to represent processes and threads[2], by adding a new data member **rt_info**. **rt_info** is a struct that contains the information specific to our scheduling algorithms. Most importantly, it contains a member, **rt_sched**, that can be used to specify which scheduling algorithm is to be executed. This member is only examined if the task is in the SCHED_FIFO scheduling class. We define a number of constants to represent each of the scheduling algorithms we implement and export them in a file to user-space so that they can be used when writing real-time programs.

The **rt_info** struct also contains information that our schedulers use but that is not typically included in the standard Linux **task_struct**. Such information includes deadlines, periods, good faith estimates of execution time, utility of tasks, etc. A full list of all the data members of **rt_info** is presented in Appendix A.

The Linux scheduler calls the **pick_next_task_rt** function to select the next rt task to run. We take advantage of this fact by calling our scheduling algorithms from within this function and setting the return value of **pick_next_task_rt** to the return value of our scheduling algorithm. Thus we keep all the dispatching and scheduling Linux code unchanged and only modify the portion that selects the next task to be executed.

The scheduling API we export to user-space starts a real-time task as a high priority SCHED_FIFO task. It then calls **setrtinfo**, a custom system call we added to the Linux kernel. This system call fills the **rt_info** struct with the appropriate scheduling parameters, reduces the priority of the task and then sets the **TIF_NEED_RESCHED** flag of the current task, causing **schedule** to be called on return from the system call. Note that the task always starts at a higher priority than the rest of the real-time tasks managed by our algorithm. This allows it to immediately gain access to the processor and invoke the system call that sends its scheduling parameters to the kernel.

Instead of calling **schedule** manually from within **setrtinfo**, we set the **TIF_NEED_RESCHED** as mentioned above. This flag is checked after a return from a system call. When the kernel finds this flag set in the current process, it deschedules the task, by calling **schedule**, and places it in the same queue as the other real-time tasks managed by our schedulers in the multi-level priority queue depicted in Figure 6.1. The call to **schedule** puts the kernel's scheduling process into motion which eventually ends up calling the **pick_next_task_rt** function which calls our custom schedulers to select the next task to execute – we do this to ensure that **schedule** function is not called in an unsafe state by allowing the kernel to automatically call it when returning from the system call with the **TIF_NEED_RESCHED** flag set.

It should be noted that we use the inverse PUD of a task, instead of its actual PUD, in our UA algorithms to accommodate the lack of floating point operations in the kernel – this is done to avoid the overhead of saving the FPU state on boundaries of each system call. This is a programming specific work around to doing floating point operations in the kernel

[2]Linux implements a 1:1 threading model as we shall discuss below.

and relies on the fact that, in our task sets, utilities are smaller than execution times and therefore the above division will always produce a valid value. Naturally, the relationship between PUD and inverse PUD is inversely proportional and this is taken into account in the code. The alternative for this would be to enable floating point operations for the PUD calculations, but this is not generally considered a good idea and can introduce unnecessary overheads into this simple process.

Timers: We now discuss the time sources available on the Linux kernel and how we use them to provide a high resolution source of time for our real-time tasks. There are various different sources of high precision timers on modern motherboards and processors. In the rest of this discussion, we shall limit our discourse to the x86 architecture since that is the architecture we use in our work. There are four different time sources on the x86 architecture.

- **The PIT Timer:** This is the old timer used on x86 based systems. It is a 16bit low resolution timer that has been traditionally in the Intel 8253 chip, but whose functionality has now largely been moved to the southbridge chipset.
- **The TSC:** The Time Stamp Counter, or TSC, is a high precision source of time on modern processors. It is a per CPU register that keeps track of the time elapsed since the last boot. It provides single digit microsecond accuracy and is very efficient to query since it is onchip. However, since there is a TSC on each chip on a SMP, there is no guarantee that the TSC on these chips will be synchronized. In addition, power management features of modern architectures can disrupt the normal flow of ticks from the TSC in deep power saving states like C2 and C3 [12]. The TSC is implemented as a 64bit register on all x86 processors since the Pentium, and recent Intel processors provide a constant rate TSC that is not affected by CPU frequency scaling.
- **The HPET:** The High Precision Event Timer, or HPET, is a high precision hardware timer that was jointly developed by Intel and Microsoft. Its intended use was in multimedia applications that required time sources more accurate than the 8253 PIT timer. It is a stable time source that does not suffer from the synchronization and power saving problems associated with the TSC on some platforms and provides timing information of equivalent resolution.

  Unfortunately, the HPET is offchip, making its access a relatively expensive operation (with respect to the timescales that its accuracy provides). The HPET is implemented as a 64bit counter which can be, if required, driven in 32bit mode. Another important feature of the HPET timesource is that it can be operated in one-shot mode, which fits very well with the tickless kernel [99].
- **PM Timer:** The Power Management timer, or the ACPI timer, is a 24bit timer present in ACPI hardware. This time source is not affected by CPU frequency scaling, throttling, processor idling and/or voltage scaling. It is thus a fail-safe time source, with resolution higher than the PIT, when the TSC is unstable and the HPET is not present.

In our testbed, we allow Linux to choose the most accurate timesource available on each node. We also disable most of the ACPI menu in the .config file used to configure the kernel expect those parts that allow us access to the ACPI timers. Specifically, we disable frequency scaling and other sources of instability for the TSC. Currently, the TSC is chosen by the kernel on all nodes except one where the TSC is marked as unstable and the HPET is chosen instead. Our experiments on the testbed indicate that there is little difference between the performance of the nodes, regardless of the time source chosen, at the timescales we consider. The clocks of the nodes in the testbed are synchronized using the Network Time Protocol (NTP) [101].

6.3 The Distributable Thread API

In this section, we briefly outline the method we use to implement distributable threads. The DT implementation consists of two parts; 1) a middleware that allows application programmers to make use of the DT semantics in user-space and 2) a set of kernel operations that keep track of the DT as it transitions across nodes. In order to better understand this implementation, we first describe the Linux threading framework.

6.3.1 Threading in Linux

Since the 2.6 kernel, the threading capabilities of Linux have become more POSIX [74] compliant. It should be noted that this compliance is spearheaded by a more compliant version of POSIX threads, currently integrated in the GNU C library, glibc, using features provided by the 2.6 Linux kernel that were not available in the 2.4 kernel. The version of the POSIX thread library provided in glibc (and libc) for earlier versions of the kernel, LinuxThreads, suffered from poor compliance to the POSIX standard, particularly in signal handling, scheduling and inter-process synchronization primitives — this was partly due to the fact that the kernel provided little support for threads. The new threading library, the Native POSIX Thread Library (or NPTL) [40], provides near complete POSIX compliance. Initially developed at Red Hat, it is now fully integrated into glibc. Since its initial inception, the capabilities of NPTL have vastly improved bringing it closer to complete POSIX compliance. It is on this threading library that we base our distributable threads work.

The NPTL provides a 1:1 implementation of threads in Linux. This means that each thread is considered a separate schedulable entity by the kernel. Thus, each thread is represented by a **task_struct** in the kernel and is scheduled using the kernel's scheduling mechanism. The violation of POSIX compliance in signal handling that was present in previous threading libraries on Linux has been addressed and the overhead of the threading library has been significantly reduced.

6.3.2 The Distributable Thread Model

In this section, we briefly describe the threading model we used to implement distributable threads on the Linux kernel. As previously stated, the implementation consists of a middleware and a set of modifications to the Linux kernel. The middleware consists of three different components:-

- **A name server:** This part of the middleware is responsible for maintaining a list of the services (functions) hosted by each node in the system. When the middleware starts, it exchanges information with its peers to get a complete picture of which services are hosted on which nodes. The name server is consulted when a program wishes to make a remote invocation in order to determine the IP address of the node that the remote invocation should be made to. In our system, this IP address can be the loopback address (i.e., 127.0.0.1) if the service requested is hosted on the local host. We do not go into the detailed implementation choices made for the name server in this dissertation since it is orthogonal to our algorithms.
- **A portable interceptor (PI):** The PI is responsible for listening for remote invocations and spawning new threads to accommodate the incoming service requests. The PI is also responsible for sending information about the spawned sections to the kernel so that it has complete knowledge of the identity of the DTs it hosts and their sections. The specifics of this part of the API are covered in detail in the rest of this Chapter as it, the PI, constitutes the core of the DT API.
- **A library of services:** The last component of the DT API is a library of services. This library implements the functionality of the services hosted on the node it is on. The functions in this library are then linked to by the PI when a request for a specific service arrives using Linux's dynamic linking functions [133]. This part of the API is not discussed further in the dissertation.

We now further elaborate on the functionality of the PI. When a remote invocation is made, the PI spawns a new thread to contain the new DT section. Information about this section is sent to the kernel via a set of system calls so that the kernel can maintain an overall view of the DTs it hosts.

In particular, the kernel uses a system-wide id, which we refer to as the **gtid**, to identify the distributable thread for scheduling purposes. Scheduling parameters are propagated to the PI during remote invocation and are enforced locally by the DT API. Each section returns its exit status to the main application so that it can determine whether or not it has successfully completed.

One important feature that should be discussed when explaining the DT API is that each section is represented by a POSIX thread on the local node hosting it. The DT API spawns

a new thread and sets its real-time parameters using the **setrtinfo** system call as indicated above. In our TUF scheduling algorithms, threads that cannot accrue utility or that make the system unschedulable are removed from the system. We accomplish this by sending the appropriate thread the **SIGUSR1** signal. The thread then runs any appropriate exception handlers it has registered in order to bring the system to a safe state and returns a status code to indicate that it was abnormally terminated. Whenever a section starts on a node, its PID (retrieved by the Linux specific **gettid** system call[3]) is sent to the kernel and stored in the DT table. This PID can then be used to communicate with the section by sending it the appropriate signals.

In order to simplify the implementation task, we take the rather unusual approach of considering each job of a real-time task as a separate thread. This fits the aperiodic task model perfectly and can be used to mimic the periodic model by releasing a new thread representing the next invocation of the task using appropriate timers. We use high precision POSIX timers (based on HPET or TSC) to start invocations of the periodic tasks we consider.

6.3.2.1 DT Implementation Details

In this section, we describe, in greater detail, our implementation of the DT abstraction. We added a set of system calls to store and retrieve information about DTs in the kernel. At the kernel level, a table, implemented using kernel linked lists, is used to store information about the DTs it hosts . The table is protected using a traditional spinlock (i.e., **raw_spinlock_t**). Appendix B contains a selected list of the system calls we added to the Linux kernel and a brief description of their function. In the rest of this section, we refer to the system calls by their function only.

We now discuss the DT API and the set of function calls it exposes to the application programmer. In the API, a new type, **dt_handle_t** is defined to identify the DTs. This data type is defined as follows:

```
typedef unsigned long long dt_gtid_t;
typedef struct dt_handle dt_handle_t;
struct dt_handle {
    dt_gtid_t gtid;
    long secid;
    pthread_t ltid;
};
```

The **gtid** member of the struct is used to store a 64 bit unique ID representing the DT, the **secid** allows us to keep track of which section we are currently processing and the **ltid** member is the traditional pthread_t type of the POSIX thread representing the first section

[3]This system call is Linux specific and reduces the portability of the code, but it suffices for the experiments we conduct.

in the DT. It is used to wait for the DT using the standard **pthread_join** NPTL function (this is discussed in further details below).

There are three main functions that a program using DTs needs to call. All of these functions take a **dt_handle_t** as at least one of their parameters. These functions are:-

```
int dt_spawn (dt_handle_t *handle, void (*function)(dt_handle_t*,
                                                    void*), void *args);
int dt_remote_invoke (dt_handle_t*, const char *name, void *args,
                                                 struct rt_info *rt_param);
int dt_join (dt_handle_t *handle);
```

These functions return 0 on success and a negative error code otherwise. We now briefly describe what each of these functions do:

The dt_spawn function: The **dt_spawn** function is used to spawn a new DT. Once this function returns, a new distributed thread has been created and the **pthread_t** handle of the POSIX thread that represents the first section in the DT is placed inside the **ltid** member of the DT's handle. Subsequently, **ltid** can be used to wait for the entire DT to complete execution using the standard **pthread_join** function as we shall see below.

In our implementation of real-time DTs, we chose to use the first section of the DT as a placeholder for the DT that does not perform any real-time computing but, instead, is responsible for calling the real-time sections of the DT. The actual real-time sections are started from within this first section using the **dt_remote_invoke** function. As a matter of fact, we use this function to start local sections as well as remote sections of the DT. Figure 6.2 depicts the interaction between sections and the DT API.

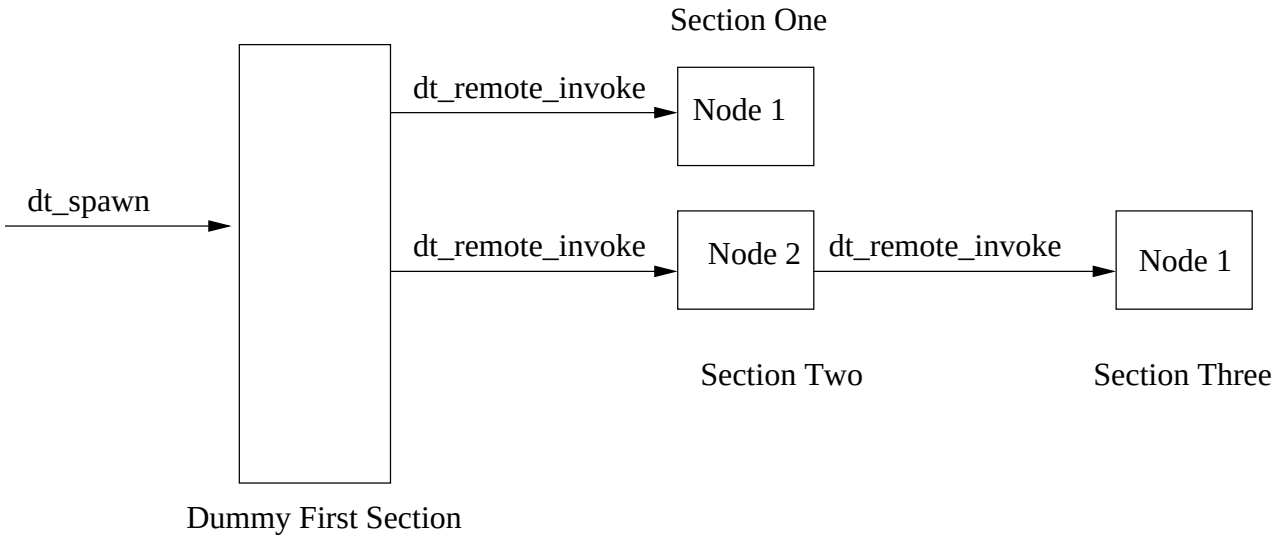

Figure 6.2: Example Invocation

Note that **dt_spawn** is used to create the first "dummy" section that then spawns the

functional sections of the DT by issuing calls to **dt_remote_invoke**. Note also, that **dt_remote_invoke** calls are used to start sections regardless of the identity of nodes they are hosted on. The name server ensures that the invocation is sent to the appropriate node, regardless of whether this node is remote or local.

When the **dt_spawn** function is called, it first invokes a system call that generates a unique 64 bit ID for the new DT being created — this involves creating a new entry for the DT in the kernel DT table. It then starts another thread to actually represent the first section of the DT. This new thread invokes a system call to store the information of the section it represents in the DT's kernel DT table entry.

As mentioned before, each POSIX thread on Linux is represented by a kernel thread. Therefore, we use the Linux specific **gettid()**[4] call to retrieve the PID of the kernel entity representing the current thread. This information is stored in the kernel DT table and is used to communicate with the section, by sending it appropriate signals, when necessary.

After successfully storing this information in the kernel, the function passed in to **dt_spawn**, in its parameter list, is called. It is this function that contains the code of the first section of the DT and can contain any application specific operation. In our case, we use it as the platform from which to invoke the real-time sections we use in our experiment.

When this function call returns, a system call is invoked to remove the information of the current section from the kernel DT table, and the DT entry is cleaned up since the termination of this first "dummy" section of the DT implies that the entire DT has finished execution.

The dt_remote_invoke function: This function is the workhouse of our DT implementation. It is this function that is used to actually invoke the real-time sections in our application.

Part of the DT API is a name server which keeps a list of services that can be executed on each node. We will not go into details of this implementation since it is orthogonal to our algorithms. When the **dt_remote_invoke** function is called, it first sends a query to the name server to see if the service we want to start actually exists on the node we are making the remote invocation to. Note that, as stated earlier, we use this function to call services that are hosted on either the localhost or a remote node. So the name **dt_remote_invoke** may be a bit of a misnomer in this case.

If the service requested does not exist, an error code is returned. Otherwise, the name server returns the IP address of the node that hosts the service we require. If the service does exist, a system call is invoked to check whether an entry for the current DT exists in the kernel or not. If this is the first remote invocation to a remote node, there will be no entry in the kernel to store information about the DT. It is at this point that the entry is created on such nodes.

[4]Since glibc does not provide a wrapper for this system call, we do a raw syscall to 224 which is the syscall number of gettid on the kernel we are using. Once again note that this is not portable.

This is followed by an invocation to a system call that sets the status of the current DT to "remote". This information can be used by other algorithms, such as TMAR protocols, to determine the status and health of a particular DT. Finally, a call is made to a library function (not a system call) that creates a UDP connection to the node hosting the service and sends it the name of the service required and the scheduling parameters to be used to schedule the section that will be spawned to execute this service. Once this information is sent, the thread blocks on a receive from the remote (possibly localhost) node.

When the remote node receives the UDP message, it spawns a new thread to handle it. The spawned thread first checks with the name server to see if the service name it has been sent exist locally. If it does, a system call is invoked to store information about the DT making the invocation. The scheduling information sent is then extracted from the UDP message received and a new thread is invoked and sent this scheduling information. The newly invoked thread is responsible for executing the remote section. We will discuss the reason a separate thread is invoked to execute the local section instead of simply using the thread that was spawned to handle the UDP message presently.

The first thing that the newly created thread does is install a signal handler for the **SIGUSR1** signal. The thread then raises its priority by calling **sched_setscheduler** with SCHED_FIFO as the scheduling class to be used. This is followed by a call to **sched_setaffinity** to change the CPU affinity of the thread to the other processor. Remember that we set the CPU affinity of the **init** process at boot time to force all functionality to move to one processor. We shall call this processor the scheduling co-processor. The scheduling co-processor is responsible for running all the DT API code and the kernel modules implementing the collaborative scheduling algorithms.

The real-time sections themselves are executed on the other processor. The call to **sched_setaffinity** implements this by moving the thread that will be hosting the real-time sections to the other CPU. Once this is done, a call to **setrtinfo** is made and the scheduling parameters received in the UDP message are passed to this function to set the real-time scheduling parameters of the thread.

Immediately following this, a system call is made to add information about the section to the kernel (this system call also sends information to the collaborative scheduling modules if they are active — this is further elaborated on in Section 6.4). This is followed by the actual real-time code. In our case, it is simply a loop that burns CPU time.

Recall that both the kernel modules implementing the collaborative scheduling and the TUF schedulers we added to the kernel send the **SIGUSR1** signal to the thread they want to terminate. The signal handler that is installed when the thread begins is programmed to return a specific code to indicate that the thread has been terminated — appropriate termination handling code is also placed in this handler.

When this thread exits, either by successfully finishing execution or receiving the **SIGUSR1** signal from the scheduler, it returns to the thread that spawned it; namely, the thread that

received the UDP message from the invoking node. This thread then checks the return code of the returning thread to see whether it has successfully finished or if it has been terminated by the scheduler. In case of the former, an "OK" message is sent to the invoking node which, if you recall, is blocked on a receive waiting for a reply from the remote node. In case of the latter, a "TR" message is sent. This way, the invoking node is able to determine whether the remote section has successfully completed or not.

The main reason that we spawn a new thread and then install a signal handler on it instead of installing a signal handler on the thread that receives and sends UDP messages from/to the invoker is that we want to send a message to the invoker and do not want the thread to be terminated until it communicates with the invoker about its status. It would be possible to carefully construct a mechanism for determining whether a message has been sent to the invoker or not and then sending it from the signal handler using asynchronous-safe system calls, but it presents too large a potential for introducing bugs. Therefore, we spawn a new thread and have it execute the section. If this thread receives the **SIGUSR1** signal from the scheduler, it exits and returns control to the thread that is responsible for communicating with the invoker. This thread can then inform the invoker that the section was terminated by the scheduler.

At this point, it is necessary to clarify one point. Specifically, we talk about sending a signal to the thread executing the section. However, the POSIX standard specifies that a signal sent to a process can be handled by any threads that have been spawned by that process. As a matter of fact, one of the major POSIX incompatibility issues that was addressed in NPTL was providing this signal handling semantics. However, we wish to send signals to specific threads within the application — the ones hosting the sections of the DTs. We do this by taking advantage of the fact that the Linux implementation of POSIX threads is a 1:1 implementation — i.e., each thread is represented by a separate kernel schedulable entity.

We use the **send_sig_info** system call, with the **task_struct** of the thread executing the section as its parameter, to ensure that the signal is delivered to this particular thread. The actual **task_struct** is retrieved by making a call to **find_task_by_pid** with the result of **gettid** on the thread being targeted. Note that we may call **send_sig_info** from within the scheduler.

send_sig_info makes a number of invocations to a set of functions that acquire locks protecting scheduling data structures. Since we may call **send_sig_info** from within the scheduler, this results in a classic deadlock lock acquiring pattern with the function trying to acquire a lock that is already held by its invoker. In order to circumvent this problem, we add a field, **about_to_abort** (see Appendix A), to the **task_struct** representing tasks in the kernel. Before calling **send_sig_info** to abort the task, we set this field to one. We then modify the code path taken by **send_sig_info** to avoid acquiring these locks if **about_to_abort** is set. This allows us to avoid the deadlock scenario by not attempting to acquire locks that where already held when **send_sig_info** was called by the scheduler. Mutual exclusion is not sacrificed in this case because the data structures are already protected by the locks acquired

from earlier on in the scheduler.

The dt_join function: This function is very simple compared to the other functions in the API, all it does is call **pthread_join** on the **pthread_t** ID of the first section in the DT. Its purpose it to provide functionality equivalent to **pthread_join** on a DT.

6.4 The Co-Scheduling Approach

As stated earlier, we use a scheduling co-processor to run the scheduler and the DT API. In this section, we describe how this is accomplished. We divide our scheduler into two logical parts. The first part is responsible for responding to distributed scheduling events (i.e., the arrival of new threads into the system, the removal of a thread from the system and node failures) and the second part is responsible for actually performing local scheduling and dispatching of the sections on the node.

Let us first discuss the responsibilities of the first part of the scheduler. Whenever a new thread enters the system, this part of the scheduler is responsible for informing other nodes about the arrival of the thread, sending them the scheduling information of the newly arriving task and requesting the start of collaborative scheduling. Based on the result of collaboration between the nodes, a set of threads, possibly empty, is selected for removal from the system.

This set of threads is sent the **SIGUSR1** signal so that they can execute their appropriate termination handlers. Note that this part of the scheduler is essentially a fancy form of admission control and does not perform any actual dispatching of tasks. Therefore, we implement this functionality in a kernel thread on the scheduling coprocessor using CPU affinity to bind the thread to the appropriate processor. The kernel module is notified of the events it should handle by waiting on a waitqueue. The events are placed on this waitqueue by the system calls issued by the DT API. The actual details of what this kernel module does depends on which algorithm we are implementing, however, there are some commonalities that we shall discuss in greater details in this section.

The second part of our scheduler is responsible for actually selecting one of the threads that remain in the system after the kernel module has eliminated a set of threads from the system in accordance to the scheduling policy it uses. In all our algorithms, the second part of the scheduler implements the EDF scheduling policy — with priority inheritance in the case of DQBUA. This is implement, as stated before, by hooking into the **pick_next_task_rt** function, sorting the ready queue at the appropriate priority in increasing order of deadline and dispatching the top of the queue.

Figure 6.3, depicts the architecture used to implement this general framework. We show two computers only in the diagram for ease of exposition. When a new real-time task arrives at either of the two computers, it sends information about itself to the collaborative scheduling kernel module and then makes a transition to the other CPU in the computer using an

invocation to **sched_setaffinity**. These steps are labeled with a "1" in Figure 6.3 to indicate that they are the first steps that occurs.

Once the call to **sched_setaffinity** succeeds and the task moves to processor one, the collaborative scheduling kernel module starts collaboration with the other nodes in the system (this is step 2 in the process). Once collaboration is over, the result of the collaboration is a set of threads that should be removed from the system. This is accomplished when the collaborative scheduling kernel module sends the **SIGUSR1** signal to the threads that need to be terminated. This is step 3 in the process. The threads remaining after this are scheduled using the EDF discipline on processor one.

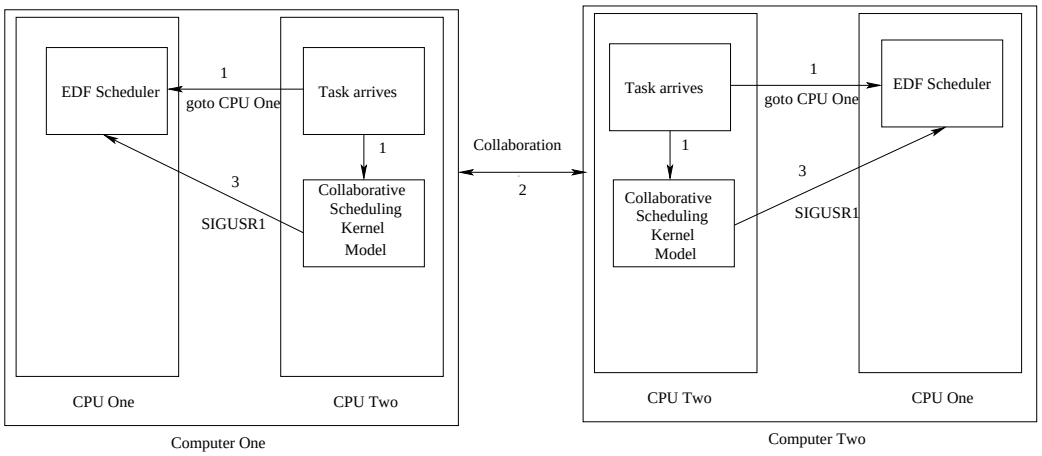

Figure 6.3: Co-Scheduling Approach

6.4.1 Commonalities in the Collaborative Schedulers

The heart of the co-scheduling approach is a high priority kernel module that waits for distributed scheduling events. A wait queue is defined in the kernel to enable communication between the DT API and the kernel module. When the DT API, described in Section 6.3.2, makes system calls involving events of interest to the co-scheduler, a corresponding event is placed on this wait queue for processing by the kernel module. The wait queue is defined as follows:

```
wait_queue_head_t my_wait_queue;
```

In addition to the system calls present in the DT API, we add a number of our own system calls to handle events that are not present in the API. Specifically, our collaborative algorithms require knowledge about all the sections of an arriving thread, present and future. In order to accommodate this, we add a system call that the application can use to register

information about its sections, both current and future, with the kernel module. Before starting a DT, the application should prepare information about the list of sections it intends to execute and send this information to the kernel using the new system call. This information is placed on **my_wait_queue** for processing by the kernel module. We also define another system call that the application can call at the end of its DT invocation so that the memory allocated for the DT in the kernel is released.

These system calls allows us to store information about the created thread locally. We also need to send this information to all other nodes in the system so that they can take part in the collaborative scheduling algorithm. This is accomplished by placing the information about the sections into a UDP packet and sending it to the remote nodes. The remote nodes have a high priority kernel module listening for such messages. Once this module receives such a message, it makes a system call using the information it receives so that the information about the newly arrived thread can be included in the kernel at its end.

This also serves another purpose. Remember that the arrival of a new thread into the system is considered a distributed scheduling event. Therefore, the arrival of a message to the kernel module listening for DT creation messages is also a trigger for collaborative scheduling; as soon as the information about the newly arrived thread is included in the kernel data structures, the co-scheduling kernel module begins executing the collaborative scheduling algorithm.

As previously mentioned, when a particular section arrives into the system, a new POSIX thread is spawned to accommodate it. The PID of this thread is sent to the co-scheduling kernel module via **my_wait_queue**. The main purpose for this operation is the ability to subsequently communicate with the section by sending signals to its PID.

A common feature of all our collaborative scheduling algorithms is the ability to create an EDF schedule and check its feasibility. We provide this functionality in a set of functions that can be called by the kernel module. Of particular interest is the fact that the threads in the system may have arrival times in the future. The code we wrote is able to construct an EDF schedule taking into account these different arrival times. The two functions that perform the functionality of this part of the scheduler are **create_schedule** and **isFeasible**.

When collaborative scheduling is triggered, each node sorts its list of sections according to its algorithm specific criteria (these are different for each of our collaborative algorithms). It then inserts these sections one by one into the ready queue. This is followed by a call to **create_schedule** which essentially sorts the threads into EDF order taking into account the possibly different arrival times. Once this is done, a call to **isFeasible** is made. If the schedule is feasible, the next thread is added to the schedule and the process is repeated. Otherwise, the thread is added to the list of threads to reject.

Depending on the scheduling algorithms being considered, this list of threads to reject are either used as inputs to a round of consensus so that all nodes agree on the set of threads to reject (in case of CUA or ACUA), or are unicast to nodes that host the sections so that

they can be removed (in case of QBUA).

6.4.2 Algorithm Specific Functionality

We now describe a set of algorithm specific functionality used by each of the algorithms we consider. This list of algorithm specific functionality does not list differences in the algorithms themselves (these are described in the relevant chapters), but rather points out additional kernel modules that need to be implemented in order to accommodate the needs of each scheduling algorithm.

6.4.2.1 ACUA

While CUA uses local information only to construct local schedules, ACUA needs information about the entire thread since it performs, among other things, global PUD computations. Therefore, a component that is present in ACUA but not in CUA is a set of kernel modules that exchange information about the current state of their sections prior to triggering the collaborative scheduling algorithm. When a distributed scheduling event occurs, the node on which the event occurs sends a begin algorithm message to all nodes involved. These nodes then go over their DT table data structure (which lists the set of DTs they host), and obtain information about the sections they currently host. This information can be obtained by querying the **task_struct** of the section retrieved by calling **find_task_by_pid** with the PID of the thread representing the section as its parameter. The nodes then send this information to each other.

6.4.2.2 QBUA and DQBUA

Since a single node performs the scheduling for the entire system, this node issues a set of messages to all nodes in the system requesting them to send it a summary of their scheduling state. Each of the nodes in the system have a kernel module that listens for this message. When this kernel module receives the relevant message, it goes over its DT table and retrieves information about the sections its hosts for each of the DTs listed in the DT table. It then sends this information to the node that is responsible for computing the system-wide schedule.

6.4.2.3 DQBUA Lock Specific Implementation

We use futexes [39] to implement locks in our algorithms. We designed two system calls that allow an application to acquire or release a lock. The system call for requesting a lock first checks if the owner field of the data structure representing our lock (see **struct mutex_data**

in Appendix A) contains the current thread's PID. If so, the system call returns successfully. This is done to allow reentrant locking.

If the lock is owned by another thread or is not owned by any thread, the system call updates the **requested_resource** field of the **struct rt_info** (see Appendix A) representing the thread in the kernel to point to the requested resource. This field is used by the schedule to identify the thread's request for a resource.

The system call then places a scheduling event on DQBUA's waitqueue and issues a call to **schedule**. This triggers ChronOS's scheduling mechanism. The scheduling code then uses the **requested_resource** and owner field of the futex to construct a resource graph on the fly and check it for deadlocks. If deadlocks exist, the code terminates the thread with the smallest PUD in order to break the cycle. This resource graph is also used to construct dependency chains for DQBUA.

Upon return from the **schedule** call, the system call updates the futex's state using the atomic **cmpxchg** to indicate that the current task is its new owner. If this does not succeed, a busy wait loop keeps retrying until the lock is acquired. By design of our resource aware scheduling algorithm, spinning is almost entirely precluded since the if a task is selected to execute, it is at the top of a dependency chain (remember that DQBUA schedules all the dependencies of a thread before it executes that thread).

Following this, the system call clears the **requested_resource** field since the thread no longer has a pending resource request and updates the owner field of the futex to point to the PID of the current thread.

Releasing a lock is similarly implemented. First, the owner field of the futex is set to zero, indicating that no thread currently owns the futex, and then all threads waiting on the futex's waiting queue are woken up to compete for the futex. This is a bit inefficient since it introduces the "thundering herd" problem [140], but is sufficient for the purpose of our experiments. Following this, an event is placed on DQBUA's waitqueue and a call to **schedule** is issued to handle the scheduling event represented by the release of a lock.

6.4.3 Uniqueness of PIDs

Since most of the identification of tasks is performed using PIDs in our algorithms, it is important to talk about the PID allocation scheme of the Linux kernel and the "uniqueness" of the resulting PIDs. Specifically, we would like to determine if reuse of PIDs could result in the algorithms sending messages to the wrong thread.

The Linux kernel uses a 32 bit data structure to store the PID of tasks [95]. Thus giving a total possible range of $0 \cdots (2^{32} - 1)$ or 4294967296 PIDs. Of course, not the entire range is used for user processes since part of this space is reserved for special processes (for example, PID 0 is reserved for the kernel and PID 1 for the **init** process).

Although the data structure used provides a range of 4294967296 PIDs, the kernel has a default cap of 32768 on the PID space for compatibility with older versions of Unix® that used smaller 16-bit types for process IDs. This cap can be changed by manipulating */proc/sys/kernel/pid_max*, thus trading a larger PID space for reduced compatibility with older Unix® systems.

Naturally, if the entire PID address space, 4294967296, is used, it is less likely for PID "collisions" to occur. However, even if the reduced address space of 32768 is used, the Linux kernel still provides short term stability and uniqueness in the PID address space by using a strictly linear algorithm for PID selection. If the last PID used is n, then the next assigned PID is $n+1$ regardless of the whether earlier PIDs have been freed for reuse. PIDs are only reused when the address space is exhausted and the algorithm cycles back to the beginning of the PID space.

Therefore, while the Linux kernel does not provide long term uniqueness in the PID address space, it does provide stability and uniqueness in the short term. Our experiments indicate that PIDs do not collide for the applications we use. We expect the issue of PID collisions to occur only when a huge number of threads is spawned in a very short period of time and, in that case, the system is probably going to run into many other problems (such as memory exhaustion) before PID recycling becomes an issue[5].

6.4.4 Experimental Evaluation of Co-Scheduling Approach

In this section, we perform a number of quick experiments to determine whether or not the approach described above can be used to schedule tasks. Toward that end, we designed a DASA scheduler using the co-scheduler approach. Specifically, the kernel module on the co-scheduling processor selects a set of tasks that need to be eliminated according to the DASA scheduling policy and sends them the **SIGUSR1** signal. The threads that are not terminated by the kernel module are scheduled using the EDF policy. We performed this experiments for two simple task sets and recorded the average DSR and AUR for fifty runs each. The 95% confidence interval for these values are also reported. Table 6.1 shows the periods and the utility of the tasks we will consider. The execution time of the tasks was varied to produce utilizations between 10 and 200. The experiments were performed on a machine with the 2.6.24.7 version of the Linux kernel, with the rt27 PREEMPT patch applied and our custom extensions to the kernel included. The node has a 2.60GHz E5300 Intel Dual-Core CPU and 1GB of memory. The TSC is selected as the timesource on this machine by the Linux kernel.

In order to burn a fixed amount of CPU time, we use Interbench [1] to determine the number of loops per unit of time that can be performed on our platform. We then use this

[5]One could argue that such a scenario is most likely the result of poor application design rather than a short-coming of the kernel.

Table 6.1: Some Task Set Parameters

	Period (μsecs)	Utility
T1	500000	17
T2	1000000	4
T3	1500000	24
T4	3000000	39
T5	5000000	18

	Period (μsecs)	Utility
T1	1000000	17
T2	2000000	4
T3	3000000	24
T4	6000000	39
T5	10000000	18

information to create tasks with specific execution times. In order to prevent the loops from being optimized out by gcc, we use the volatile keyword and O2 level of optimization. We performed three different experiments for these task sets; 1) normal DASA without using the DT API, 2) DASA using the DT API (this incurs the overhead of the DT API) and 3) DASA using the co-scheduling kernel module approach (this incurs the overhead of the DT API and that of the co-scheduler approach). Figures 6.4, 6.5, 6.6 and 6.7 depict the result of these experiments.

As can be seen, the third approach (depicted using the legend "DASA DT 3" in the plots), closely tracks the other two approaches. This indicates that the approach is feasible. It should be noticed that there are slight differences between DASA DT 3 and the other two approaches just after the 100% utilization bound is passed and at the upper end of utilizations close to 200%. This occurs due to three reasons; 1) DASA DT 3 performs theoretical analysis and ignores the real overheads inherent in any scheduling mechanism when selecting a set of tasks to eliminate from the schedule and 2) since the DATA DT 3 approach does not stop the entire system when making its decision, it is possible for some tasks that have been selected for elimination by the co-scheduler to actually complete execution before they are notified of the fact that they should be terminated, thus taking up CPU time without benefit and 3) The co-scheduling approach incurs lower overhead than the bare-bones scheduler in certain cases since it eliminates stopping the tasks for the duration of the DASA algorithm (which is more computationally complex than EDF scheduling).

We believe that the third point, that the algorithm does not stop the system for the duration of running the DASA algorithm, is responsible for the gains in both DSR and AUR that can be seen in Figures 6.4 and 6.5. This is also evident in some of the data points in Figures 6.6 and 6.7. As can be seen, the performance of the DASADT3 approach, the co-scheduler approach, is better than the other two approaches for most data points in those plots. We believe that this is due to the fact that the DASA algorithm is run in parallel to other threads in the system on the co-processor and only affects the system when it sends a termination signal to the relevant threads. Otherwise, the system is only stopped for the duration of the EDF algorithm, which is the algorithm used to select the task to run from those tasks that have not been sent a termination signal by the co-scheduler.

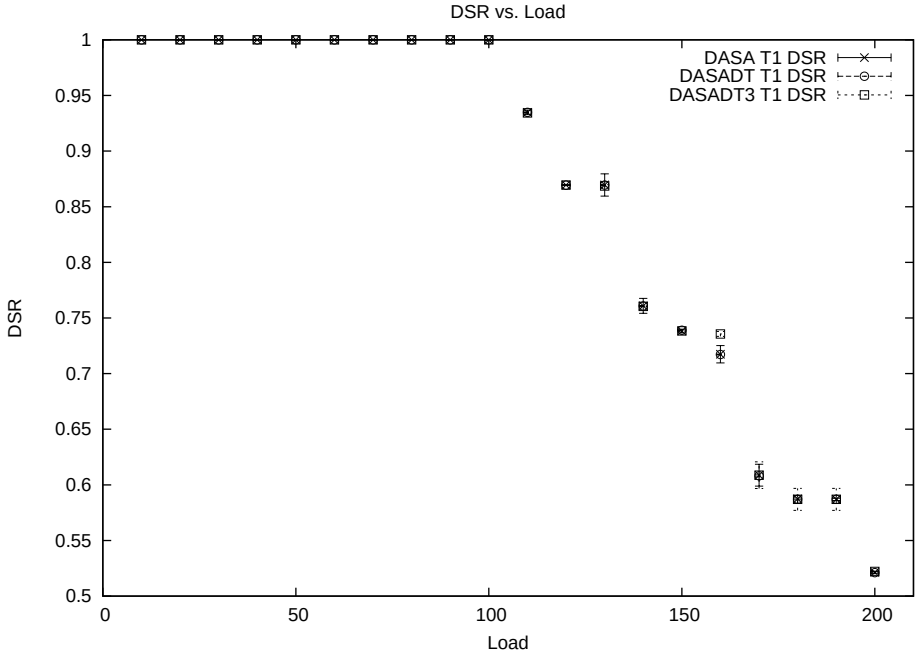

Figure 6.4: DSR vs. Utilization: Task set I

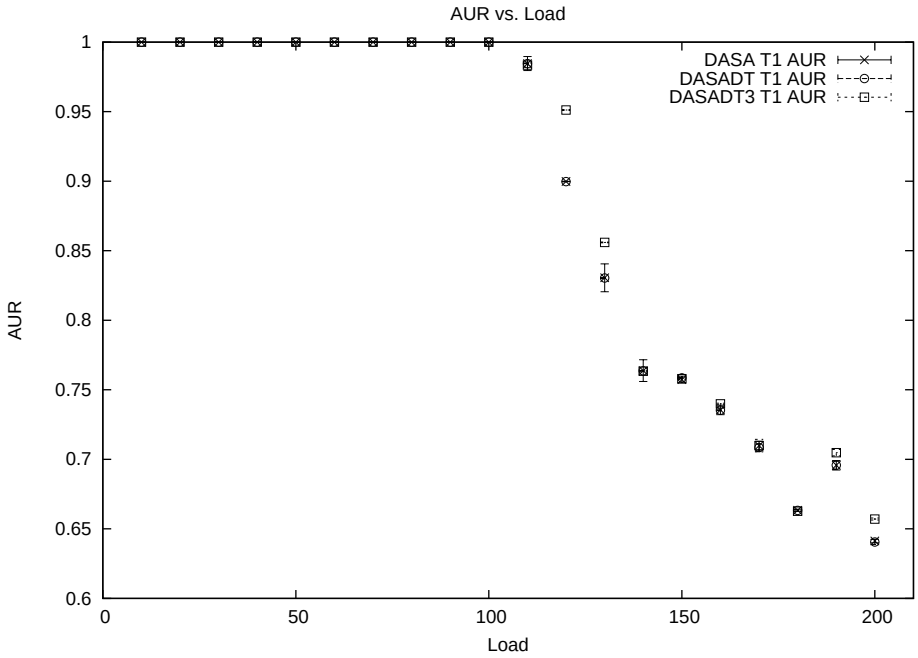

Figure 6.5: AUR vs. Utilization: Task set I

It should be noted, however, that at the 200% data point in Figures 6.6 and 6.7, the co-scheduler approach performs worse than the two others. This is probably due to the first two points mentioned earlier, namely, 1) the theoretical analysis performed by the co-scheduler does not take real overheads into account and 2) it is possible that a task that the co-scheduler selects for elimination may have terminated by the time the co-scheduler has made its decision.

From these observations, we note that there is a period of time during which the co-scheduling approach does not lead to immediate compliance to the policy of maximizing accrued utility. We note that this period of time is equal to the duration of time that the scheduling algorithm is constructing its schedule – because during this time, the tasks in the system continue execution without paying heed to the scheduling algorithm performing its calculations in parallel.

Therefore, the period of time during which the co-scheduling approach does not attempt to maximize utility is equal to the scheduling overhead of the scheduling algorithm executing. Note that this is similar to what would happen if the co-scheduling approach were not used – while the scheduler is executing in a system without the co-scheduling approach, the tasks in the system are stopped and cannot accrue utility and so during that period of time utility accrual is not being maximized.

The main difference between these two scenarios is that while in the co-scheduling approach the system continues execution during this time period, in systems without the co-scheduling approach the system stops execution entirely. We express this behavior in a semi-formal way as follows.

Definition 6 (NBI)**.** *We define the Non-best-effort interval (or NBI) of an algorithm to be equal to the duration of time after the arrival of a scheduling event during which its ready queue is not identical to the ready queue of a hypothetical zero overhead scheduling algorithm whose ready queue is always ordered according to a certain best-effort criteria.*

Note that Defintion 6 is orthogonal to the definition of a particular best-effort criteria. For any given best-effort criteria, NBI is defined as a period of time during which the ready queue, after the arrival of a scheduling event, of a scheduling algorithm is not identical to the ready queue of a hypothetical zero overhead scheduling algorithm that orders its queue according to that particular best-effort criteria.

Given this definition of NBI, we now try to quantify the behavior of the co-scheduling approach.

Claim 53. *The Co-Scheduling approach has a NBI equal to the overhead of the scheduling algorithm it implements.*

Proof. As soon as a scheduling event occurs, the co-scheduler begins execution. Until the co-scheduler is done, the ready queue of the system is not ordered according to the best-effort criteria since the co-scheduler has not finished computing that order. As soon as the

co-scheduler is done, it updates the ready queue of the system and, at that point, the ready queue conforms to a ready queue sorted according to the best-effort scheduling criteria being used. □

Note that the overhead in Claim 53 should not be larger than the execution time of one task in the system. If it is, then that particular task cannot be scheduling using this algorithm in the first place. As a matter of fact, the smaller the ratio between this overhead and average task execution times, the better the compliance of the algorithm to the best-effort property. Note that a similar relationship exists in scheduling algorithms that do not use the co-scheduling approach.

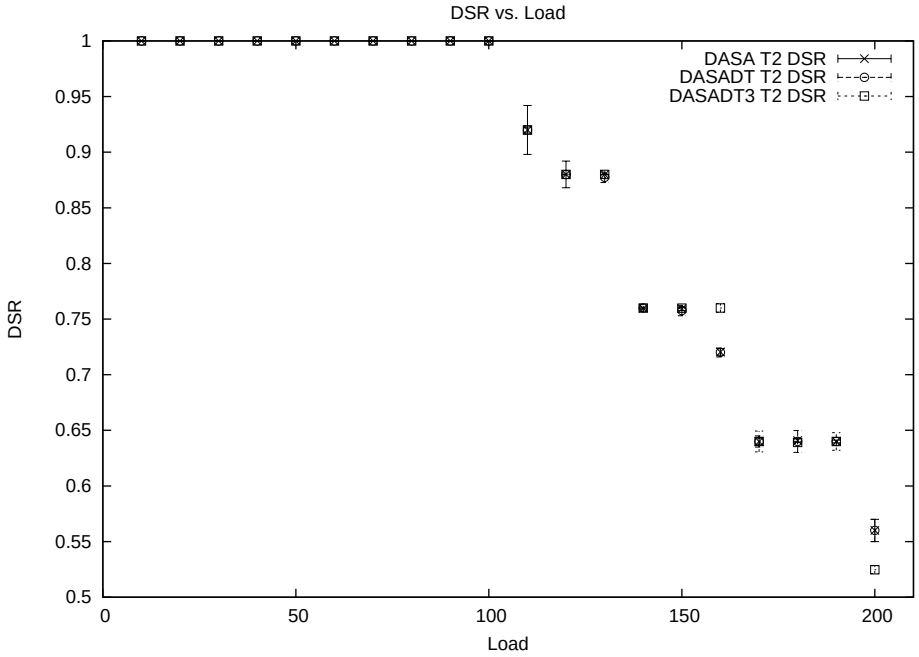

Figure 6.6: DSR vs. Utilization: Task set II

6.5 Choosing the Case Two Competitor

As previously mentioned, we compare the performance of our collaborative scheduling algorithms to case two scheduling algorithms. Specifically, in our first set of experiments, we compare the performance of the collaborative scheduling algorithms to the performance of HUA [115]. HUA defaults to the scheduling semantics of case two scheduling with DASA scheduling performed on each node when the overhead of failure handlers is negligible.

In this section, we perform a number of experiments to compare a number of case two scheduling algorithms. We perform these experiments to verify that the performance of

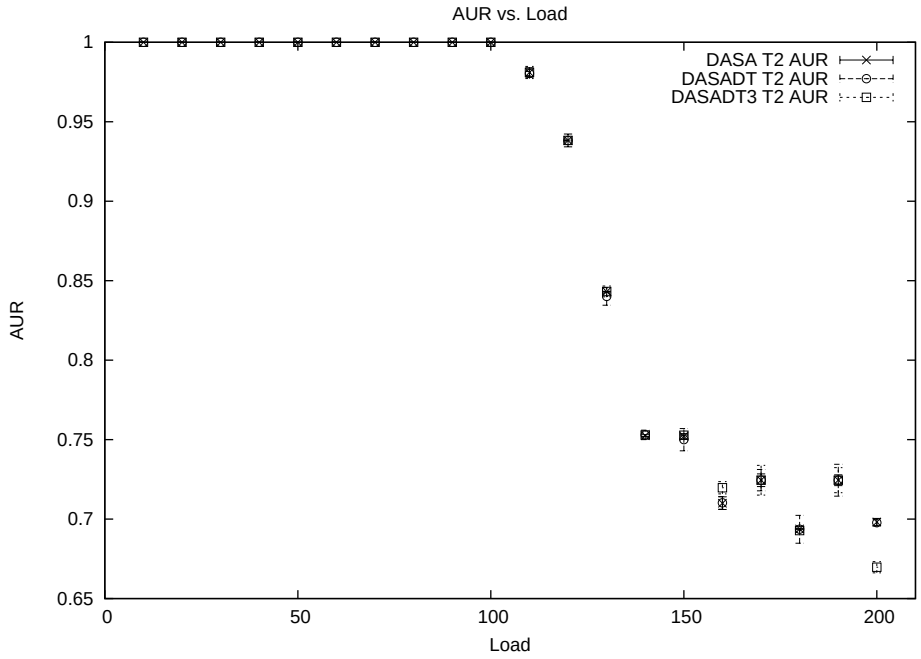

Figure 6.7: AUR vs. Utilization: Task set II

case two scheduling with DASA, and hence HUA, is superior to other case two scheduling algorithms. We then use HUA as the case two competitor in the rest of this chapter.

We compare four different case two scheduling algorithms in this section, namely; case two with DASA (equivalent to HUA when failure handler overheads are negligible), case two with EDF, case two scheduling with RMS (the approach used in [130]), and case two scheduling with LBESA.

Figures 6.8, 6.9, 6.10 and 6.11 depict the result of this experiment for task set described in Table 7.2. As Figures 6.10 and 6.11 indicate, case two with DASA always performs better than the other case two scheduling algorithms, with LBESA a close second.

As expected, case two scheduling with EDF is optimal during underloads (i.e., it does not miss any deadlines as long as it is possible to meet all deadlines) while case two with RMS is not. This is evident in Figure 6.10 where case two with RMS begins to miss deadlines at the 140% transactional load point, while case two with EDF only begins to miss deadlines after the 180% transactional load point.

However, after the 180% transactional point is passed, case two with EDF quickly degenerates into very poor performance (with DSR only slightly above 0.1 at the 240% transactional load point). At the same time, RMS performs quite well, with DSR competitive with DASA and LBESA. A glance at Figure 6.11 shows that despite the fact that RMS has DSR competitive with DASA and LBESA (and sometimes has better DSR than DASA and LBESA), the UA algorithms consistently outperform it when AUR is the metric. This occurs because the UA

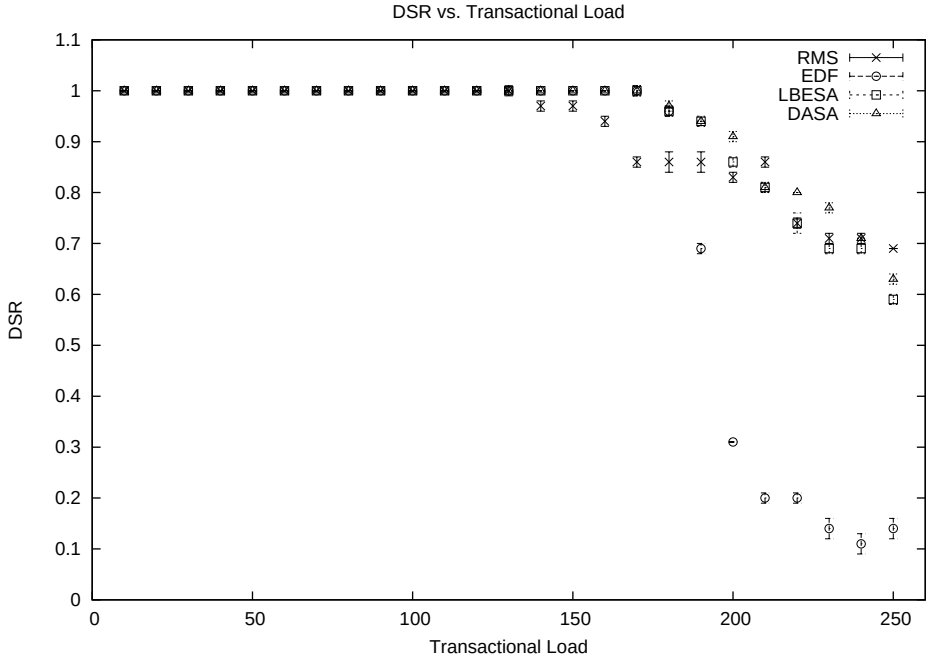

Figure 6.8: DSR vs. Utilization: T9

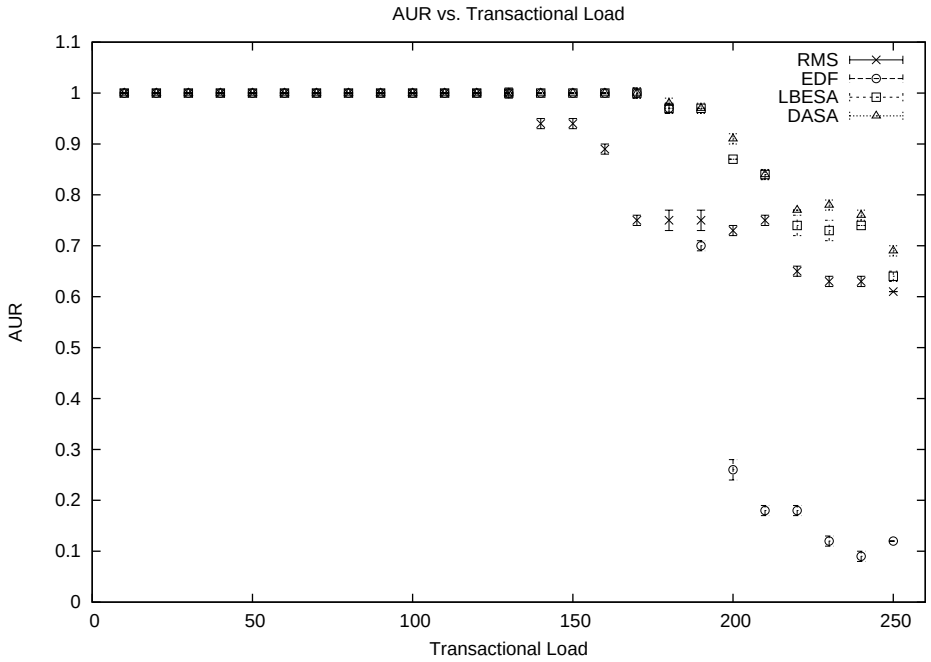

Figure 6.9: AUR vs. Utilization: T9

algorithms choose the "right" set of tasks in order to increase the utility accrued.

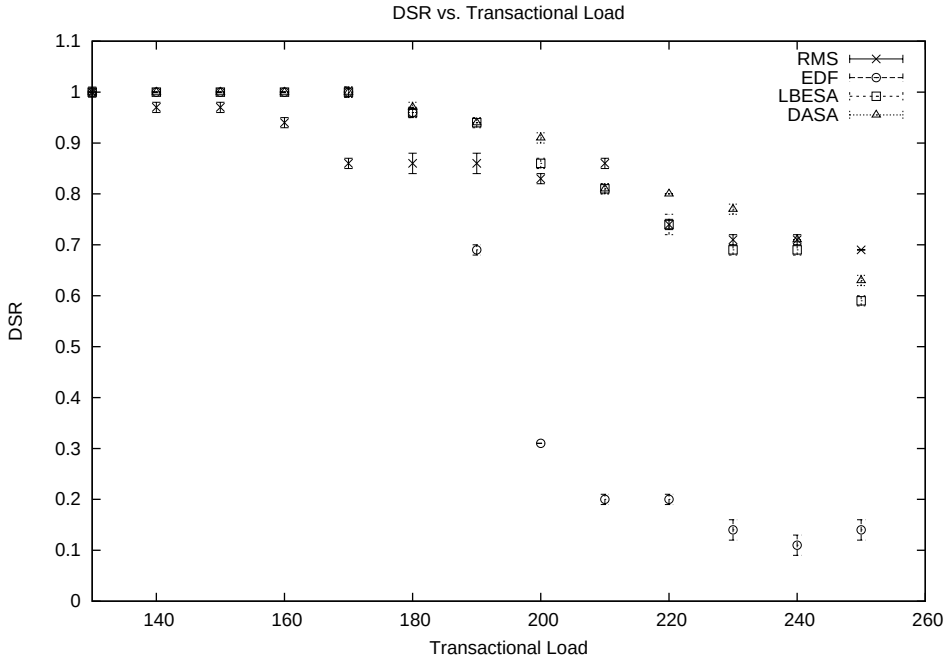

Figure 6.10: DSR vs. Utilization: T1, During overloads

In the previous experiment, the thread utilities are randomly assigned. In order to further elucidate the relative behavior of the case two scheduling algorithms. We try the same experiment with the utilities assigned in inverse proportion to task periods and with the utilities assigned in direct in direct proportion to the task periods.

The results in Figures 6.12, 6.13, 6.14 and 6.15 are for the inversely proportional assignment of utilities i.e., threads with short periods are assigned high utilities while threads with longer periods are assigned small utilities. Naturally, this biases towards RMS because that algorithm selects the tasks with the smaller periods and will thus always favor high utility tasks. The consequences of this can be see in the plots, particularly Figure 6.15 where the AUR accrued by RMS is quite close to the utility accrual algorithms. It should be note that the utility scheduling algorithms still manage to accrue utility at least equal to the RMS algorithm and out perform RMS in several data points because the heuristic they use allows them to always favor the set of tasks that will accrue the most utility to the system for the least effort. On the other hand, RMS, when the inversely proportional method of assigning utilities is employed, always favors high utility tasks regardless of the amount of effort that needs to be invested in these tasks in order to take advantage of their utilities.

The results are different for the proportional assignment of utilities though. In this case, threads with short periods are assigned small utilities while tasks with large periods are assigned large utilities. Naturally, this biases against algorithms that do not use utilities in their scheduling decisions, but rather depend on periods and/or deadlines to reach their

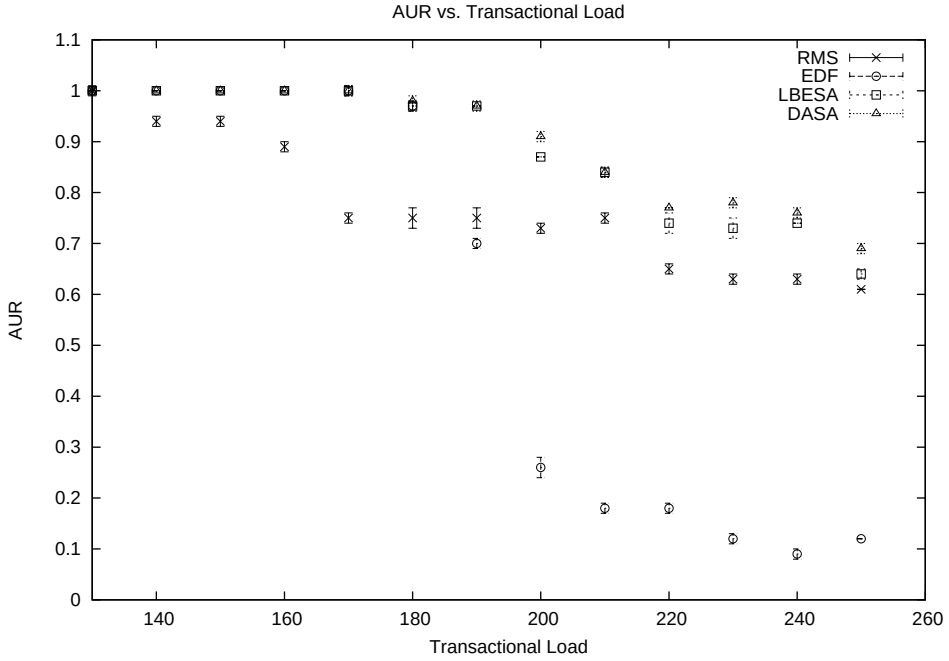

Figure 6.11: AUR vs. Utilization: T1, During overloads

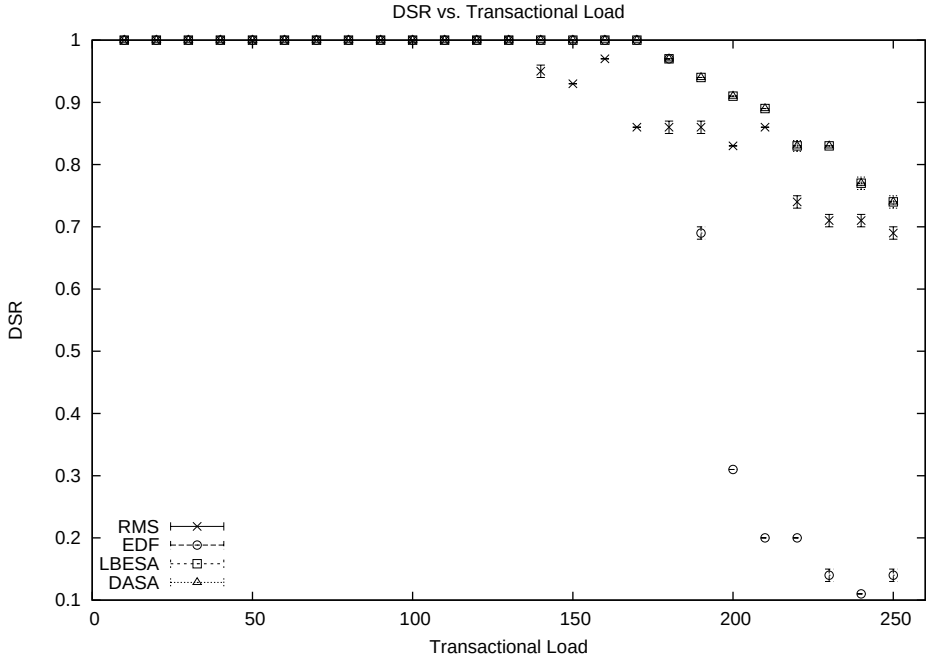

Figure 6.12: DSR vs. Utilization: T1, Inversely Proportional

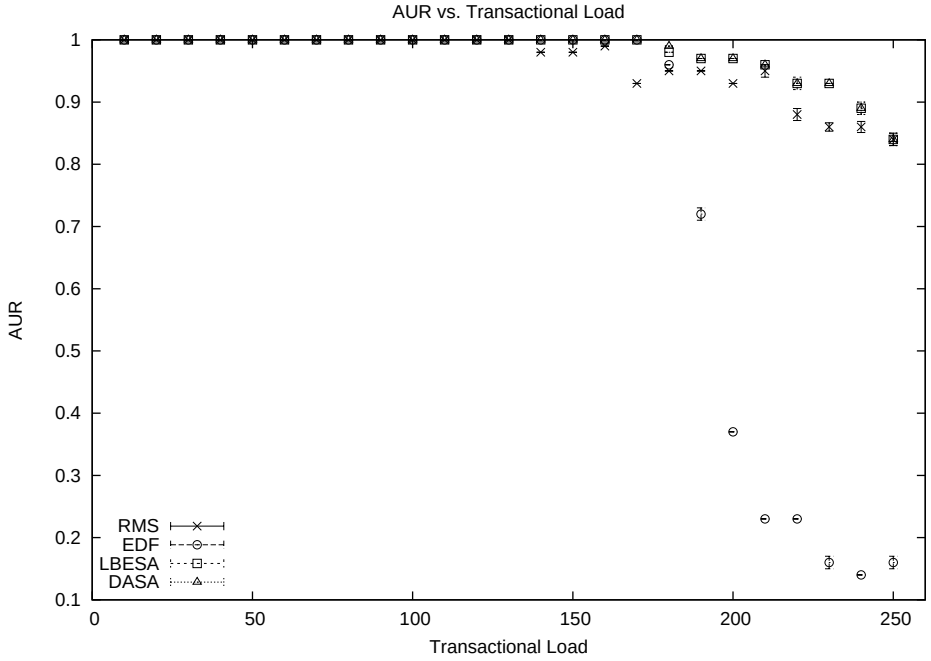

Figure 6.13: AUR vs. Utilization: T1, Inversely Proportional

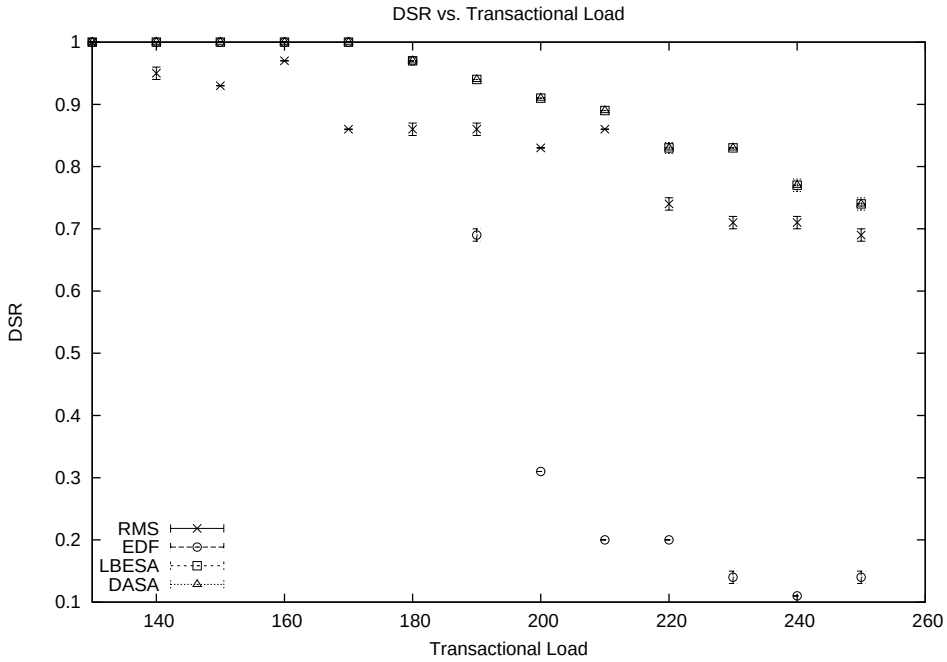

Figure 6.14: DSR vs. Utilization: T1, During overloads, Inversely Proportional

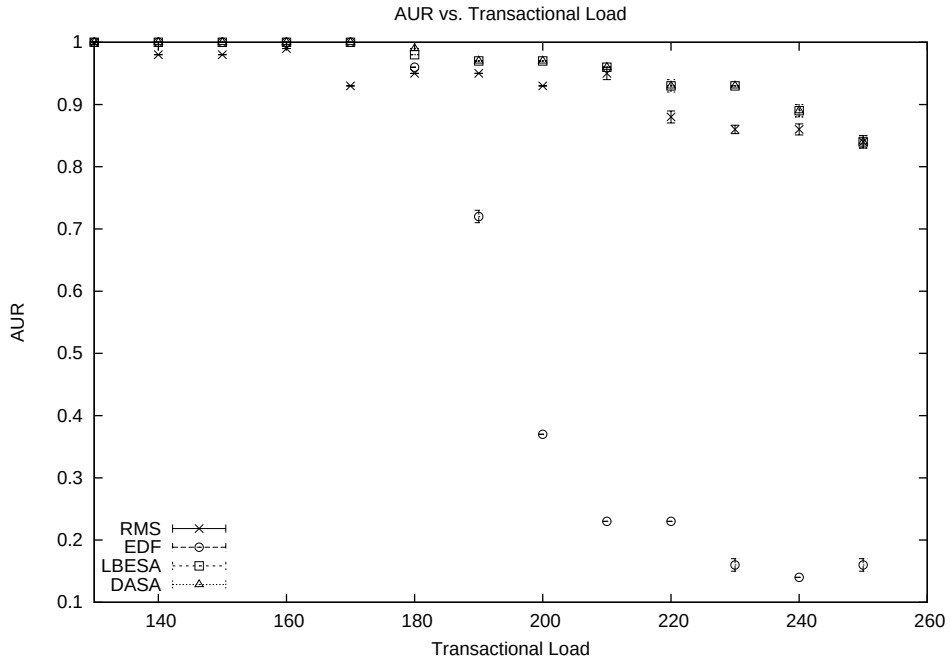

Figure 6.15: AUR vs. Utilization: T1, During overloads, Inversely Proportional

decision. This occurs because by favoring threads with small periods and/or deadlines, they are actually favoring threads with low utilities.

One thing that can be seen from all the results, regardless of the method for assigning utilities, is that while EDF is optimal during underloads, it is completely outclassed during overloads. For example, at its worst performance, EDF is out classes by DASA to a tune of 66%, 64% and 77% for the case of random, proportional and inverse proportional assignment of utilities respectively when AUR is used as the performance metric.

RMS, on the other hand, is quite competitive during overloads if the utilities are assigned in a specific manner. For example, it is out classed by DASA, at its worst performance, to the tune of 25%, 27% and 6.6% for the case of random, proportional and inverse proportional assignment of utilities respectively when AUR is used as the performance metric. Note, particularly, the number when the proportional method for assigning utilities is used – 6.6%.

After repeating several such experiments, we have come to the conclusion, that case two DASA/HUA outperforms all of the other algorithms. Therefore, it is this case two algorithm, in the form of HUA in order to handle the case where exception handlers are used, is chosen as the competitor to which we compare our collaborative algorithms.

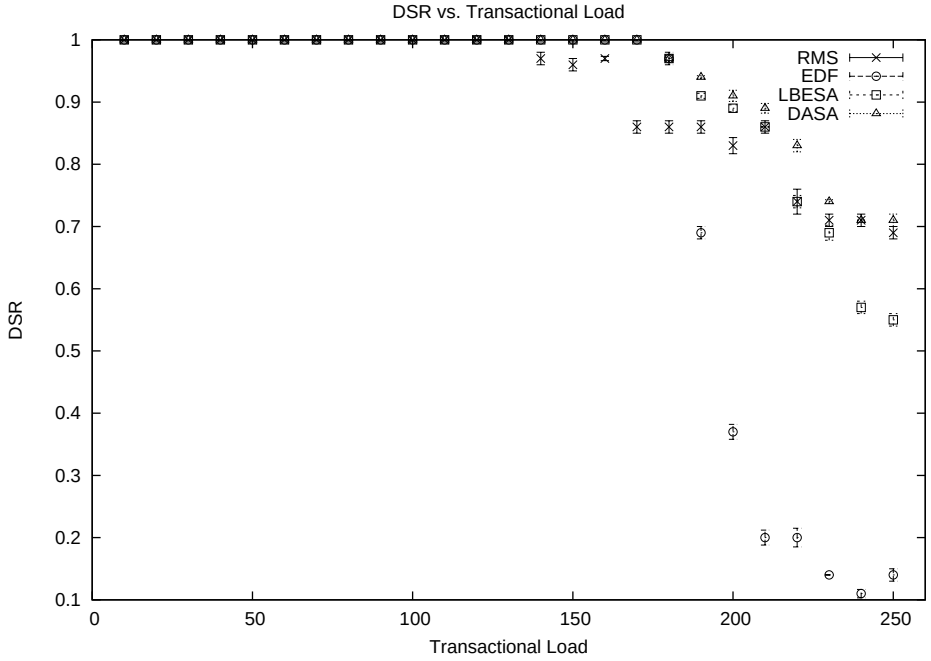

Figure 6.16: DSR vs. Utilization: T1, Directly Proportional

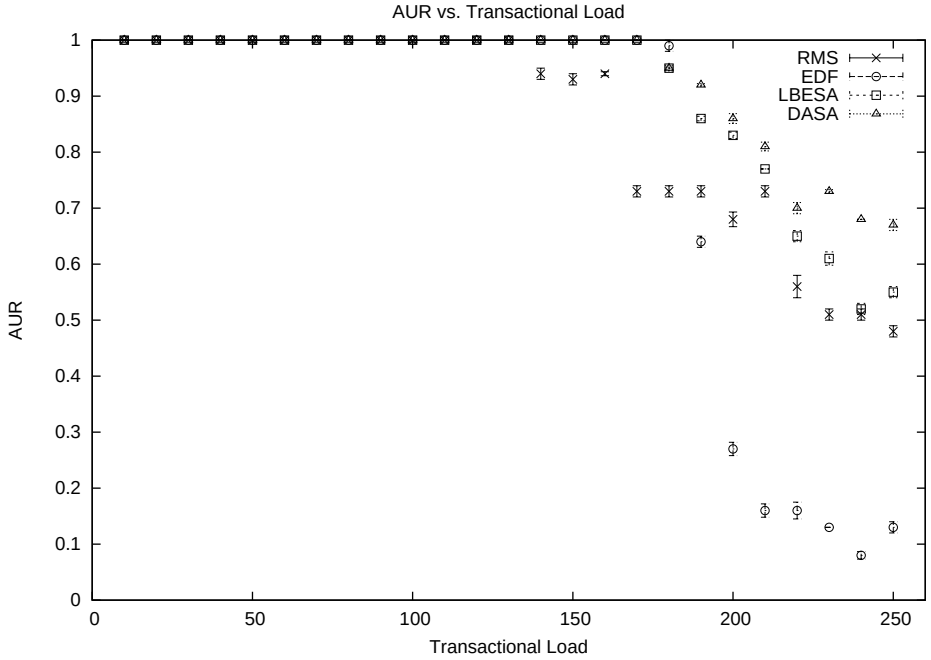

Figure 6.17: AUR vs. Utilization: T1, Directly Proportional

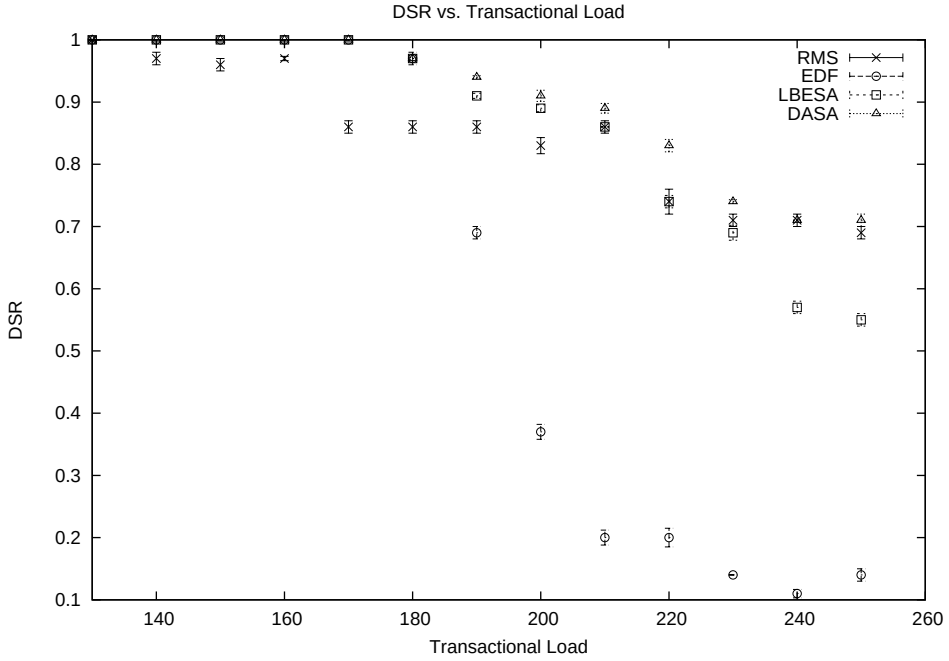

Figure 6.18: DSR vs. Utilization: T1, During overloads, Directly Proportional

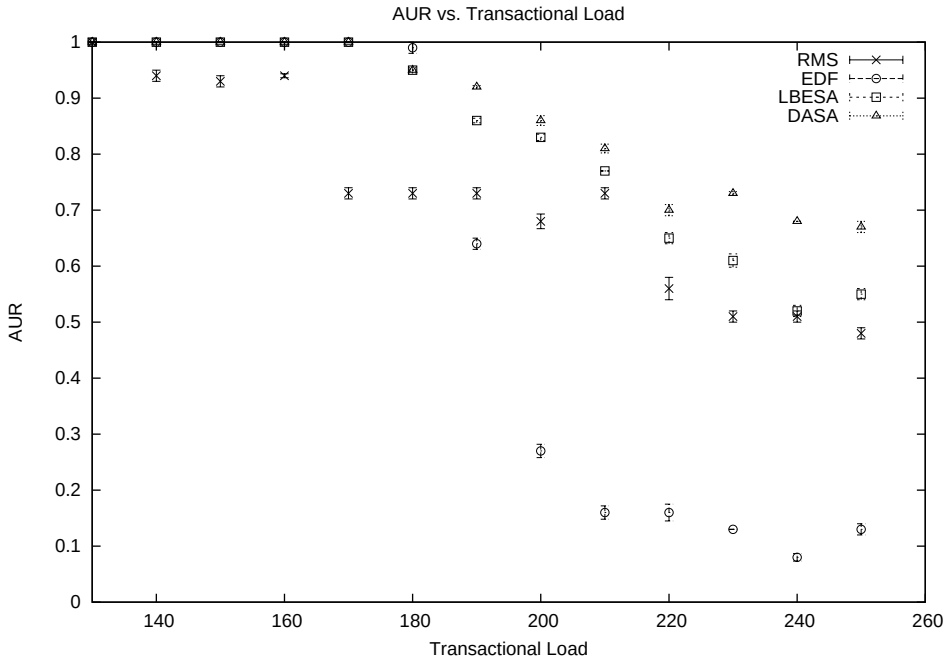

Figure 6.19: AUR vs. Utilization: T1, During overloads, Directly Proportional

Chapter 7

Experimental Results without Dependencies

7.1 Consensus-Based Collaborative Schedulers

Now that we have described the general approach taken, we will turn our attention to the consensus-based collaborative scheduling algorithms developed in this work. Below, we describe the behavior of CUA and ACUA on task sets of different parameters and three different methods for deadline decomposition – worst-case deadline decomposition, proportional slack and ultimate deadline decomposition. Note that the algorithm descriptions in the relevant chapters use the worst-case deadline decomposition method, but any deadline decomposition method mentioned below can be used. After we conduct a series of experiments using all three different methods for deadline decompositions, we scope out the ultimate deadline decomposition method since, while it highlights the differences between collaborative scheduling and independent scheduling more than any other decomposition method, it leads to more deadline misses since it allows earlier sections in a DT to assume that they have more slack than they really do.

In the experiments we conduct, five DTs are considered. Each thread consists of two real-time sections. The first section starts on one node and then an invocation is made to another node. Once the remote invocation returns, the entire DT terminates. The periods, maximum utility of the threads and the execution times of the sections are varied in each of the experiments conducted. In all experiments, we assume that relative deadline is equal to period and that all tasks start with zero offset.

To highlight the properties of collaborative scheduling, two different classes, class A and class B, of task sets are used. Class A task sets are designed to ensure that different local schedules are constructed on each of the nodes. Toward that end, we number the DTs with IDs in the range $0 \cdots 4$. The sections belonging to DTs with even IDs have a WCET ratio

of 1:3 (i.e., the execution time of the second section is three times that of the first section) while sections belonging to DTs with odd IDs have a WCET ratio of 3:1.

Class B task sets have sections whose ratio of WCET is 1:1. This should mask the inherent differences in the local schedules produced on different nodes, leaving only residual differences that are a result of different arrival patterns on different nodes. The intuition behind this choice is to see how collaborative scheduling fares when there is no inherent differences that occur.

We assign the same arrival time to both sections of a DT initially i.e., when a DT arrives into the system, that arrival time is recorded as the arrival time of both of its sections in the scheduler. Of course, once a section physically arrives at a node, it updates this value to the time at which it arrived at the node. However, prior to the arrival of a section on a node, the arrival time of the entire DT is used as the arrival time of the section when making scheduling decisions. In addition, when a section arrives at a certain node and other sections have not yet arrived, their arrival time is updated to the arrival time of the section that has just arrived. This is done to avoid unnecessary pessimism in estimating the arrival times on the one hand and the high overhead of getting more accurate estimates of arrival times on the other.

Two machines were used in these experiment, both of them have the 2.6.24.7 version of the Linux kernel, with the rt27 PREEMPT patch applied and our custom extensions to the kernel included. The first node has a 2.16GHz T3400 Intel Dual-Core CPU with 2GB of memory. The HPET is selected as the timesource on this machine by the Linux kernel. The second node has a 2.60GHz E5300 Intel Dual-Core CPU and 1GB of memory. The TSC is selected as the timesource on this machine by the Linux kernel. The nodes are connected via 100Mb/s Ethernet through a "OfficeConnect ADSL Wireless 11g Firewall Router" from 3Com.

We record the average and 95% confidence interval of the DSR and AUR for 100 runs of each setting for each experiment. These values are plotted against the transactional load. We define the transactional load as the ratio between the total execution times of the sections of a transaction and its period. Note that transactional load does *not* correspond to the load experienced by the processors in the system. Rather, transactional load is defined as the ratio between the sum of the execution times of all the sections in the DT and its end-to-end period.

The periods of the tasks are generated from a uniform distribution with parameters 10000000μsecs and 100000μsecs, while the utilities are generated from a uniform distribution with parameters 1 and 100. The actually values obtained from these uniform distributions for each experiment is summarized in a table for each experiment we conducted.

Table 7.1 provides a key for the captions we use to describe each of the plots in this chapter and in Chapter 8.

Table 7.1: Caption Keys

Key	Definition
T<#>	Thread set number #
OL	During overloads
HL	Handler load
WD	Worst-case deadline decomposition
PS	Proportional slack
UD	Ultimate deadline
CS	Critical section length
DD	Distributed dependencies
NL	Number of locks

Table 7.2: Task Set Parameters for Consensus-based Scheduling

	Period (μsecs)	Utility
T1	1524000	11
T2	977000	12
T3	2135000	17
T4	1337000	25
T5	2533000	35

7.1.1 Worst Case Deadline Assignment

In this section, we use the worst case deadline assignment method to decompose the end-to-end deadline among the sections of a DT. In this method, the deadline of a section is equal to the deadline of its successor section minus the execution time of the successor section. The deadline of the last section, i.e., the section that has no successor, is equal to the end-to-end deadline. The first task set we use for this experiment is depicted in table 7.2. The WCETs are chosen according to the Class A task set approach. In this experiment, we consider negligible overhead failure handlers. Therefore, Definition 2 in Chapter 3 is equivalent to normal underload.

Figures 7.1, 7.2, 7.3 and 7.4 depict the result of this experiment. As can be seen, ACUA is better than its two competitors. Specifically, the algorithms start out accruing equal utility until the 200% transactional load point. Thereafter, HUA diverges from the rest of the algorithms by accruing less utility than they do. The largest difference between HUA and ACUA, 13%, is at the 250% transactional load point. CUA and ACUA continue accruing about the same utility until the 240% transactional load point when ACUA begins to outperform CUA. The largest difference between ACUA and CUA in this experiment is 6% at the 270% transactional load point.

HUA, CUA and ACUA are designed to actively reserve processor time for exception handlers – a section is not allowed to execute unless its handler can be successfully executed. In the

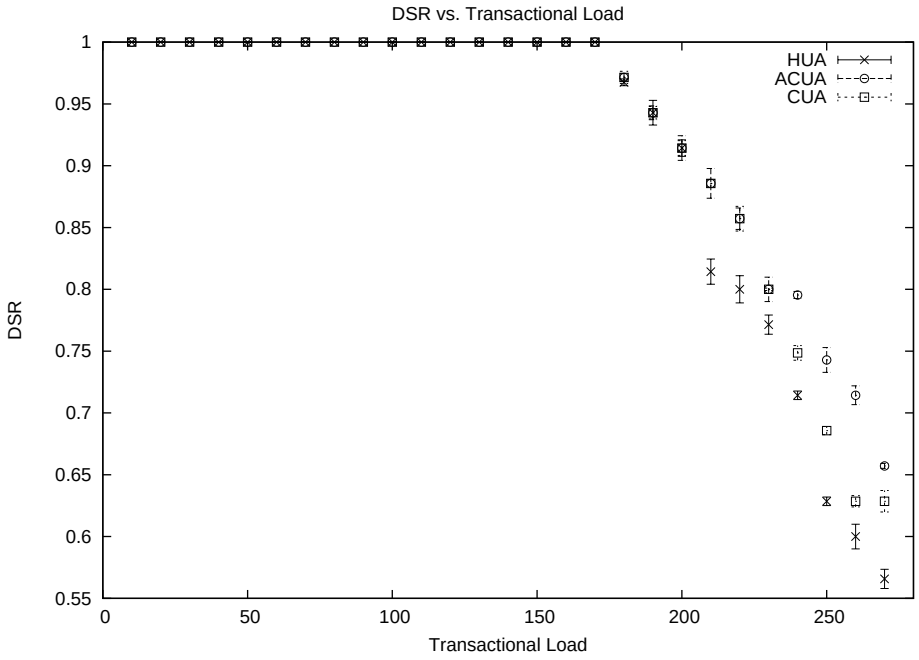

Figure 7.1: DSR vs. Transactional Load (WD, T1)

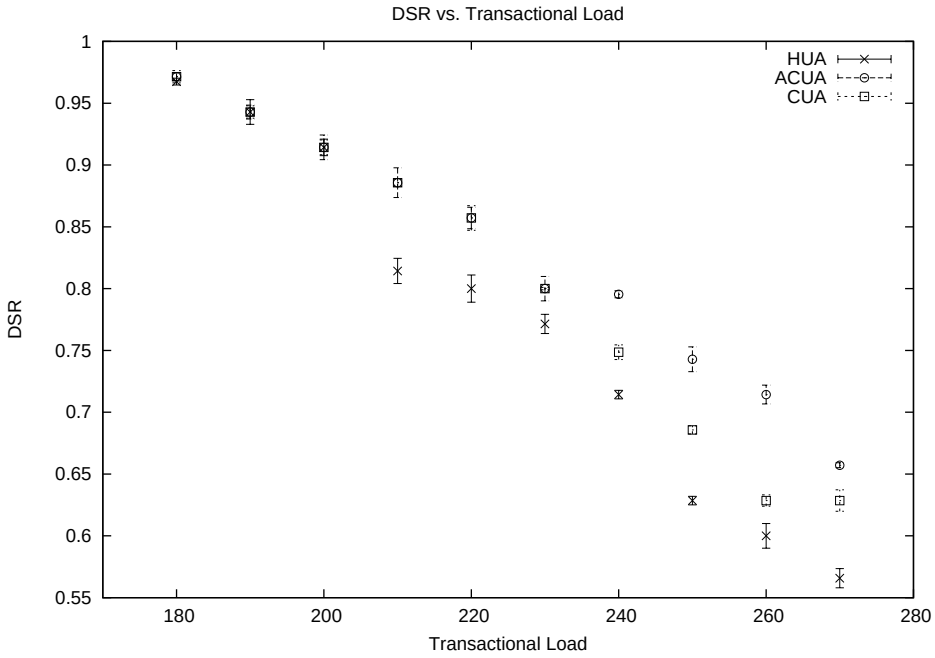

Figure 7.2: DSR vs. Transactional Load (OL, WD, T1)

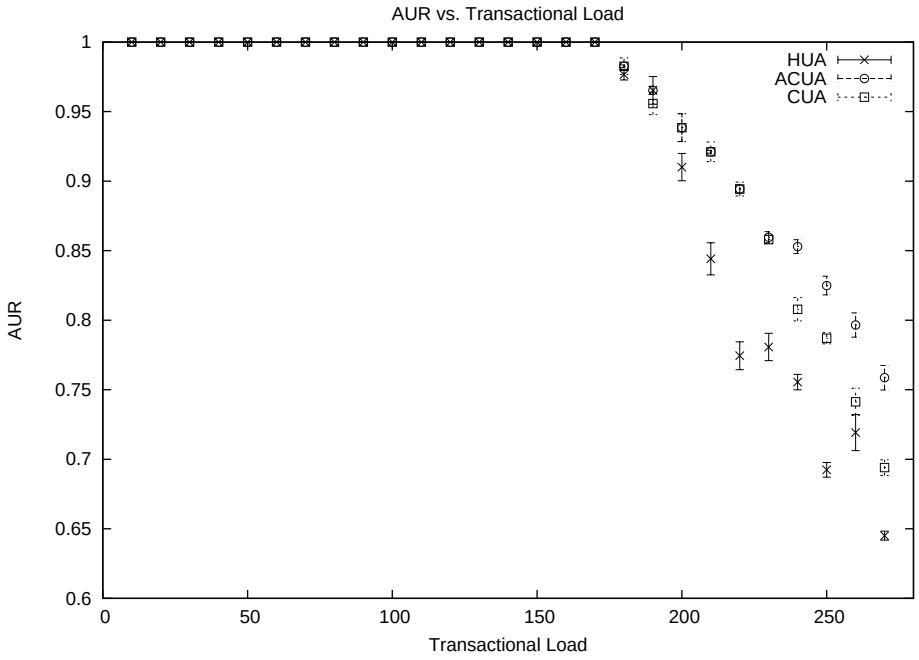

Figure 7.3: AUR vs. Transactional Load, (WD, T1)

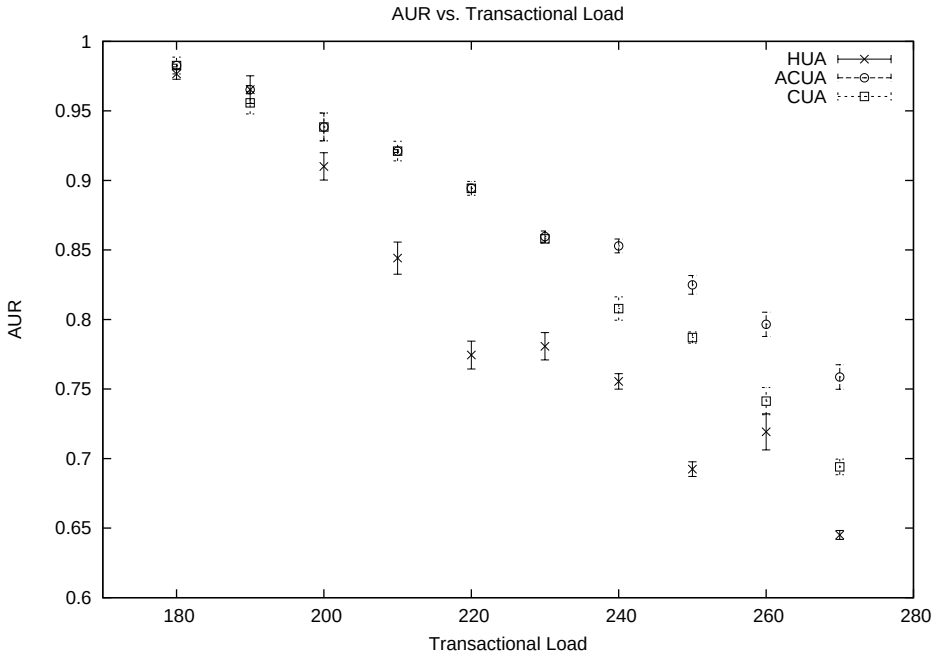

Figure 7.4: AUR vs. Transactional Load (OL, WD, T1)

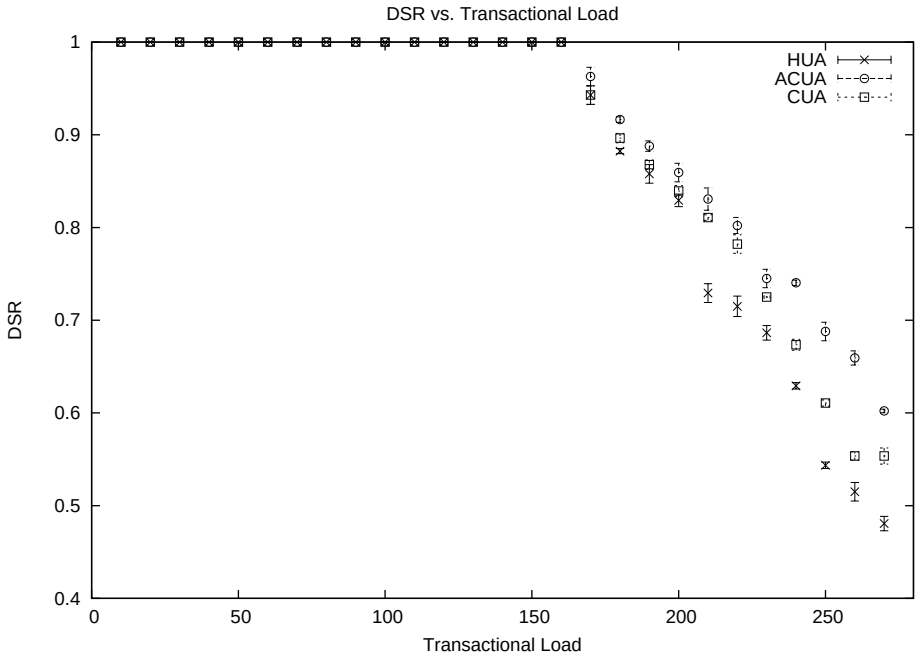

Figure 7.5: DSR vs. Transactional Load (WD, T1, HL=0.1)

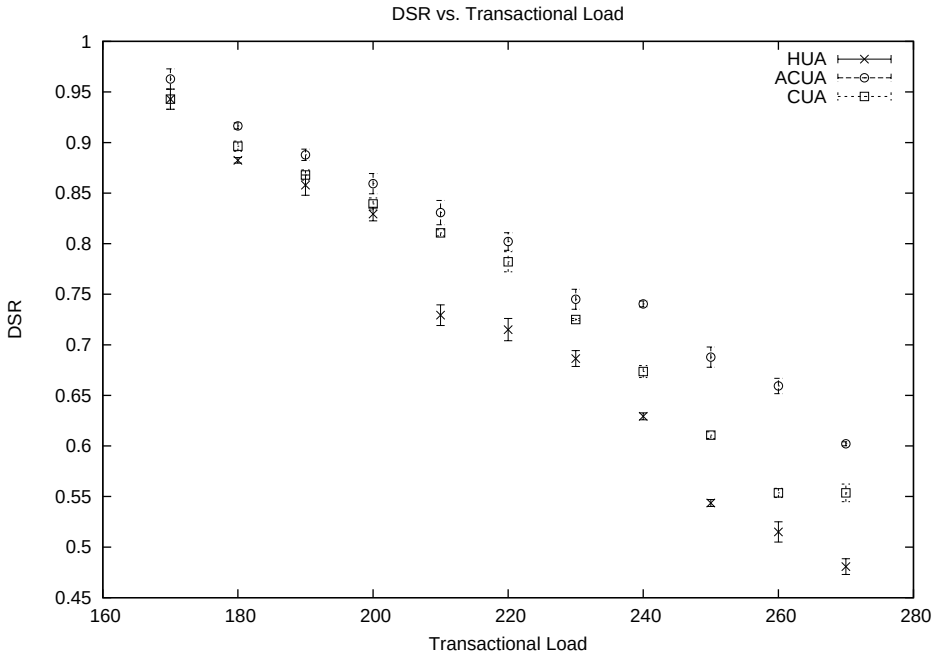

Figure 7.6: DSR vs. Transactional Load (OL, WD, T1, HL=0.1)

first version of the first experiment we conducted, we considered handlers with negligible overhead (they essentially involved a call to **pthread_exit** only). In the second version of the first experiment, we burn CPU time in the exception handlers. We chose the execution time of the exception handlers such that their average offered load is equal to 10% of processor time. This leaves only 90% of processor time for actually executing the sections. In this scenario, the Handler Underload State (see Definition 2 in Chapter 3) is equal to 0.9.

This has two consequences, first the tasks will begin missing deadlines when the load on any given processor exceeds 0.9, not 1.0, this corresponds to a different transactional load than for the case of negligible overhead handlers (170% instead of 180% for this task set). Second, this will force the algorithms to abort more tasks since the feasibility region for the system shrinks. Because more tasks are aborted, this offers greater opportunity for algorithms that choose the "right" set of tasks to outperform others – i.e., it favors ACUA over the other algorithms. The results for this experiment are shown in Figures 7.5, 7.6, 7.7 and 7.8. In this experiment, ACUA has a maximum lead over HUA of about 17% at the 250% transactional load point and a maximum lead over CUA of about 10% at the 270% transactional load point. Also, notice that the algorithms first begin to miss deadlines at the 170% transactional load point instead of the 180% transactional load point. This is due to the fact that processing time is reserved for the termination handlers and these handlers, unlike in the first version of this experiment, do not have zero overhead. As can be seen, ACUA gained a larger lead over its competitors since the addition of non-zero overhead termination handlers force the system to abort more tasks, thus highlighting the performance benefits of algorithms that make the "right" choices in these scenarios.

The task set used for the second experiment performed using this method of deadline decomposition is depicted in Table 6.1. Again, the first iteration of this experiment uses negligible overhead failure handlers. Figures 7.9, 7.10, 7.11 and 7.12 depict the result of this experiment. Again, the Class A WCET assignment method is used in this experiment. ACUA outperforms the other two algorithms at virtually all data points when considering AUR. ACUA outperforms CUA by a maximum value of 10% at the 240% transactional load point. The largest difference between ACUA and HUA is about 9% at the 240% transactional load point.

As before, the second iteration of this experiment uses failure handlers whose offered load is 0.1. The results for this experiment are depicted in Figures 7.13, 7.14, 7.15 and 7.16. As for the first experiment, the introduction of non-zero overhead handlers tips the scale towards ACUA as it gets to showcase its superior selection of which tasks to keep and which tasks to reject. Also, note that the system now begins to miss deadlines at the 160% transactional load point instead of the 180% transactional load point. The maximum difference between ACUA and its competitors occurs at the 240% transactional load point, with ACUA accruing about 11% more utility than HUA and 10% more utility that CUA.

For the next experiment, whose results are depicted in Figures 7.17, 7.18, 7.19 and 7.20, we consider the task set described in Table 6.1. However, we assign the WCET of its sections

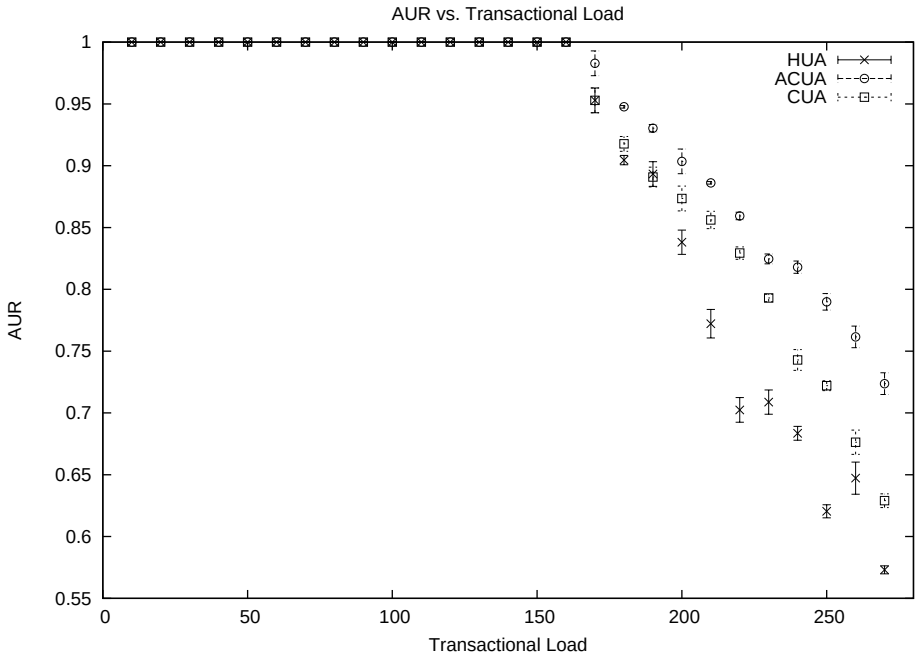

Figure 7.7: AUR vs. Transactional Load, (WD, T1, HL=0.1)

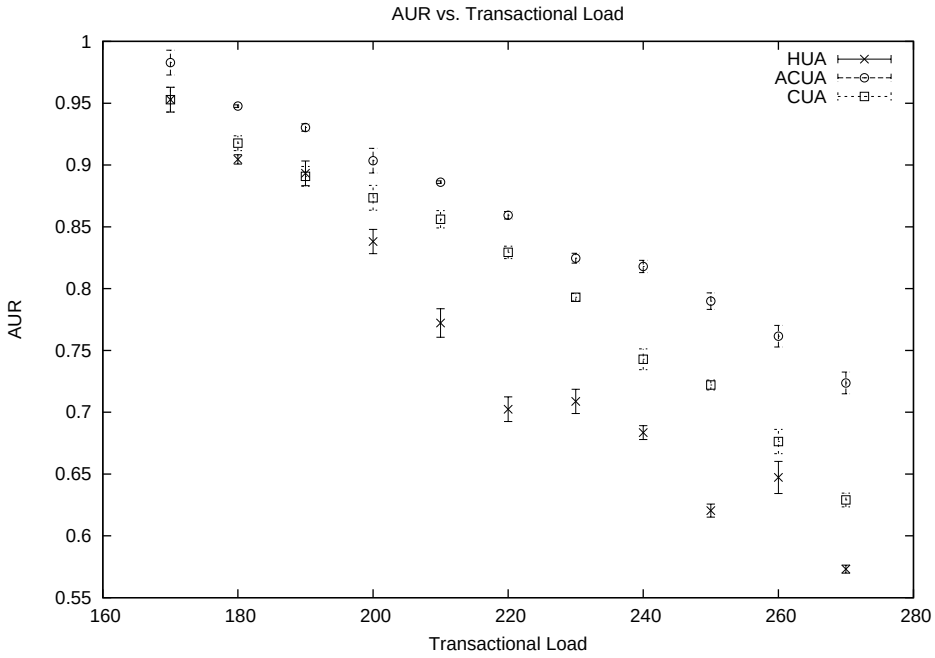

Figure 7.8: AUR vs. Transactional Load (OL, WD, T1, HL=0.1)

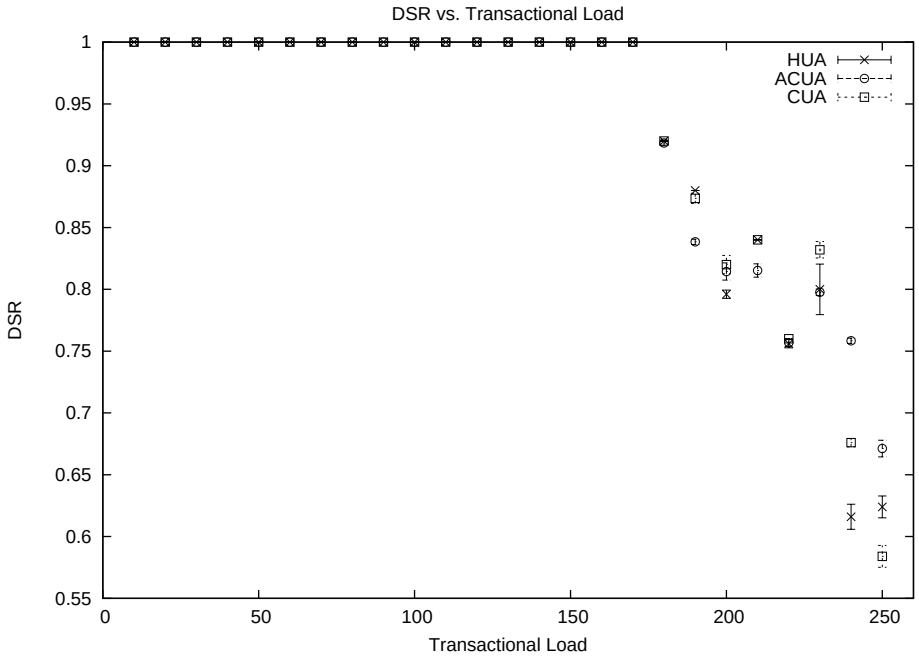

Figure 7.9: DSR vs. Transactional Load (WD, T2)

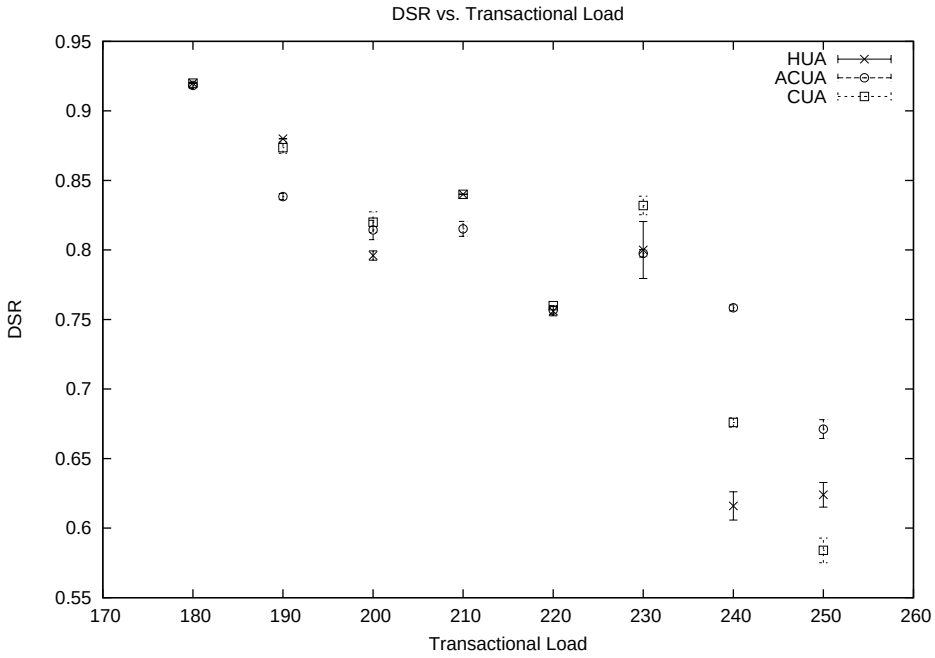

Figure 7.10: DSR vs. Transactional Load (OL, WD, T2)

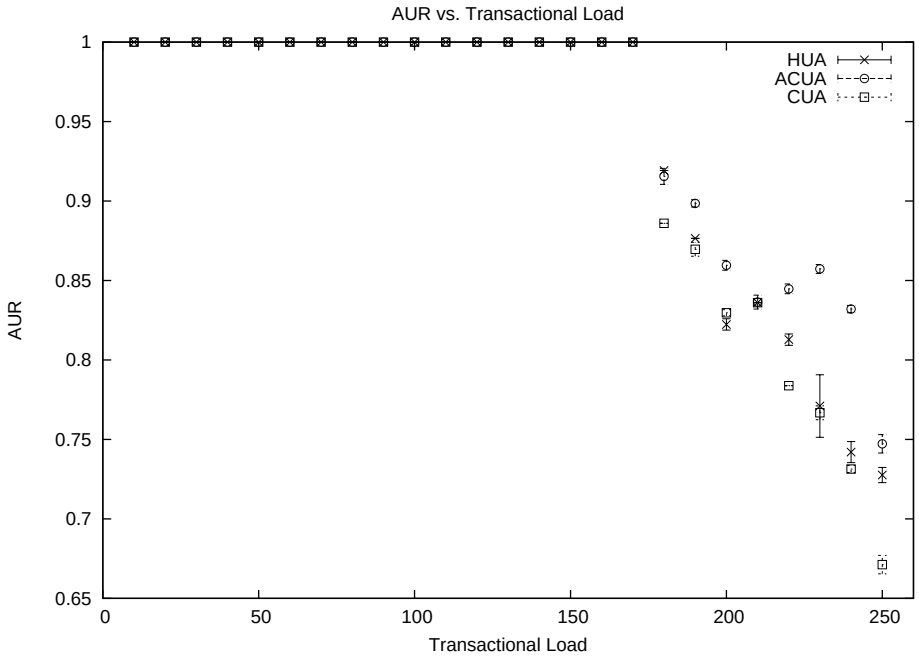

Figure 7.11: AUR vs. Transactional Load (WD, T2)

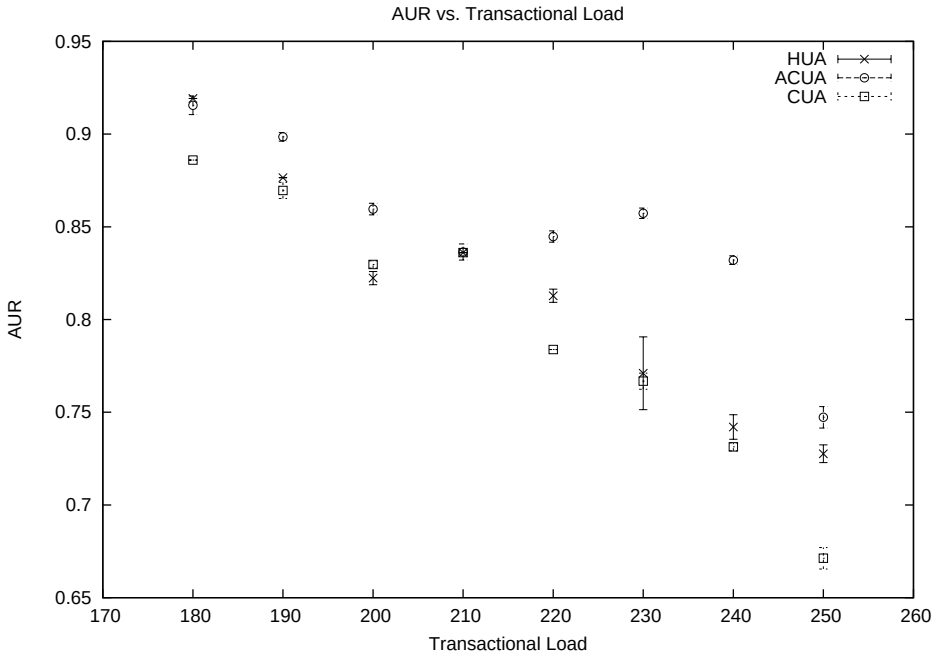

Figure 7.12: AUR vs. Transactional Load (OL, WD, T2)

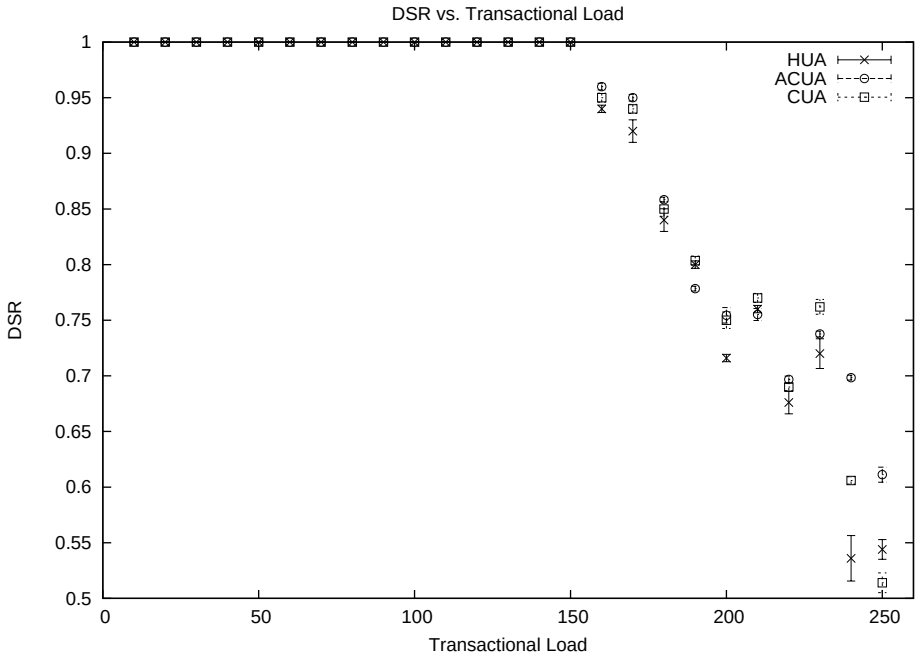

Figure 7.13: DSR vs. Transactional Load (WD, T2, HL=0.1)

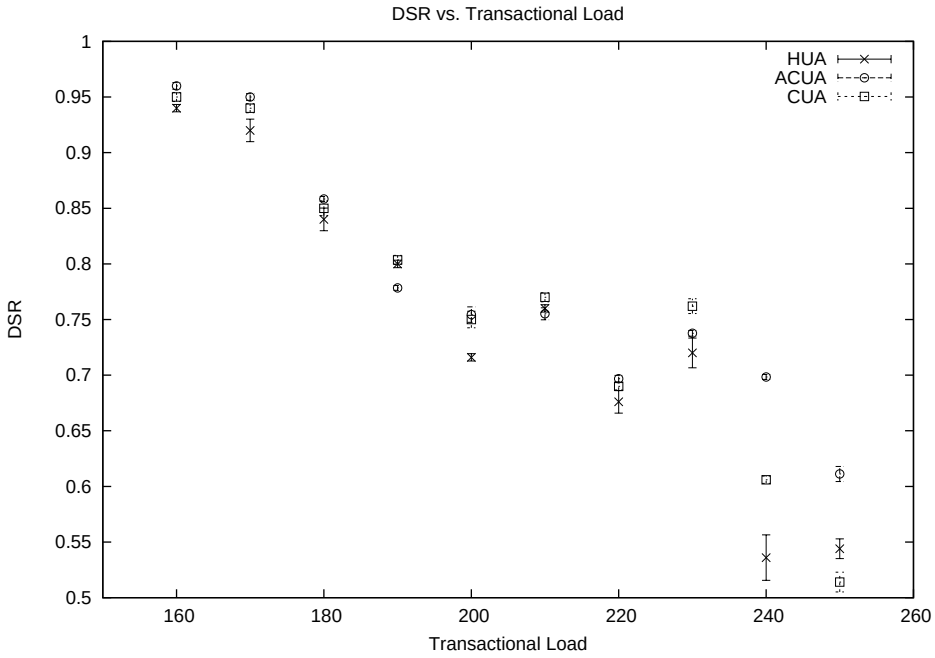

Figure 7.14: DSR vs. Transactional Load (OL, WD, T2, HL=0.1)

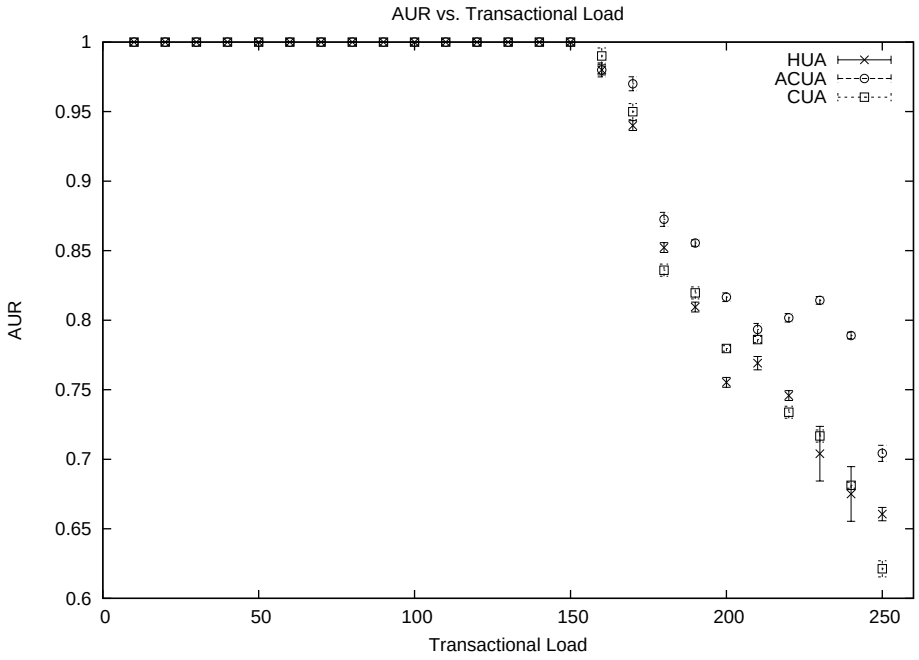

Figure 7.15: AUR vs. Transactional Load (WD, T2, HL=0.1)

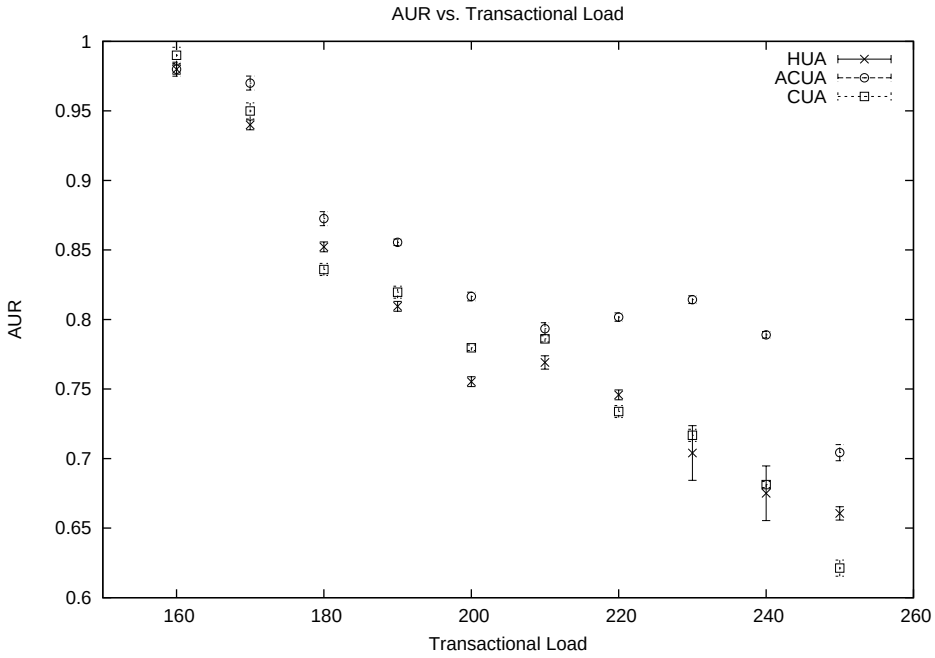

Figure 7.16: AUR vs. Transactional Load (OL, WD, T2, HL=0.1)

using the Class B method. The WCET of the sections in Class B thread sets have a 1:1 ratio. The rationale behind choosing this parameter set is that since the execution times of both sections of the DTs are equal, the scheduling decisions of the different nodes hosting them is not *inherently* different.

While the scheduling decisions are not necessarily *inherently* different, they *can* be different if different threads are at different stages of execution thus resulting in different PUDs at different nodes. This is borne out by the fact that ACUA still accrues more utility than the two other at the 190%, 210% and 250% transactional load points. However, unlike for the other task sets, ACUA is not *consistently* better than the other two algorithms. This occurs because there is no inherent benefit for collaboration for this particular task set, beyond the ability to terminate a DT early if its subsequent sections cannot meet their deadlines, and therefore the higher overheads of collaboration do not pay off. Note that the trend in the results are not linear. This is due to the fact that transactional load does not correspond to the instantaneous load on processors.

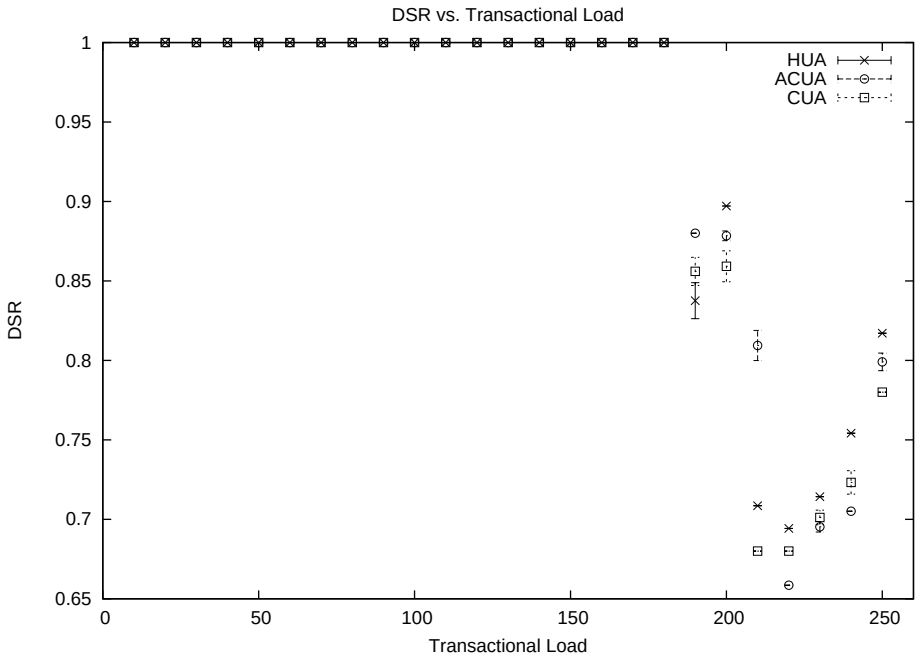

Figure 7.17: DSR vs. Transactional Load (WD, T3)

This behavior is seen for other task sets using the Class B WCET assignment method. We do not reproduce these plots to avoid clutter. We also do not reproduce the results for this particular thread set in the other deadline decomposition methods.

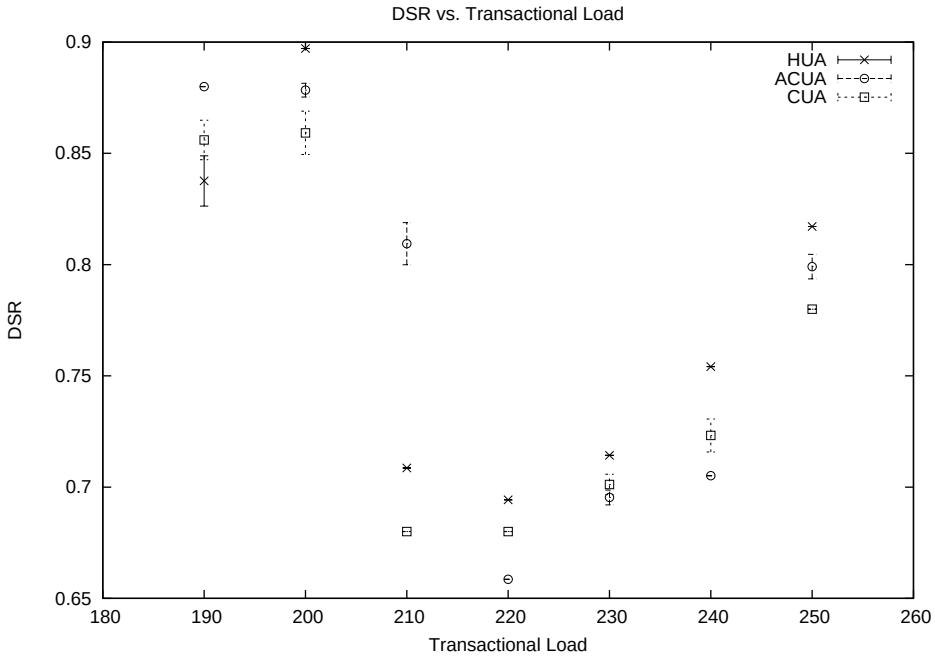

Figure 7.18: DSR vs. Transactional Load (WD, UD, T3)

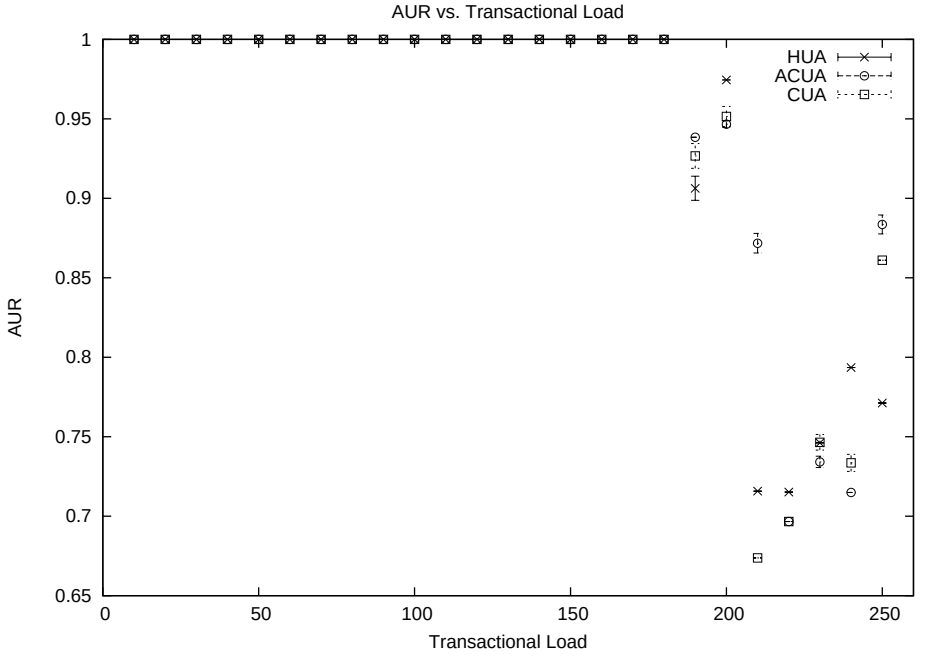

Figure 7.19: AUR vs. Transactional Load (WD, T3)

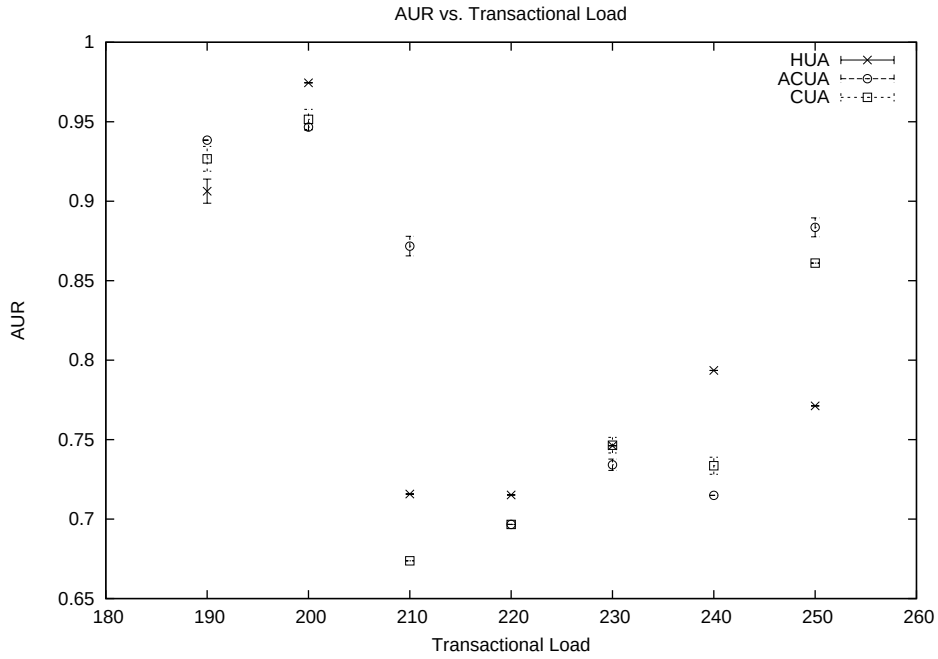

Figure 7.20: AUR vs. Transactional Load (OL, WD, T3)

7.1.2 Proportional Slack Assignment

In this section, we describe the second approach we took regarding deadline decomposition. The section arrival times are still estimated to be equal to the arrival time of the entire thread until they actually arrive at their respective nodes, but the end-to-end deadline of the DT is decomposed among the sections in proportion to their execution times. Specifically, we compute the slack time of the DT i as follows:

$$S_i = D_i - \sum_{\forall j} WCET_j \tag{7.1}$$

We then divide this slack time in proportion to the execution time of each section. For example, if the ratio of the execution times of a DT with two sections is 1:3, the slack is divided into $S_i/4$ for the first section, while it is $S_i \times \frac{3}{4}$ for the second section. The rationale for this approach is to obtain a fairer distribution of slack among the sections of a DT. In [130], this approach is shown to have, in practice, better results than other approaches for deadline decomposition when case two scheduling with RMS is used.

For the first experiment performed using this method of deadline decomposition, we use the task set described in Table 7.2. Figures 7.21, 7.22, 7.23 and 7.24 depict the results. in terms of AUR, ACUA outperforms the other two competitors. ACUA has a maximum lead of 14% over HUA at the 230% transactional load point and a maximum lead of 6% over CUA at

the 190% transactional load point.

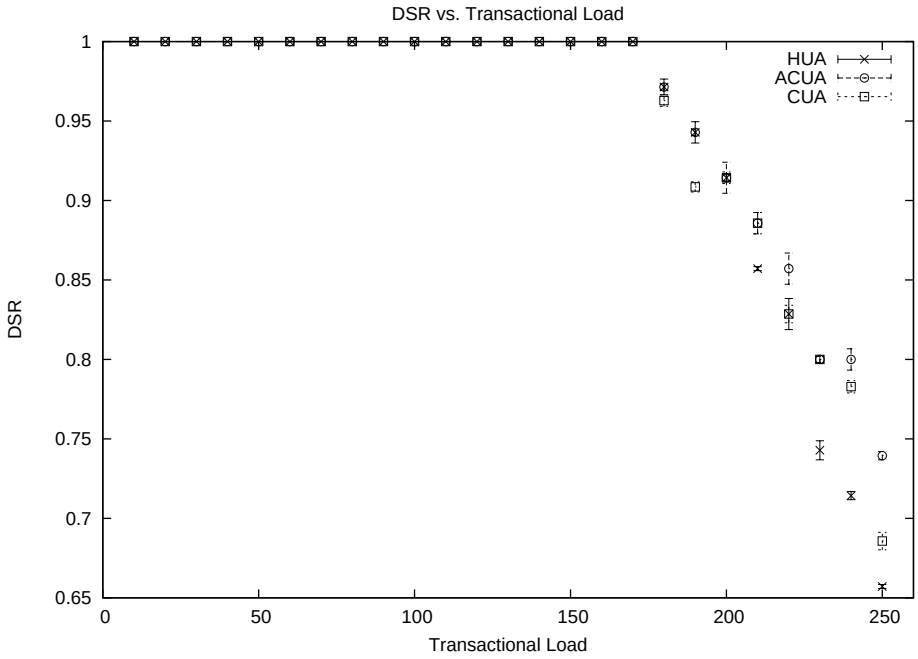

Figure 7.21: DSR vs. Transactional Load (PS, T1)

In the second iteration of the first experiment, we consider failure handlers with 0.1 offered load. The results of this experiment is presented in Figures 7.25, 7.26, 7.27 and 7.28. In this experiment, the addition of failure handlers with non-negligible overhead forces the system to shed tasks earlier on and to a greater extent. This, in turn, allows ACUA to show-case its better best-effort property. This can be seen in the results, specifically, the maximum lead of ACUA over HUA is now 17% while the maximum lead of ACUA over CUA is now 7%. Also, as in earlier experiments, the transactional load at which deadlines are first missed is pushed back because the handlers need to be accommodated in the schedule. For the rest of this section and in Section 7.1.3 we do not perform this experiment again, since the results follow the same trend as shown in this section.

The second task set used in this experiment is depicted in Table 6.1. The results are depicted in Figures 7.29, 7.30, 7.31 and 7.32. As can be seen, ACUA outperforms all other algorithms at all points expect at the 190% transactional load point. We discuss the cause of this interesting behavior later in this section. For this task set, ACUA has a maximum lead over CUA of 11% at the 220% transactional load point, and a maximum lead over HUA of about 6% at the 210% transactional load point.

One thing that is particularly obvious in the results for this task set, is that although ACUA mostly underperforms when considering the DSR metric (see Figure 7.30) it is better than, or at least as good as, the other algorithms when considering the AUR metric (see Figure 7.32).

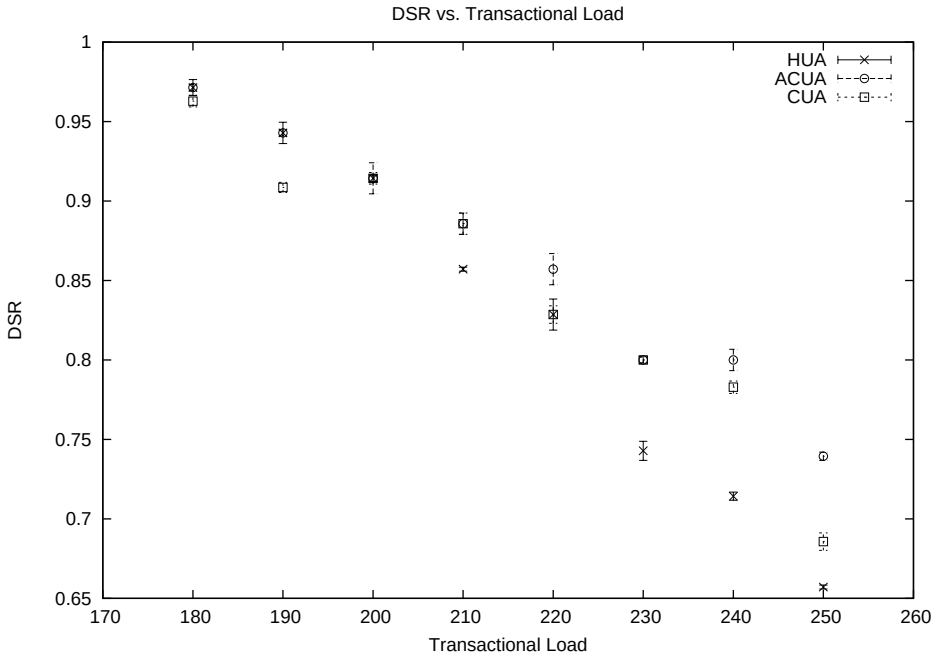

Figure 7.22: DSR vs. Transactional Load (OL, PS, T1)

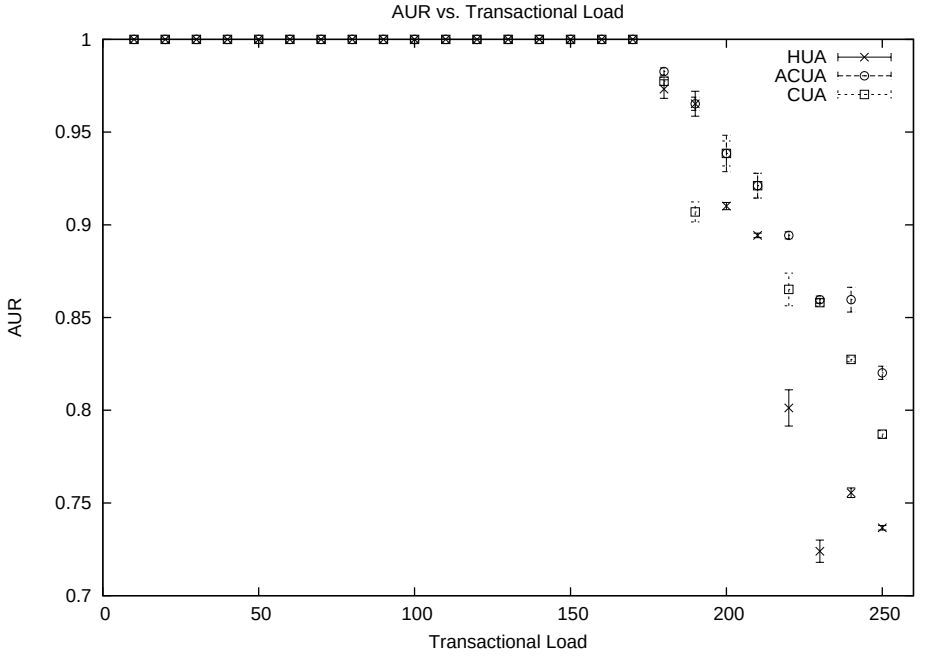

Figure 7.23: AUR vs. Transactional Load, (PS, T1)

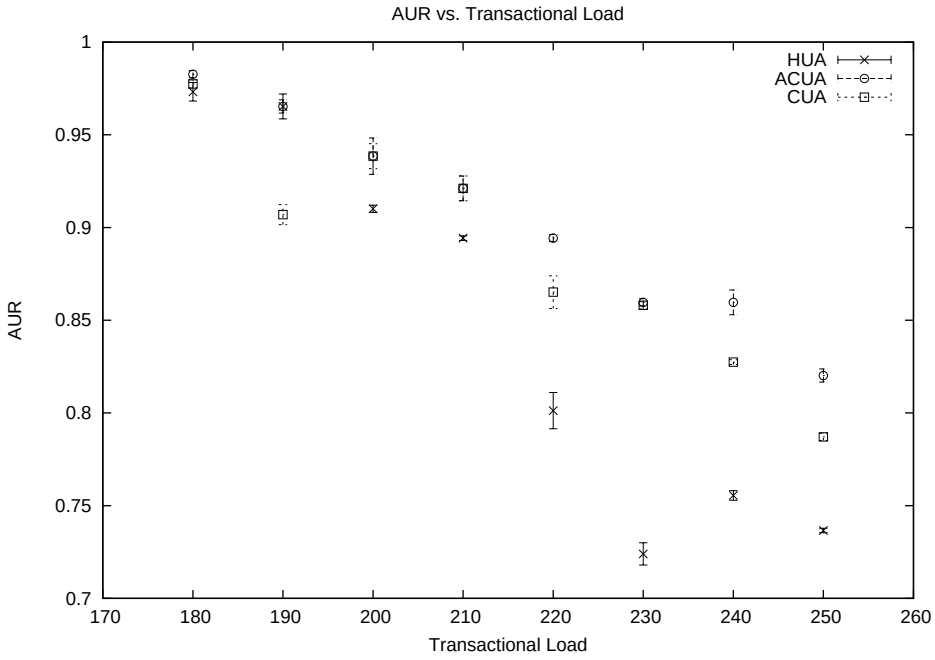

Figure 7.24: AUR vs. Transactional Load (OL, PS, T1)

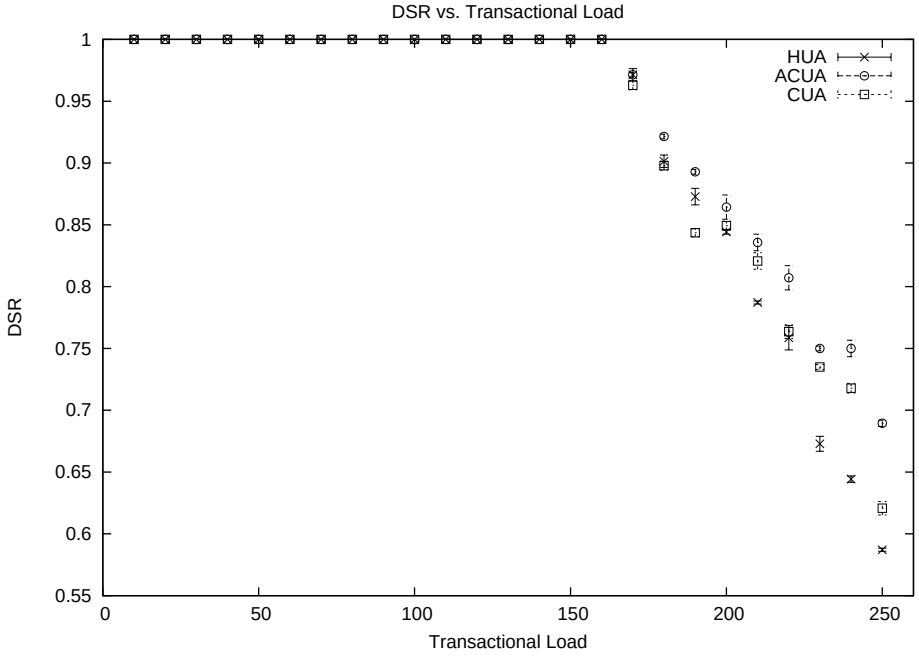

Figure 7.25: DSR vs. Transactional Load (PS, T1, HL=0.1)

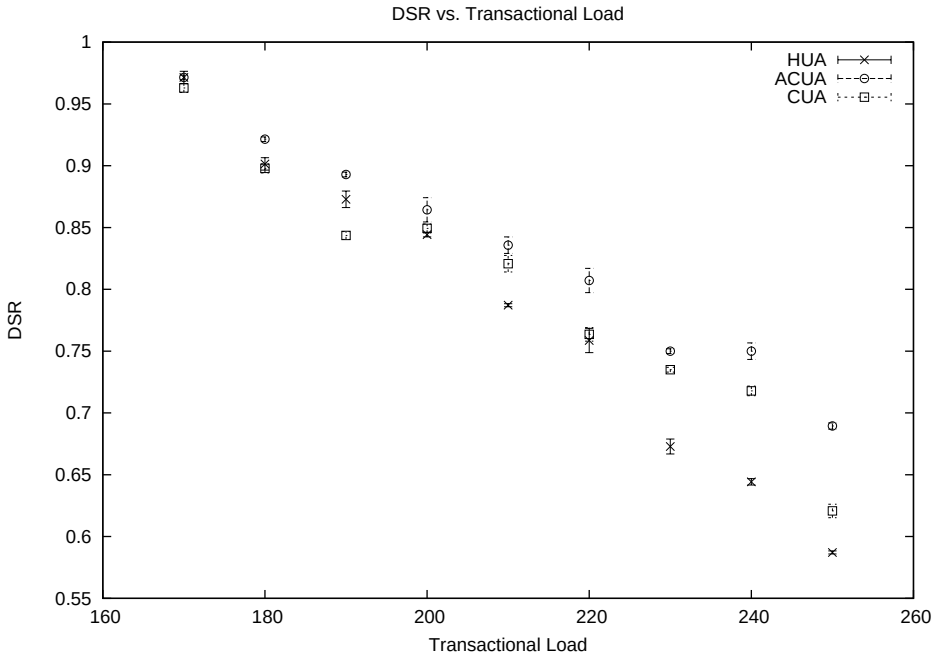

Figure 7.26: DSR vs. Transactional Load (OL, PS, T1, HL=0.1)

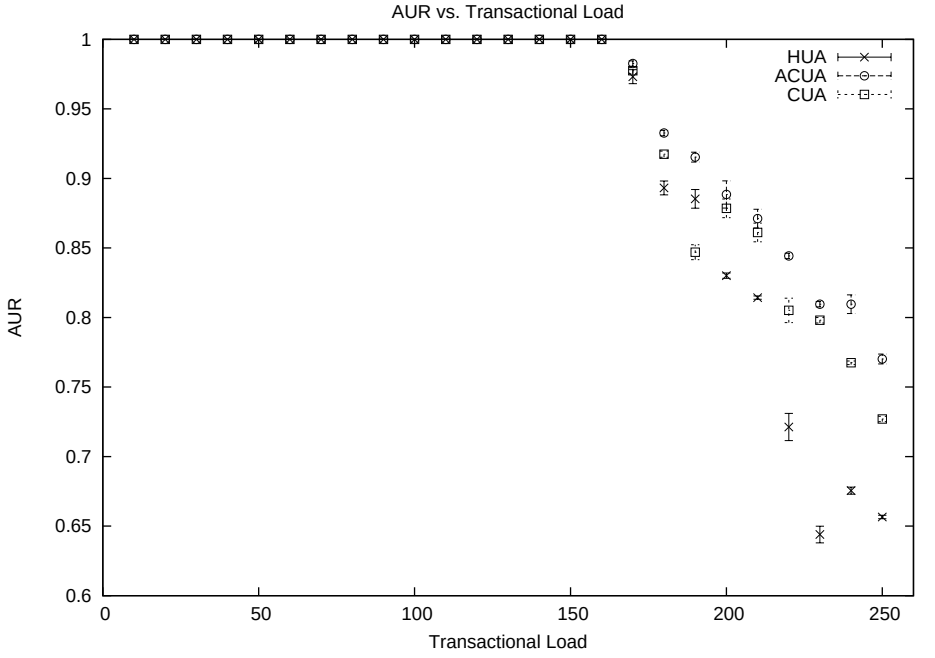

Figure 7.27: AUR vs. Transactional Load, (PS, T1, HL=0.1)

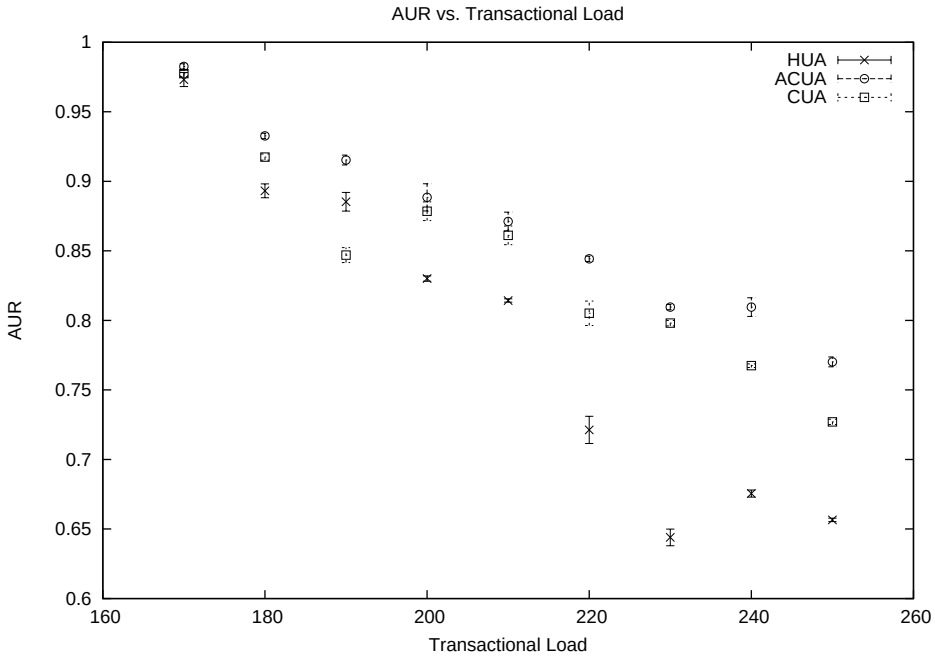

Figure 7.28: AUR vs. Transactional Load (OL, PS, T1, HL=0.1)

Again, this is due to the fact that ACUA chooses the "right" set of tasks to allow to execute and eliminates the tasks that would contribute the least amount to the utility accrued to the system. Thus it trades off losses in the number of tasks that actually meet their deadline with allowing high utility tasks to complete at the expense of others.

One data point in the plots that is particularly interesting is the 190% transactional load point in both Figures 7.30 and 7.32. At this data point, ACUA is outperformed by both CUA and HUA when DSR is the performance metric and by HUA when AUR is the performance metric.

The question now arises: Is this scenario caused by the higher overhead of ACUA or is there something inherent about this data point that causes this behavior? In order to further understand the scheduling decisions that led to the numbers for this data point we refer to the scheduler logs that were created during the scheduling.

In particular, we note that the schedule constructed by ACUA differs from the schedule constructed for CUA and HUA in two points. We now reproduce these two points below and discuss the difference between the schedule created by ACUA and the schedule created by the other algorithms. Both of these points originate on node two, therefore we shall only show the schedule of this node in the discourse that follows.

As noted earlier, the arrival of a new thread into the system is a distributed scheduling event for both CUA and ACUA, we now zoom in on the arrival of two different threads and the schedule that is created when these threads arrive. We shall call these two threads, for ease

Table 7.3: Schedule on Arrival of Thread1 for ACUA

PID	Start	WCET	Deadline	End
365	0 sec, 366 ms	374 ms	1 sec, 363 ms	0 sec, 740 ms
368	0 sec, 740 ms	428 ms	1 sec, 363 ms	1 sec, 168 ms
354	1 sec, 168 ms	1037ms	2 sec, 363 ms	2 sec, 205 ms
369	2 sec, 205 ms	218 ms	2 sec, 363 ms	2 sec, 423 ms
366	2 sec, 423 ms	285 ms	4 sec, 363 ms	2 sec, 708 ms

of reference, Thread1 and Thread2.

When Thread1 arrives, the schedule in Table 7.3 is constructed. In Table 7.3, we use the last three decimal digits of the 64bit DT id to represent threads to reduce unnecessary clutter in the table – we also state the timing information to the nearest millisecond to reduce clutter. In addition, we normalize the seconds parts of the times present in the log. Specifically, we set the seconds field of the earliest time in the log to zero and normalize all the other time values based on this. Again, this is to reduce clutter in the table and make it more understandable. The schedule depicted occurs when attempting to schedule the task with the lowest PUD in the system, thread 369 (we will discuss shortly why this is the task with the lowest PUD at this point).

As can be seen, the schedule is not feasible. The thread that makes the schedule infeasible is highlighted in green in Table 7.3. Thus, thread 369 is removed from the schedule and included in the reject set of node two. In CUA, thread 354 is considered the thread with the lowest PUD at this point and it is removed instead of thread 369. First, let us see why threads 354 is chosen as the thread with the lowest PUD in CUA while thread 369 is chosen as the thread with the lowest PUD in ACUA.

The logs indicate that thread 369 has two sections, section one has a WCET of 656 ms and section two has a WCET of 218 ms and that its maximum utility is 4. Thread 354 also has two sections, the first of which has a WCET of 475 ms and the second of which has a WCET of 1425 ms. The log files indicate that thread 369 has just arrived into the system, and so its sections have not had the chance to execute yet, while thread 354 is in the middle of executing its second section after its first section has finished executing. Specifically, the second section of thread 354 now has 1037 ms of its 1425 ms remaining for execution. The logs also indicate that thread 354 has a maximum utility of 18.

As stated in Section 6.2.2, we actually work with the inverse PUD of threads. We do this since the kernel is not capable of efficiently handling floating point operations and the task utilities we consider are in the tens range of magnitude while the WCET, when considered in nanoseconds, are several thousands, at least, in magnitude – note that we summarize these times in milliseconds in this section to avoid clutter. Since ACUA uses the global PUD

for each task while constructing its schedule, it computes an inverse PUD of (note that we truncate the floating point parts of the inverse PUDs below because the kernel does not handle floating point numbers):

$$\frac{656+218}{4} = 218$$

On the other hand, thread 354 has a global inverse PUD of:

$$\frac{0+1037}{18} = 57$$

Remember that we are dealing with inverse PUDs, therefore the highest inverse PUD corresponds to the lowest PUD. In this scenario, thread 369, with an inverse PUD of 218, is obviously the thread with lower PUD when compared to thread 354 which has an inverse PUD of 57. Therefore, ACUA chooses thread 369 as the thread with the lowest PUD.

Let us now turn our attention to CUA. Remember that CUA uses normal PUD, not global PUD, on each node to locally order the list of sections it manages. Therefore, at the same point in time above, CUA would compute the inverse PUD of thread 369 as:

$$\frac{218}{4} = 54$$

While the inverse PUD of thread 354 would remain the same at 57. Obviously, at this point, thread 354 is considered the thread with the lowest PUD since its inverse PUD, 57, is larger than the inverse PUD, 54, of thread 369. Therefore, CUA removes thread 354 from the schedule and includes it in its reject set. Given this snap shot, it is easy to see that ACUA makes the "correct" decision in allowing the thread that can accrue the most utility to remain in the system. Specifically, it allows thread 354, with utility 18, to continue execution and terminates thread 369, with utility 4, from the system. If no more threads arrived after this point in time, ACUA would be the winner in terms of accruing more utility since its decision would accrue 18 units of utility to the application while CUA's decision would accrue only 4 units of utility. Why then does CUA accrue more utility for this data point?

In order to answer this question, we need to look at the scheduling decision that occurs when a new thread enters into the system after the above decision has been made. Table 7.4 depicts the schedule when the next thread, which we shall call Thread2 for ease of exposition, arrives into the system.

As can be seen, at this point 365 and 368 have completed execution and left the system, and thread 370 (our Thread2) has arrived into the system. Thread 354, which had been allowed to remain in the system when ACUA made its previous scheduling decision, is still in the system at this instant of time. Its remaining execution time has now shrunk to 841 ms. As

Table 7.4: Schedule on Arrival of Thread2 for ACUA

PID	Start	WCET	Deadline	End
354	1 sec, 366 ms	841 ms	2 sec, 363 ms	2 sec, 207 ms
370	2 sec, 207 ms	427 ms	2 sec, 363 ms	2 sec, 634 ms
366	2 sec, 635 ms	285 ms	4 sec, 363 ms	2 sec, 920 ms

can be seen in the table, the schedule is not feasible (the task that will miss its deadline is highlighted in green). So, another task needs to be removed from the system. ACUA chooses thread 354 as its candidate for removal. In CUA, the schedule is feasible at this point, because the task it chose to remain in the system at the previous scheduling event, thread 369, has a much smaller execution time for its local section (218 ms compared to the 1425 ms of thread 354). Therefore no other thread is removed from the schedule.

We now have a complete explanation for the behavior seen at this data point. ACUA makes the correct decision at the current instant of time by favoring the threads that would accrue the most utility to the system. However, since the algorithm is not clairvoyant, it cannot know if its decision at the current instant of time will lead to deadline misses when a new thread enters into the system. This behavior is inherent to the PUD heuristic for favoring tasks and the greedy nature of the resulting algorithm [31]. Despite this scenario, ACUA outperforms CUA and HUA in most instances since this is a very special scenario that does not occur very often. This can be seen from the other data points in the plots.

7.1.3 Ultimate Deadline Method

In this section, we describe the results of experiments conducted on CUA and ACUA using the ultimate deadline method, in which the deadline is not decomposed among the sections of the DT but each section is assigned the entire end-to-end deadline of the DT. This is attractive for its simplicity, but can sometimes mislead the scheduler into believing that earlier sections in the DT have more slack than they really do.

The task set used for the first experiment using the ultimate deadline method is depicted in Table 7.2. The result of this experiment is depicted in Figures 7.33, 7.34, 7.35 and 7.36. In this experiment, the AUR differences between case two scheduling and case three scheduling are very pronounced, with ACUA gaining 39% more utility than HUA at the 230% transactional load point. However, the difference between ACUA and CUA is less pronounced with ACUA having a maximum lead of 4% at the 230% transactional load point.

An interesting point to note is that CUA has better DSR than ACUA for all data points in this experiment, despite the fact that ACUA accrues more utility in these data points. Again, this is due to the fact that ACUA chooses the “right” set of tasks to execute – i.e., it

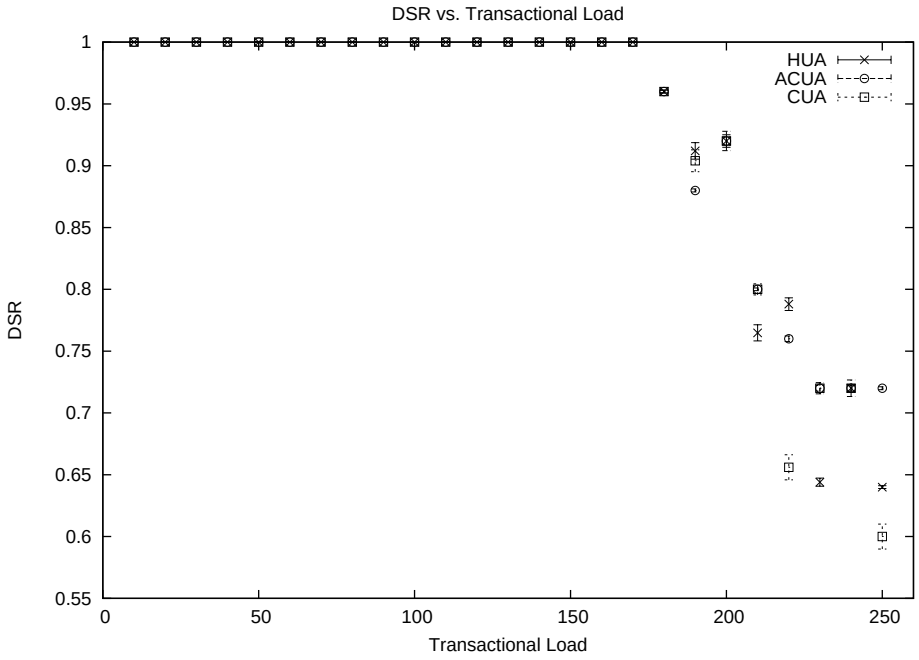

Figure 7.29: DSR vs. Transactional Load (PS, T2)

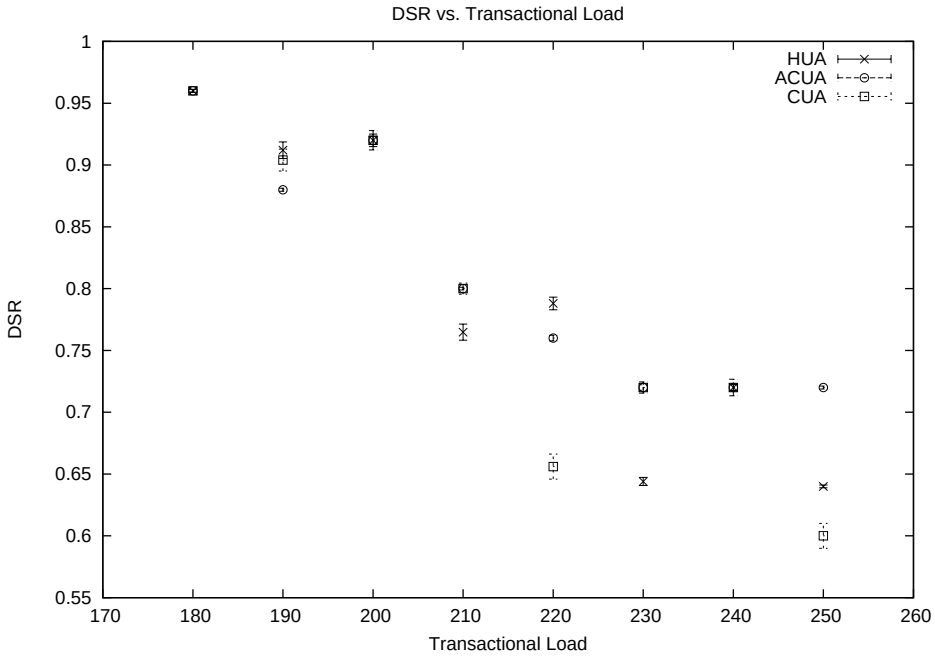

Figure 7.30: DSR vs. Transactional Load (OL, PS, T2)

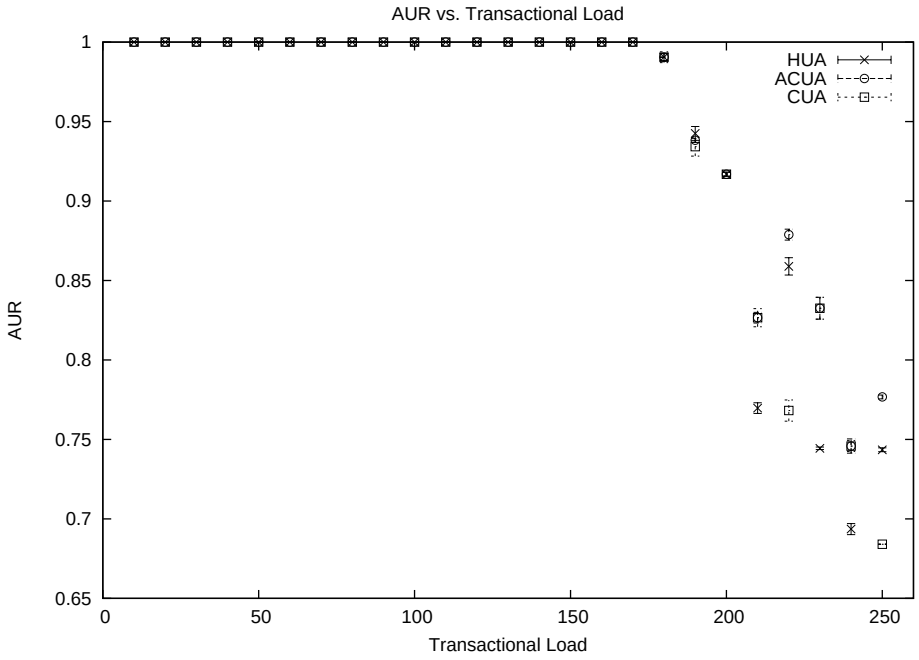

Figure 7.31: AUR vs. Transactional Load (PS, T2)

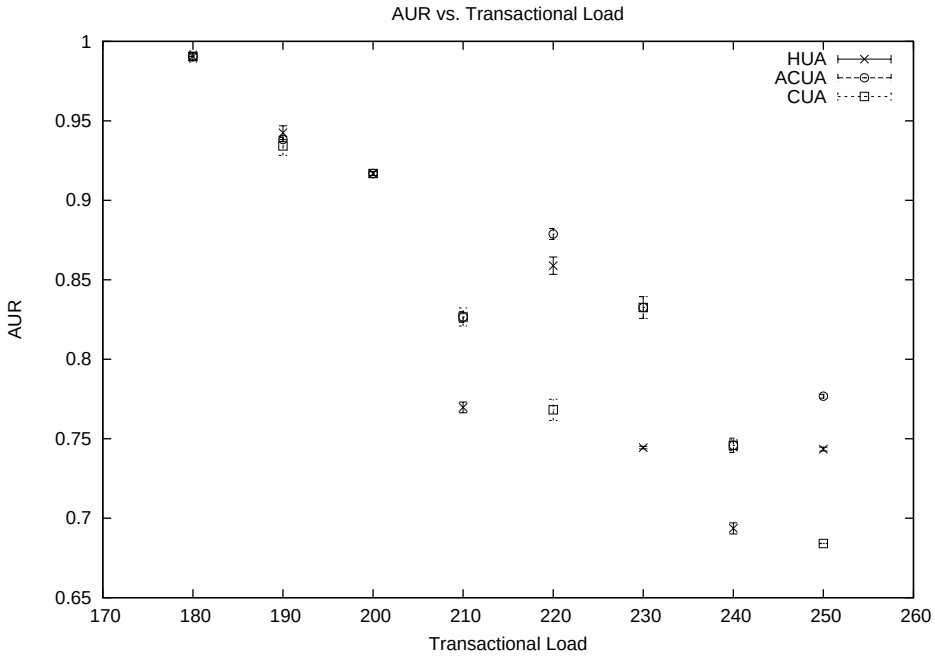

Figure 7.32: AUR vs. Transactional Load (OL, PS, T2)

eliminates more tasks from the system in order to allow high utility tasks to complete their execution.

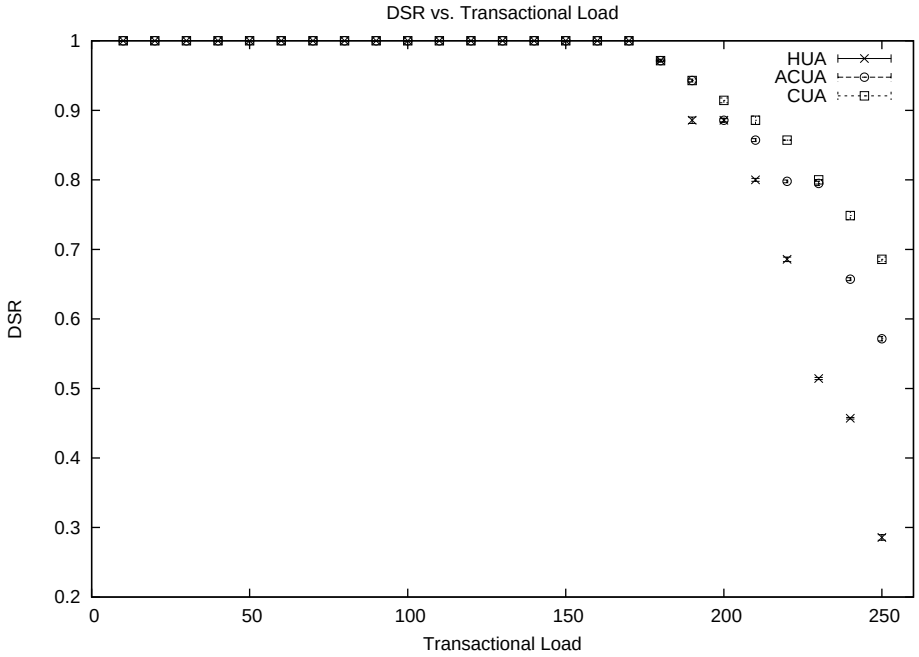

Figure 7.33: DSR vs. Transactional Load (UD, T1)

We believe that it is the ultimate nature of the deadline that cause the performance of ACUA and CUA to be similar. Remember that ACUA has a better best-effort property, therefore, when tasks need to be eliminated it can do a better job that CUA. However, since we are using the ultimate deadline method, the decision to eliminate tasks is delayed as much as possible, thus resulting in less opportunity to show-case ACUA's better best-effort property. HUA fairs worse since it has no way of eliminating threads if their future sections cannot execute. Therefore, it actually waits until a remote invocation is made and some execution on the remote node occurs before it decides that the thread is infeasible and removes it. This causes unnecessary consumption of CPU time, thus leading to the generally lower results for independent schedulers.

Figures 7.37, 7.38, 7.39 and 7.40 depict another set of results for this decomposition method using task set in Table 6.1. As can be seen, ACUA performs better than both HUA and CUA using the AUR metric. It should also be noted that HUA sometimes outperforms ACUA using the DSR metric but performs worse than ACUA in the AUR metric. This indicates that ACUA chooses the "right" set of threads in order to increase the utility accrued to the system. Again, the collaborative schedulers significantly outperform the independent scheduler, but their result is relatively close to each other.

While the ultimate deadline decomposition method increases the difference in performance between the collaborative schedulers and the independent schedulers, it does not lend itself to

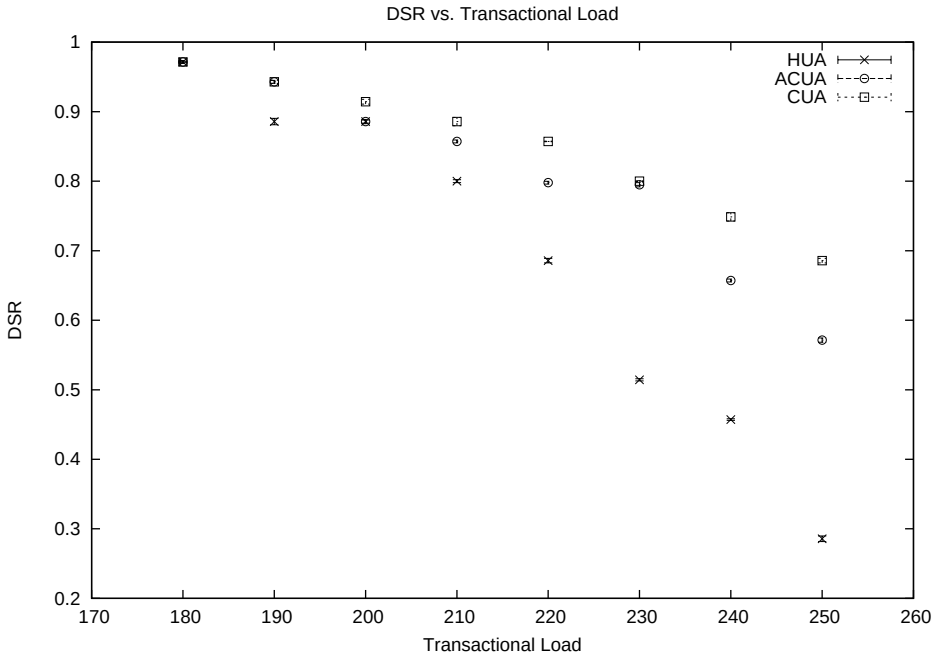

Figure 7.34: DSR vs. Transactional Load (OL, UD, T1)

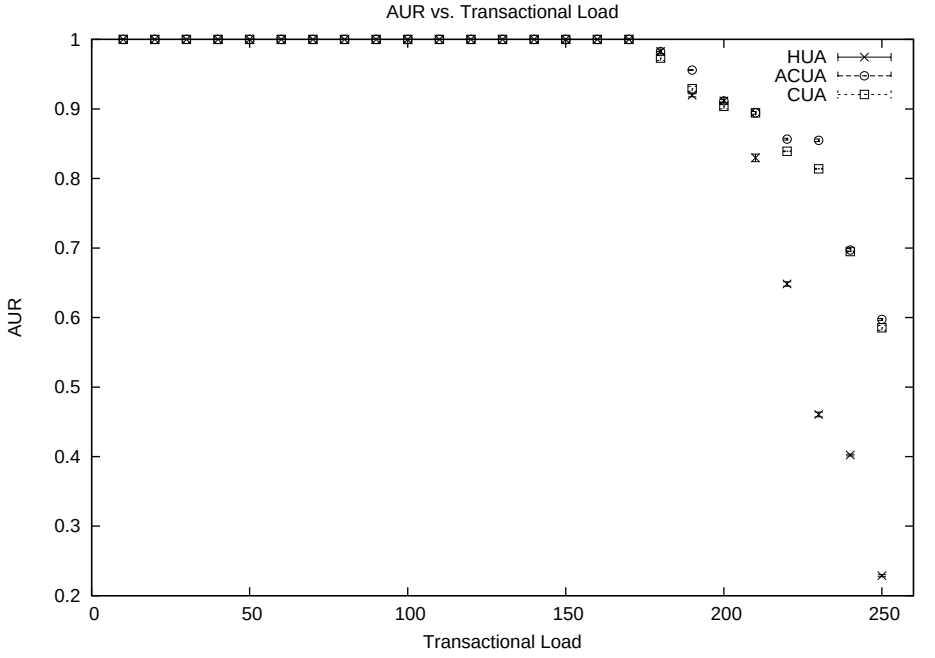

Figure 7.35: AUR vs. Transactional Load (UD, T1)

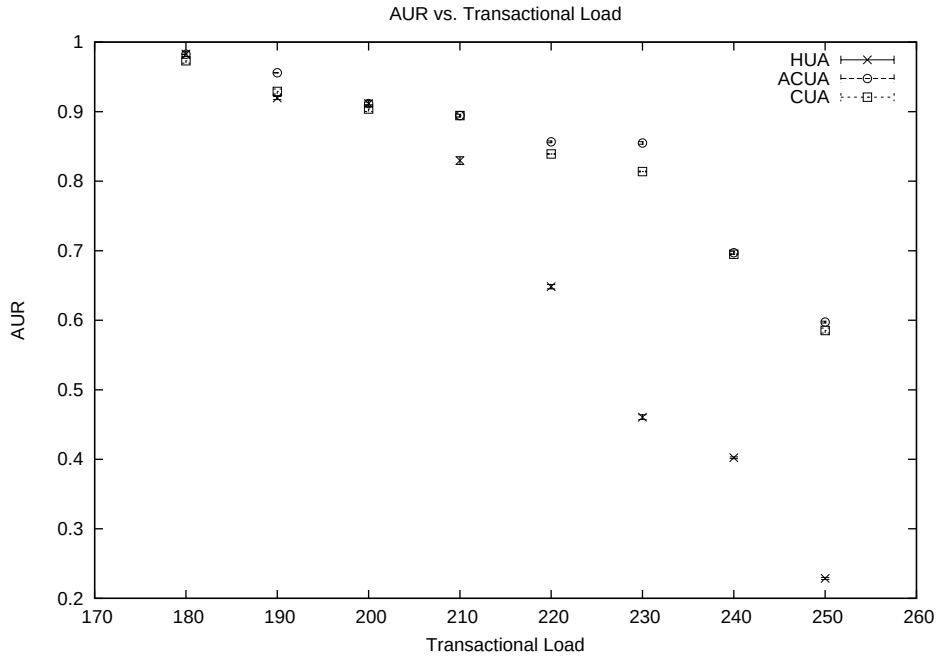

Figure 7.36: AUR vs. Transactional Load (OL, UD, T1)

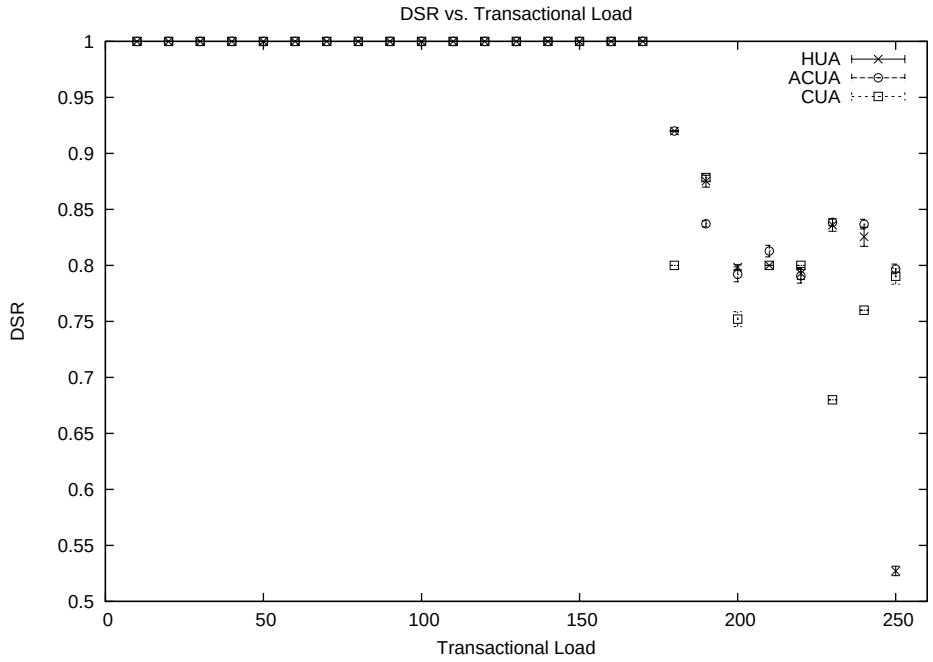

Figure 7.37: DSR vs. Transactional Load (UD, T2)

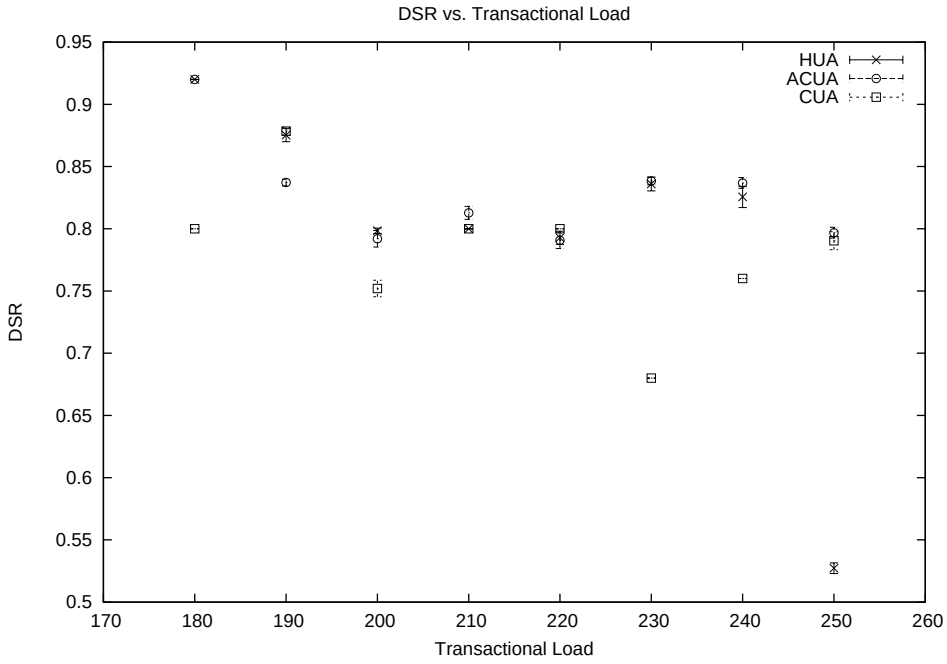

Figure 7.38: DSR vs. Transactional Load (OL, UD, T2)

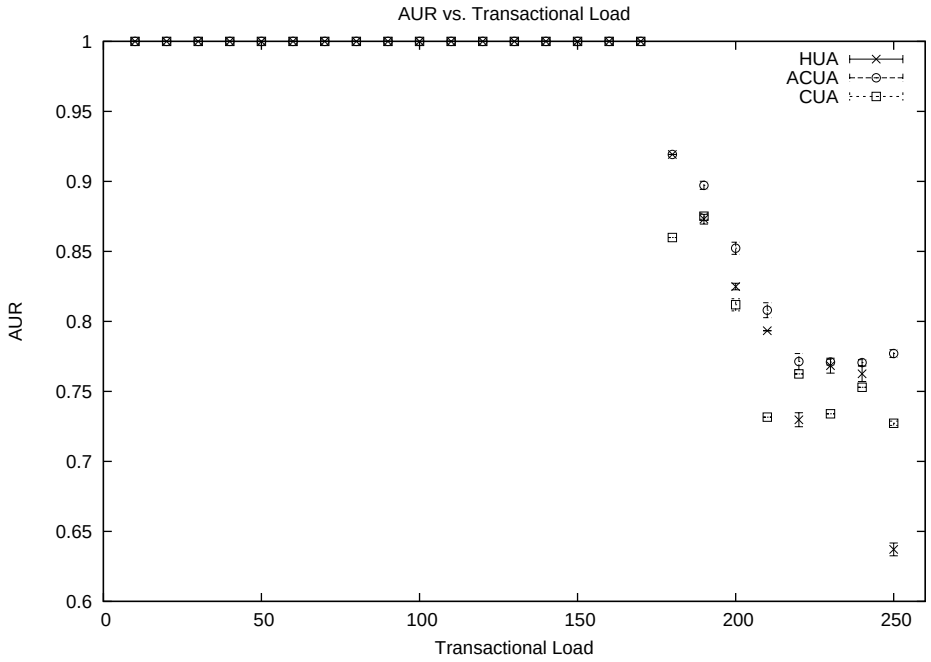

Figure 7.39: AUR vs. Transactional Load, (UD, T2)

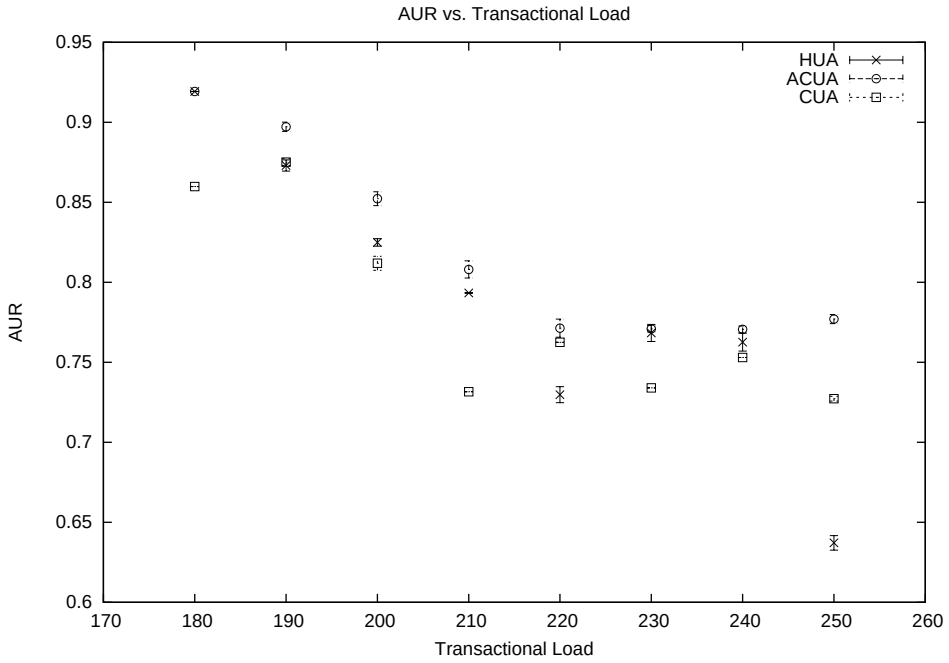

Figure 7.40: AUR vs. Transactional Load (OL, UD, T2)

practical applications since it completely ignores the scheduling needs of subsequent sections while scheduling the current sections. This is particulary problematic for systems where threads of significantly different length (i.e., with significantly different number of sections) exit. For that reason, we shall not produce results for this particular method of deadline decomposition in the rest of the thesis now that we have seen how it behaves in general.

7.2 Quorum-Based Collaborative Scheduler

In this section, we show the results for QBUA, a quorum-based scheduling algorithm. As previously mentioned, we attempted to reduce the overhead of the consensus-based algorithms by using a quorum-based algorithm. This reduces the communication patterns from broadcasts to multicasts and is invariant with respect to the number of failures in the system. In the following sections, we describe the results we obtained for the quorum-based algorithm we developed.

7.2.1 Worst Case Deadline Decomposition

First task set experimented on is depicted in Table 7.2. Figures 7.41, 7.42, 7.43 and 7.44 show the results of this experiment. As can be seen, QBUA outperforms all other algorithms, but

follows closely the performance of ACUA. The maximum difference between QBUA and all the other algorithms in this experiment occurs at the 270% transactional load point. QBUA has a maximum lead of only about 2% over ACUA, about 8% over CUA and 13% over HUA.

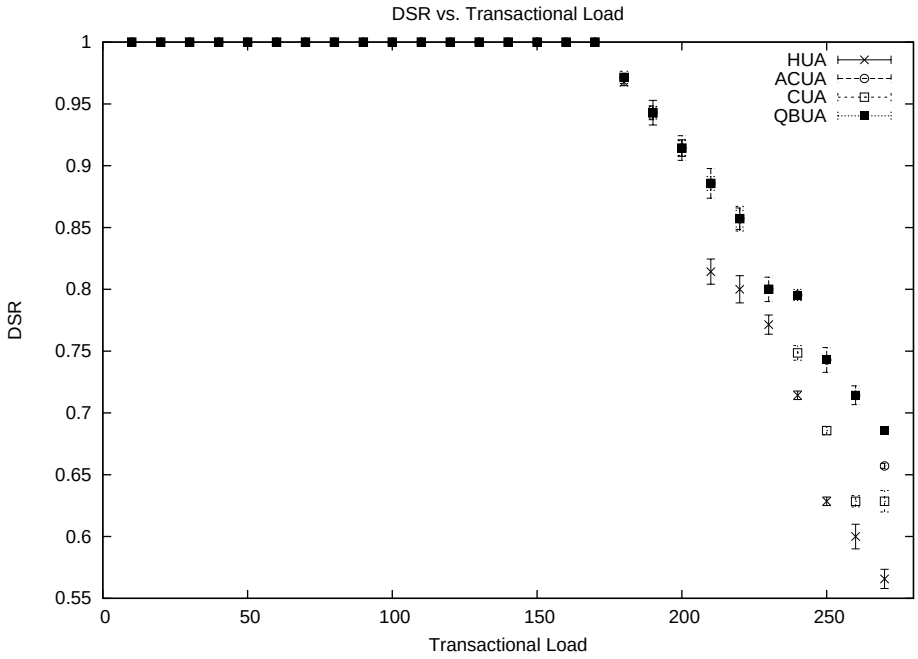

Figure 7.41: DSR vs. Transactional Load (WD, T1, QBUA)

As before, we consider the case of using non-zero overhead failure handlers to the tune of 10% of processing time. The results are depicted in Figures 7.45, 7.46, 7.47 and 7.48. As can be seen, adding the failure handlers once more allows the algorithms to differentiate themselves as each of them get a better opportunity to highlight their ability to choose the "correct" set of tasks to keep when the number of tasks that should be removed from the schedule increases. The difference between HUA and QBUA now increases from 13% to 18%. The difference between CUA and QBUA expands from 8% to 13%, and the difference between ACUA and QBUA increases from 2% to 3%. Note that ACUA still tracks QBUA quite closely i.e., it is minimally affected by the increased opportunity to show-case its best-effort behavior against QBUA due to the increased deadline misses introduced by the failure handlers. This is due to the fact that the main difference between QBUA and ACUA are their overhead terms, not their best-effort properties.

The next task set we experimented on is depicted in Table 7.5. The results are depicted in Figures 7.49, 7.50, 7.51 and 7.52. For this data set, the performance of all collaborative scheduling algorithms follow each other closely. The collaborative algorithms have a maximum lead over HUA of about 15% at the 250% transactional load point. Note that all our comparisons have been using the AUR metric, since it is the best metric for informing us how well the algorithms conform to the best-effort criteria. As noted when discussing previous

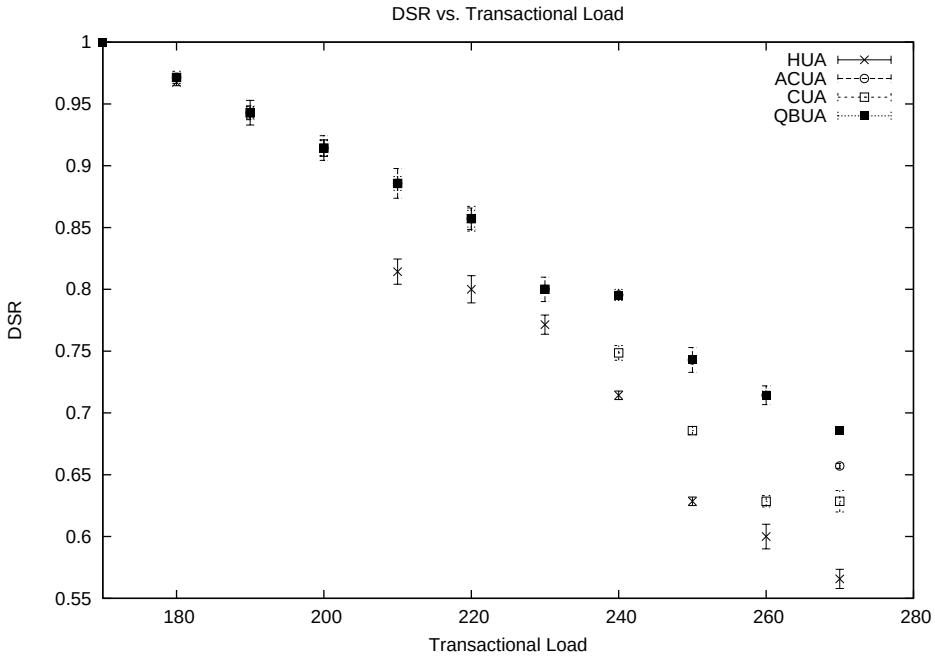

Figure 7.42: DSR vs. Transactional Load (OL, WD, T1, QBUA)

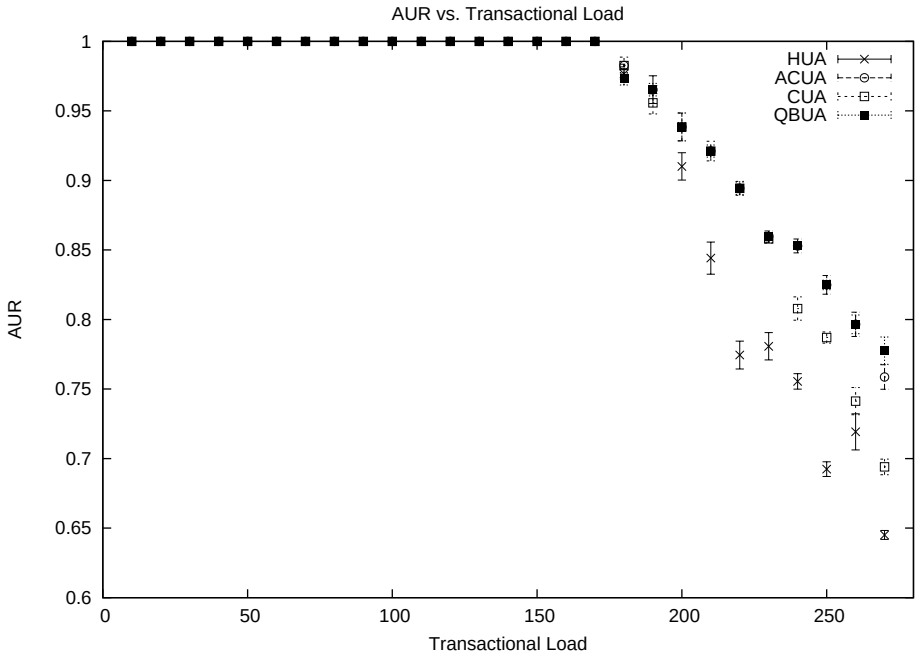

Figure 7.43: AUR vs. Transactional Load, (WD, T1, QBUA)

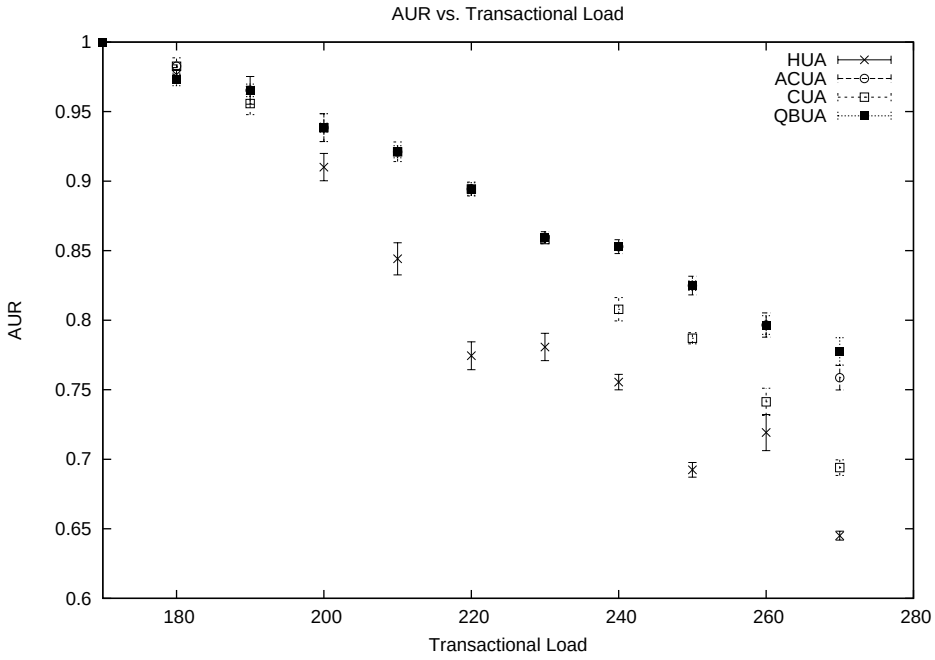

Figure 7.44: AUR vs. Transactional Load (OL, WD, T1, QBUA)

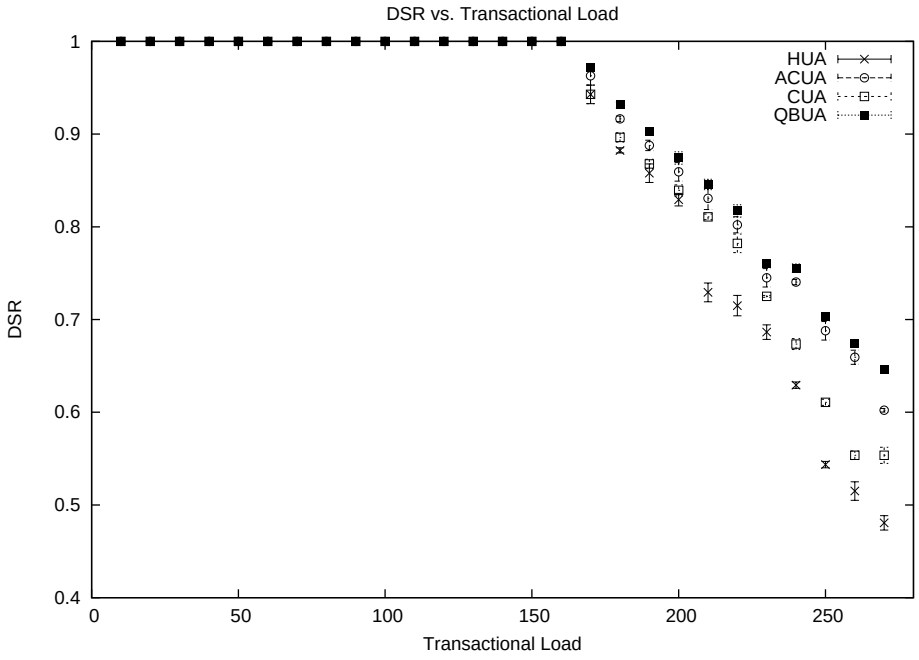

Figure 7.45: DSR vs. Transactional Load (WD, T1, QBUA, HL=0.1)

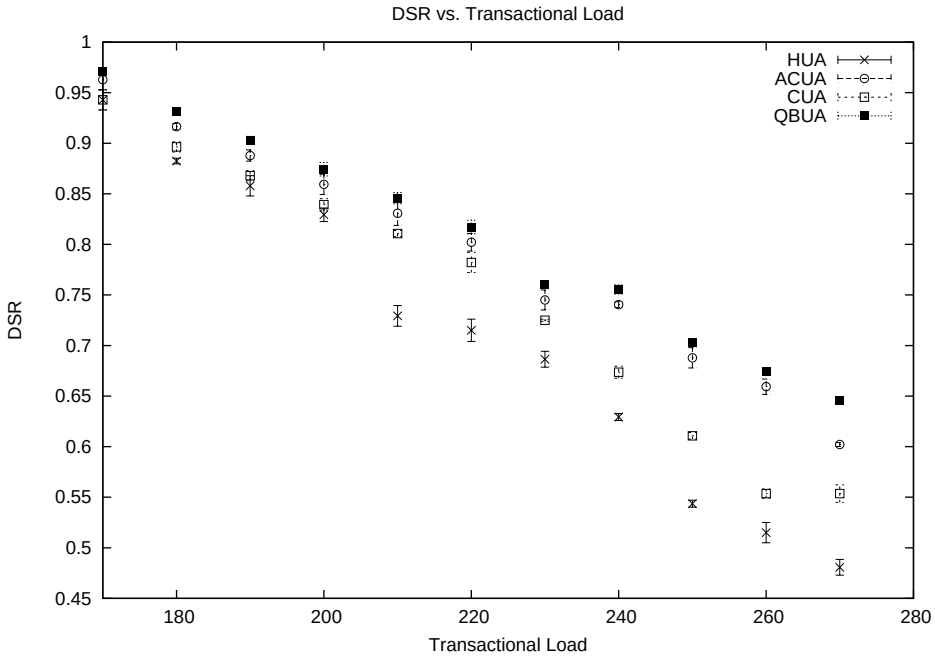

Figure 7.46: DSR vs. Transactional Load (OL, WD, T1, QBUA, HL=0.1)

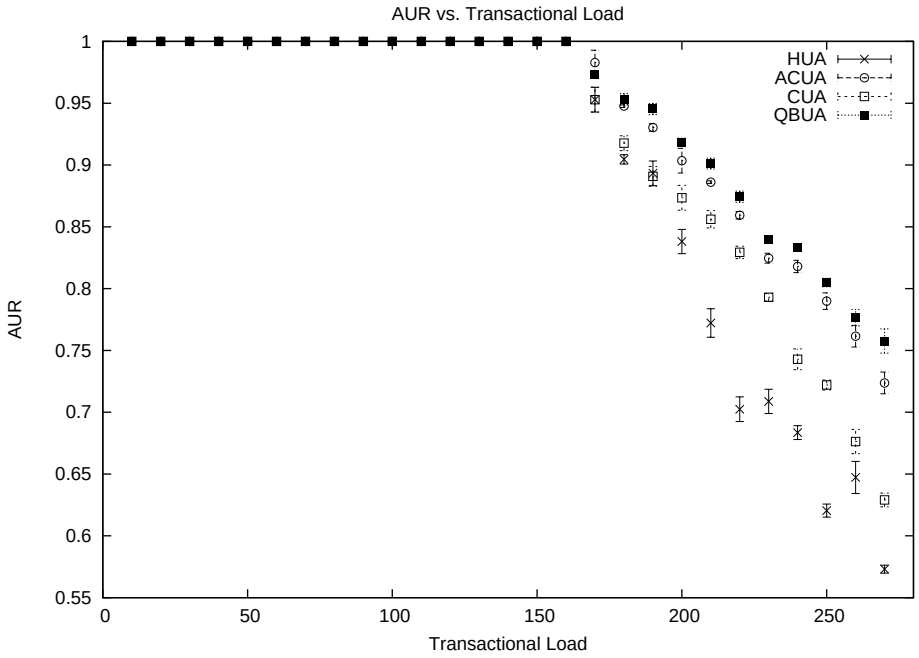

Figure 7.47: AUR vs. Transactional Load, (WD, T1, QBUA, HL=0.1)

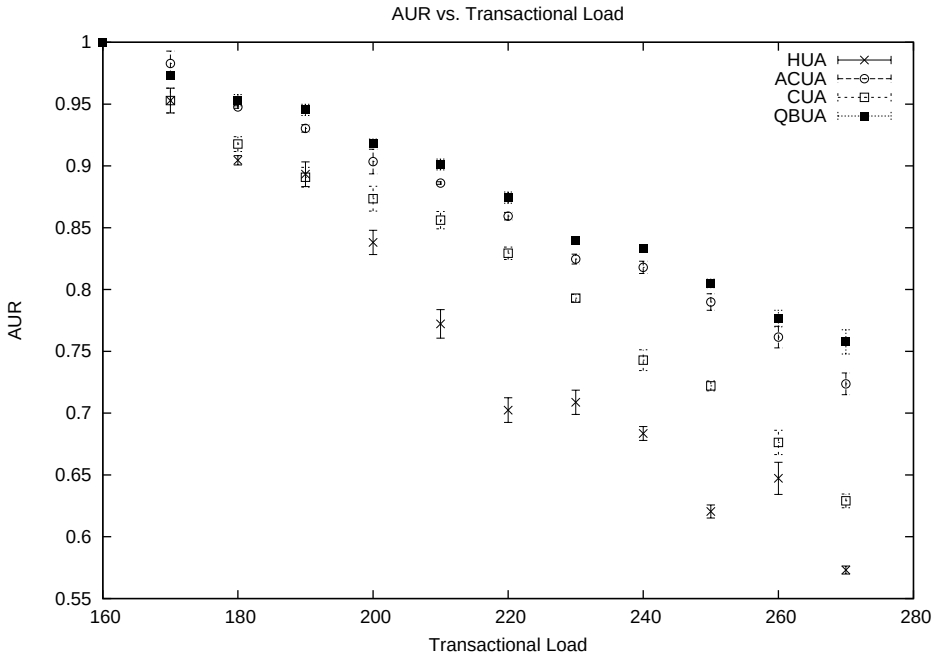

Figure 7.48: AUR vs. Transactional Load (OL, WD, T1, QBUA, HL=0.1)

Table 7.5: Another Set of Task Set Parameters

	Period (μsecs)	Utility
T1	2823000	80
T2	807000	55
T3	2915000	84
T4	2966000	76
T5	3659000	66

results, the DSR may not always follow AUR because best-effort algorithms can trade-off meeting more deadlines for meeting more important, but fewer, deadlines. Therefore, it is possible to notice a drop in DSR and a corresponding increase in AUR for certain data points.

Next, we perform the same experiment as above, i.e., on the task set described in Table 7.5, but use failure handlers with an offered load of 10%. The results are depicted in Figures 7.53, 7.54, 7.55 and 7.56. Again, adding the non-zero overhead failure handlers allows the algorithms to differentiate themselves from each other. ACUA still closes tracks QBUA, with a maximum difference in AUR of slightly about 1%, but HUA is outperformed to the tune of 19% at the 250% transactional load point, while CUA is outperformed to the tune of about 7% at the 220% transactional load point. Now that we have seen the effect of non-zero overhead failure handlers we skip the plots for these experiments in Section 7.2.2 since it follows exactly the same pattern.

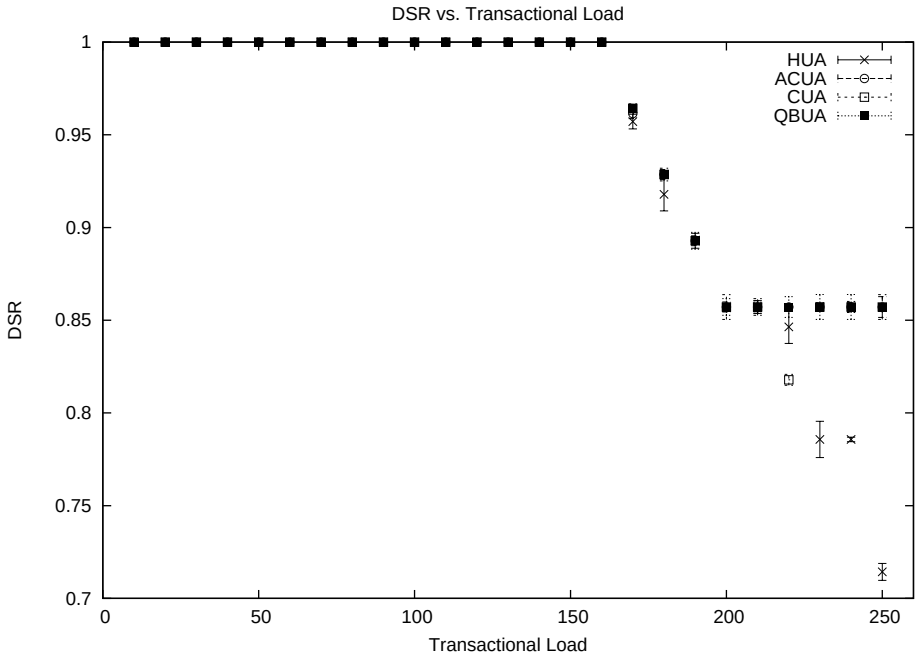

Figure 7.49: DSR vs. Transactional Load (WD, T4, QBUA)

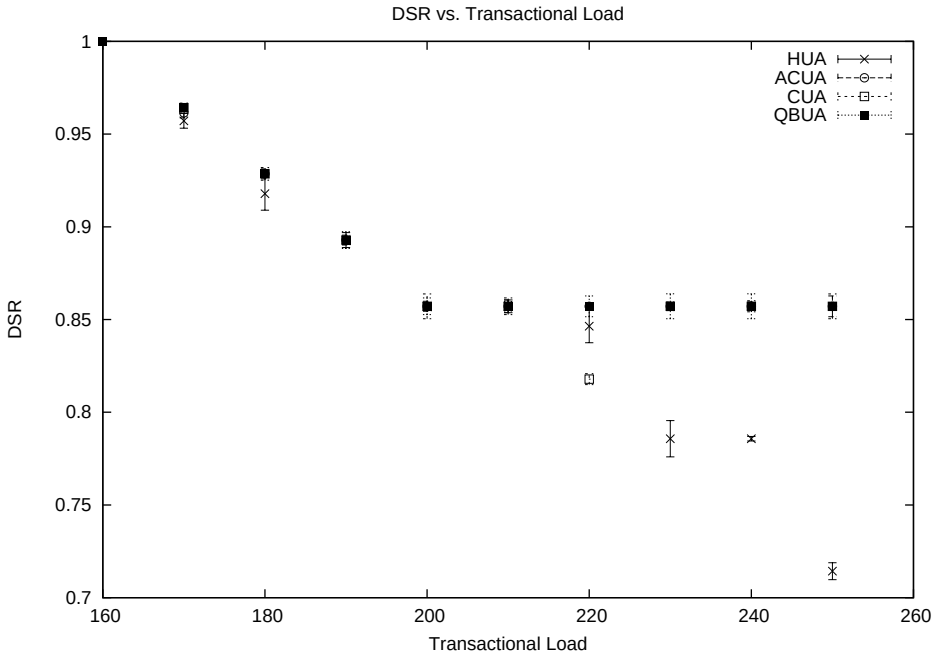

Figure 7.50: DSR vs. Transactional Load (OL, WD, T4, QBUA)

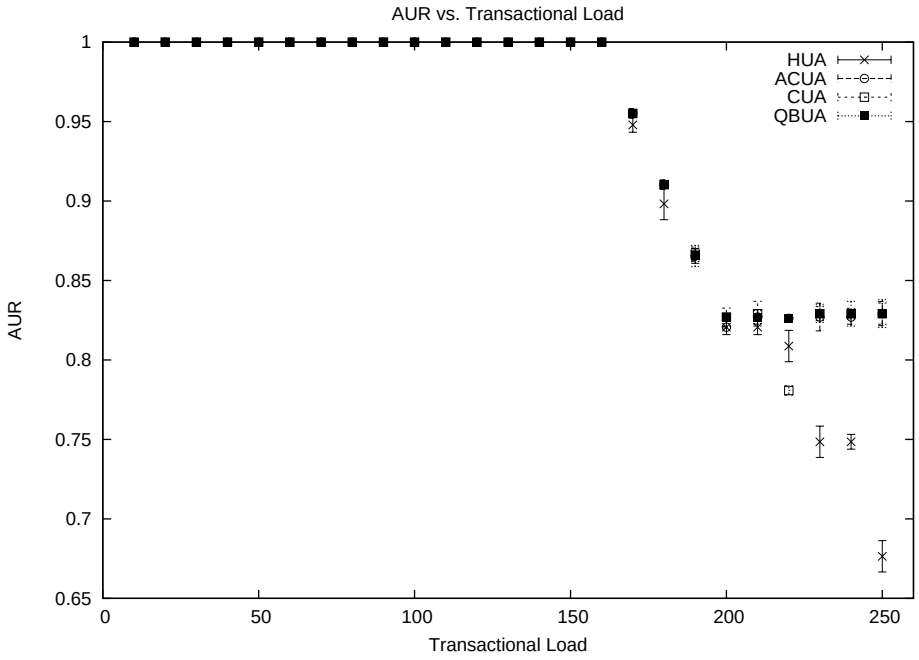

Figure 7.51: AUR vs. Transactional Load (WD, T4, QBUA)

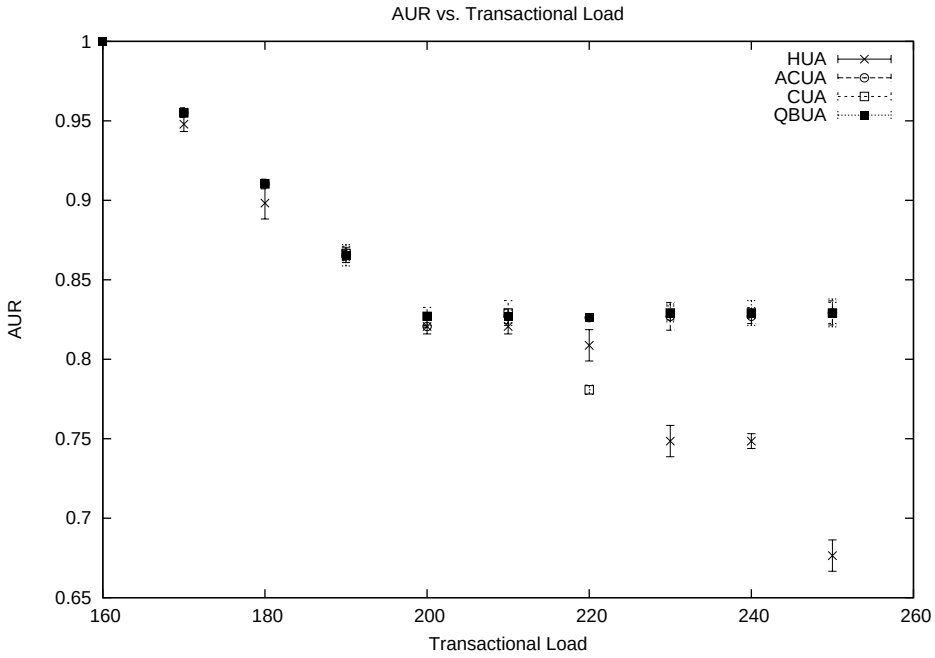

Figure 7.52: AUR vs. Transactional Load (OL, WD, T4, QBUA)

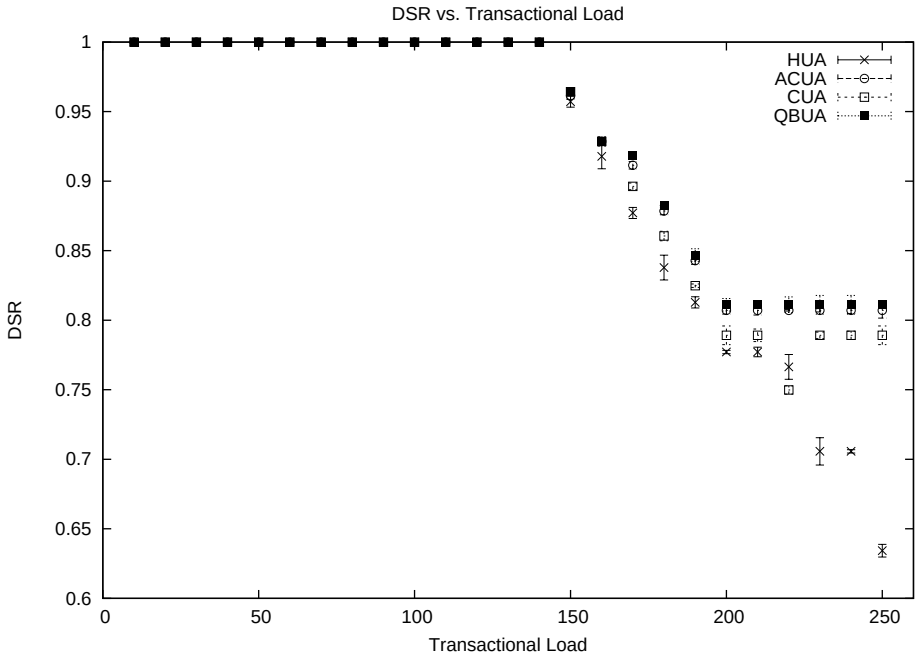

Figure 7.53: DSR vs. Transactional Load (WD, T4, QBUA, HL=0.1)

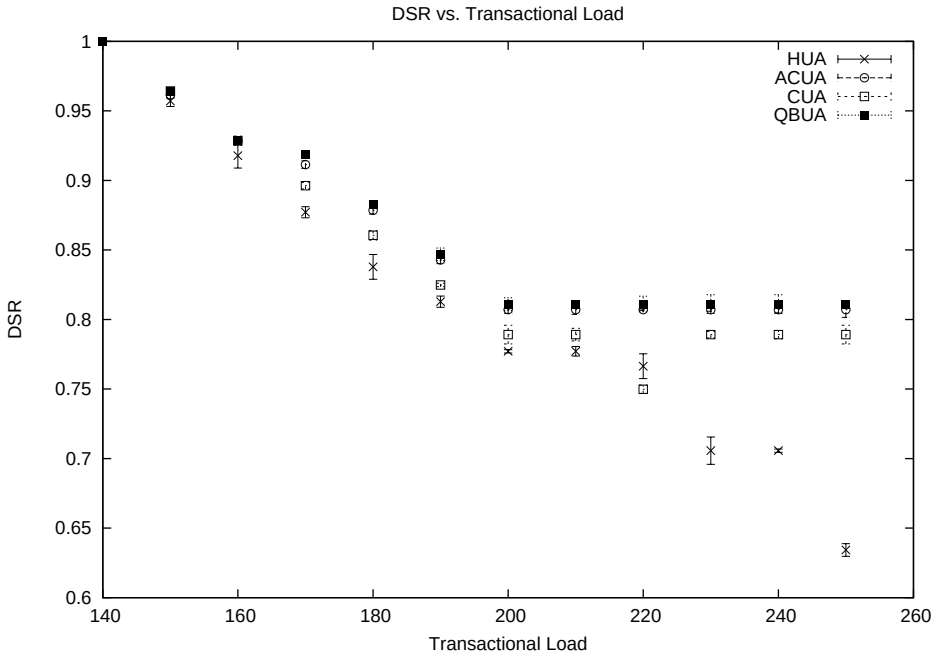

Figure 7.54: DSR vs. Transactional Load (OL, WD, T4, QBUA, HL=0.1)

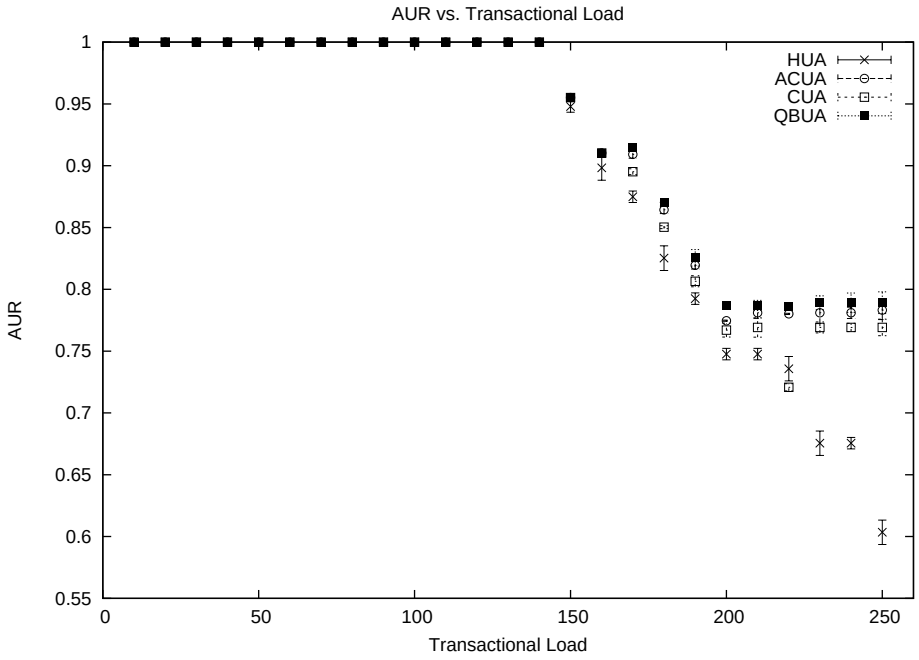

Figure 7.55: AUR vs. Transactional Load (WD, T4, QBUA, HL=0.1)

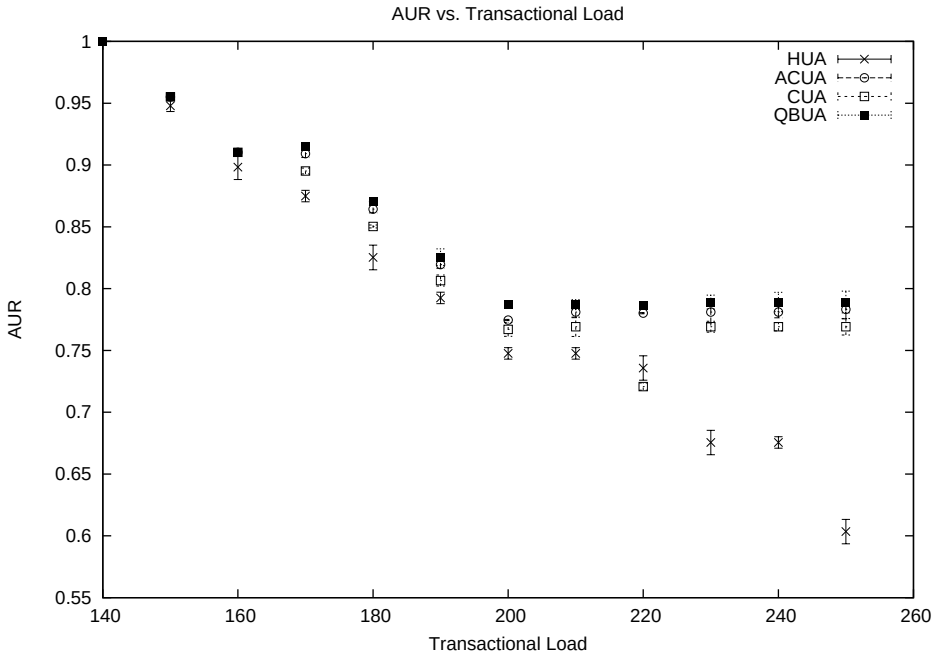

Figure 7.56: AUR vs. Transactional Load (OL, WD, T4, QBUA, HL=0.1)

7.2.2 Proportional Slack Assignment

The data set we consider for this deadline decomposition method is depicted in Table 7.5. The result for this experiment is depicted in Figures 7.57, 7.58, 7.59 and 7.60. Again, QBUA outperforms the other algorithms in terms of AUR and this is particularly obvious towards the end of the AUR plot. QBUA has a maximum AUR gain of 5% over HUA at the 210% transactional load point, a maximum gain of 3% over ACUA at the 250% transactional load point, and a maximum gain of 5% over CUA at the 250% transactional load point.

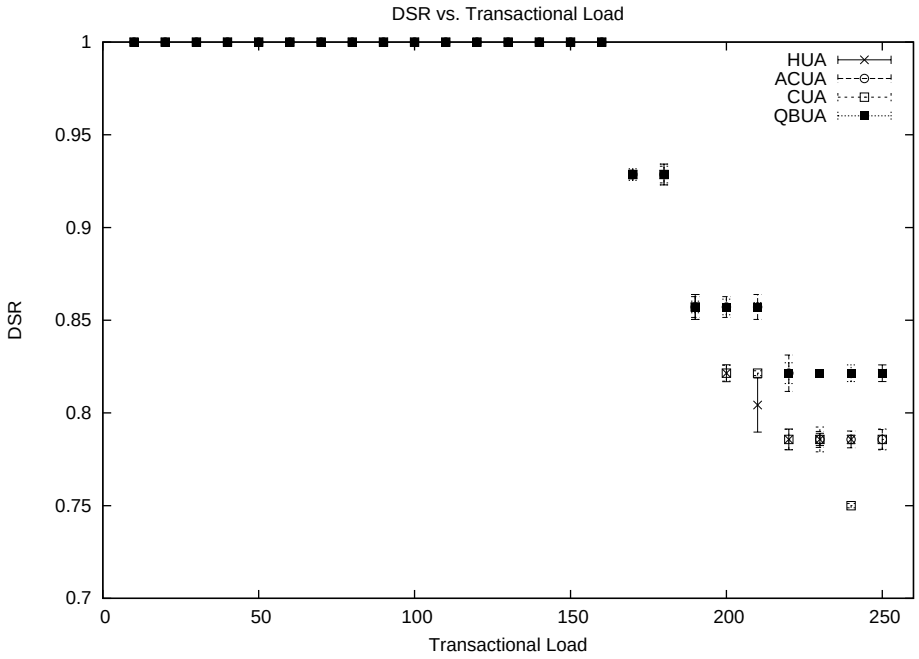

Figure 7.57: DSR vs. Transactional Load (PS, T4, QBUA)

The 190% transactional load point provides an interesting data point for the algorithm. In this data point, the "right" set of tasks to accept using the global PUD criteria actually results in less utility accrual than making isolated decisions on each node. The explanation for this is similar to the detailed analysis that we provide in Section 7.1.2 and we do not reproduce it here again.

7.3 Failures

In this section, we perform a number of experiments to investigate the effect of failures on the collaborative scheduling algorithms. We limit our experiments to the worst-case deadline decomposition method and report the AUR and DSR accrued for thread failure rates ranging from 0% to 40%. In order to achieve this failure rate, we programmatically fail one of the

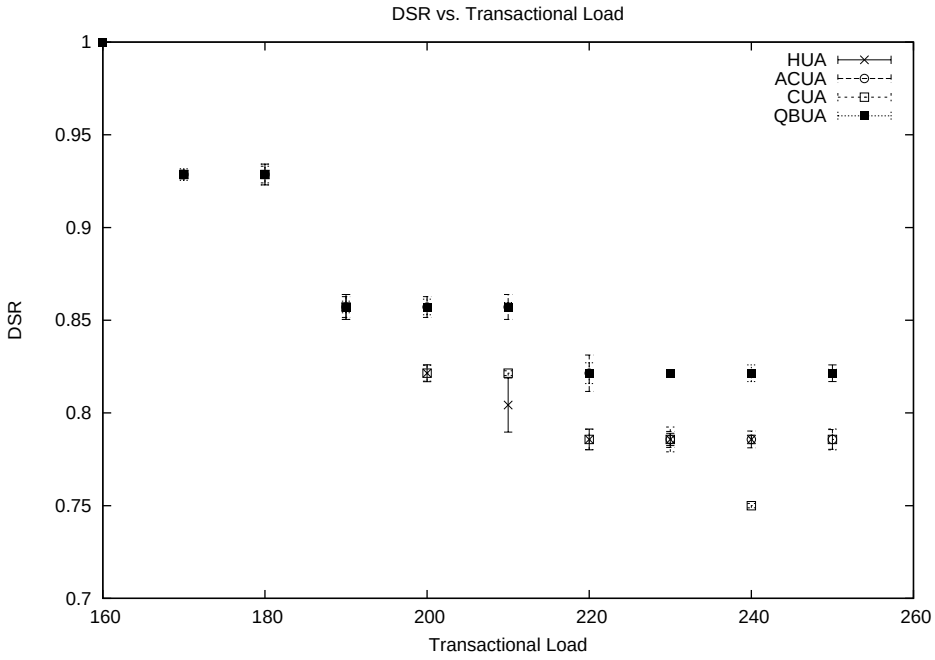

Figure 7.58: DSR vs. Transactional Load (OL, PS, T4, QBUA)

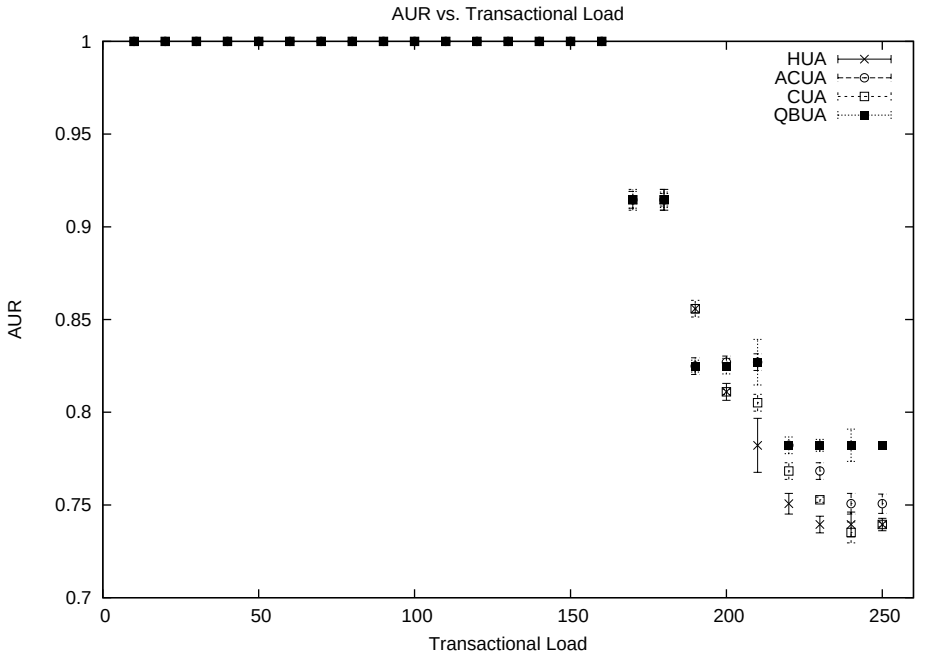

Figure 7.59: AUR vs. Transactional Load (PS, T4, QBUA)

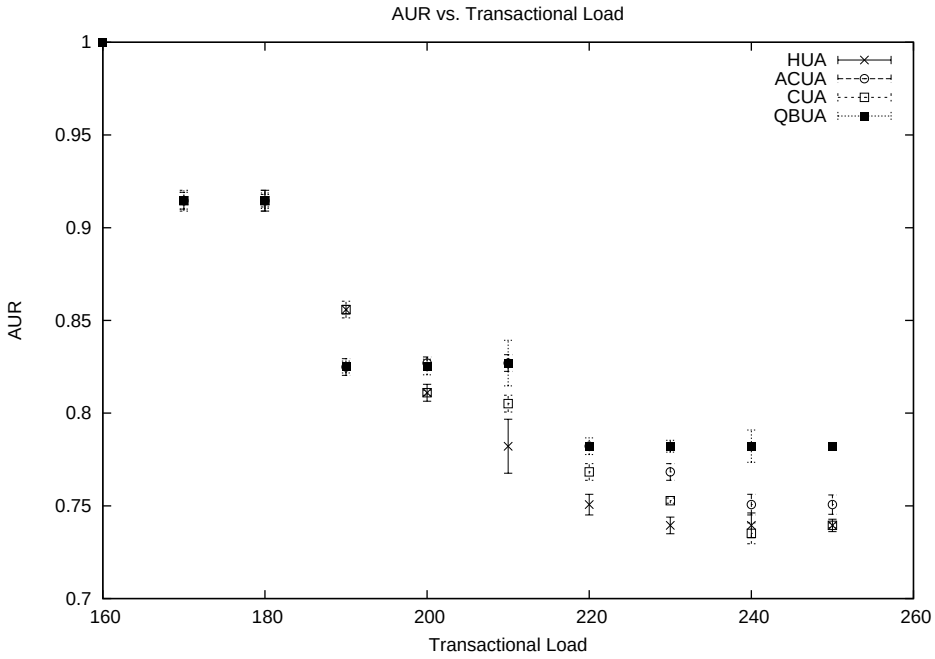

Figure 7.60: AUR vs. Transactional Load (OL, PS, T4, QBUA)

nodes in the system. Before the experiment starts, we run the setup without any failures. During this run, we take note of the total number of threads executed. Subsequently, we can determine how many threads need to be failed in order to achieve the failure rate desired. During the experiment, we programmatically attempt to fail the node at random instants of time as long as the number of hosted threads on the node at the instant of failure does not bring the total number of failed threads over the desired thread failure rate. Note that due to the way we do this, it is not always possible to fail the exact number of threads desired. So, for example, a failure rate of 25% on a system with 50 threads will actually be a 24% failure rate since it is impossible to fail 12.5 threads. Nevertheless, we report the results against the desired failure rate and discard results that are more that 2% points away from the desired failure rate.

We perform the experiment on the thread set described in Table 7.2, and consider 0.1 overhead failure handlers. We fix the transactional load at the 270% point and report the average and 95% confidence intervals for the AUR and DSR measurements. Figures 7.61 and 7.62 depict the results of this experiment.

It will be noted that while the drop in DSR is relatively linear in the number of threads failed, the AUR isn't. This occurs because the threads failed at a lower failure rate are not necessarily an exact sub-set of the threads failed at higher failure rates. Therefore, it is possible for the higher failure rates to occur by failing more low PUD threads, this may allow UA algorithms to accrue better utility at these failure rates. The behavior of other thread sets in the presence of failure is similar to this behavior and, so, is not reproduced

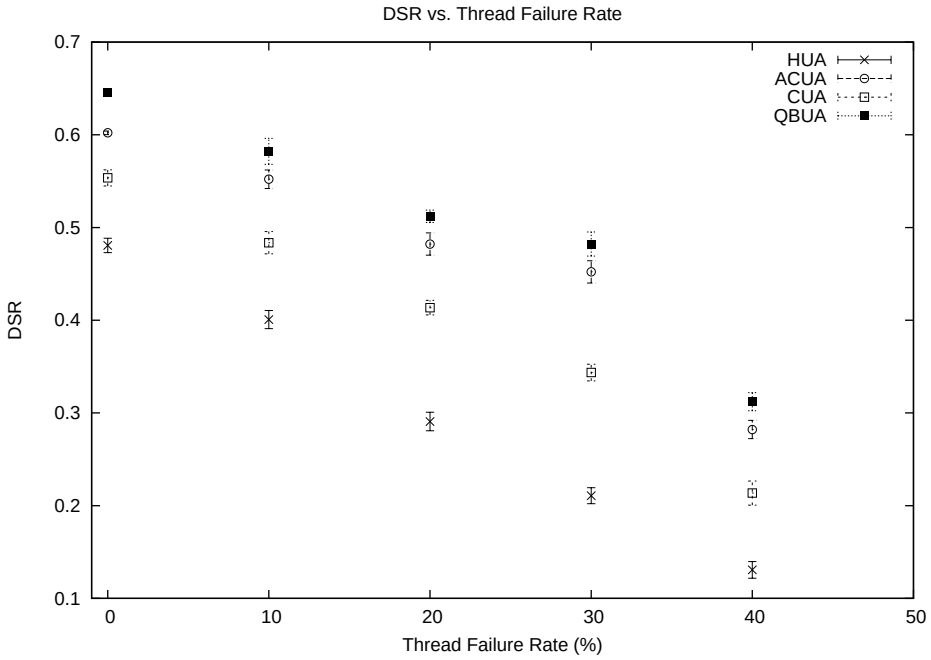

Figure 7.61: DSR vs. Thread Failure Rate (WD, T1)

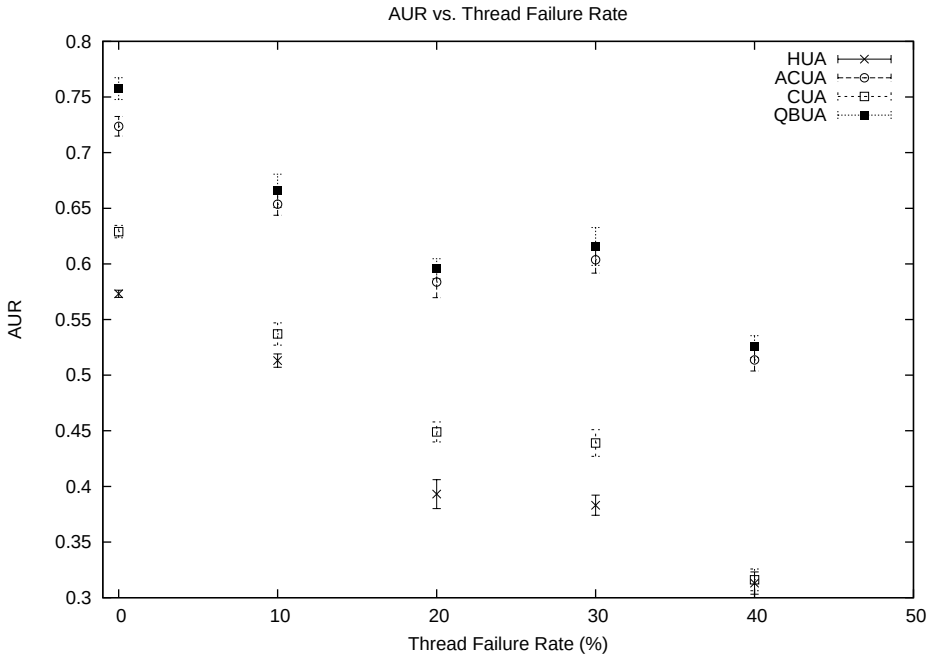

Figure 7.62: AUR vs. Thread Failure Rate (WD, T1)

here to avoid clutter. In this experiment, QBUA has a maximum lead over ACUA of about 3% at the 0% failure rate, a maximum lead over CUA of 21% at the 40% failure rate and a maximum lead over HUA of about 23% at the 30% failure rate.

7.4 Conclusions

In this chapter, we performed a number of experiments on collaborative scheduling algorithms. The results of these experiments indicate that collaborative scheduling can provide superior timeliness performance during overloads for a number of reasons. First, the collaborative scheduling algorithms we design take into account system-wide properties of the thread set being considered and therefore can avoid local minima in their decision making. These local minima occur when the execution times of the sections belonging to a single thread vary significantly – particularly, when this variability is different for different threads.

The variability in execution times leads to conflicting notions of the "goodness" of that particular thread on different nodes. For example, suppose that there are two competing threads, threads A and B, in a system. Assume that the maximum utility for thread A is larger than the maximum utility for thread B, but that the execution time of thread A's section hosted on the current node is much larger than the execution time of thread B's section hosted on the same node. This may lead to the decision to abort thread A because local information would imply that thread B should be favored. By allowing collaboration, a system-wide view of the threads in the system and the load on the nodes they are executing on would avoid such a local minimum and make a decision that would attempt to maximize system-wide accrued utility.

For such thread sets, collaborative scheduling outperforms its independent competitors. For example, for one such experiment, depicted in Figures 7.5, 7.6, 7.7 and 7.8, the average improvement of ACUA over HUA, its independent scheduling competitor, in terms of AUR is about 10% (with a standard deviation of 5%), while the maximum difference between ACUA and HUA is about 17%. Similarly, for the experiment depicted in Figures 7.45, 7.46, 7.47 and 7.48, QBUA outperforms its independent competitor, HUA, during overloads by an average of about 12% (with a standard deviation of 6%) and a maximum of about 18% for the AUR metric – similar trends are seen in other thread sets that have the same property.

Another reason that the collaborative schedulers considered in this dissertation may accrue higher utility than independent schedulers is that they consider future sections while making their scheduling decisions. By taking these sections into account, they can decide early if a particular thread will meet its end-to-end deadline. This early decision making allows the system to abort the thread and reclaim its resources for other threads that are expected to meet their end-to-end deadlines, thus increasing the overall benefit to the system.

That being said, collaborative scheduling is not suitable for all thread sets. Collaboration has a high overhead that is dominated by communication costs. This high overhead means that

only threads with large execution times (this is discussed further in Chapter 8) can actually benefit from collaboration. Also, even thread sets with large execution times that do not have significant variability in their section execution times are not expected to benefit from collaboration. For these thread sets, the high overhead of collaboration is not mitigated by any gain in utility accrued due to better scheduling decisions. An example of such a scenario is depicted in the results presented in Figures 7.17, 7.18, 7.19 and 7.20. Note that collaborative schedulers actually perform worse in this scenario.

Also, while the collaborative schedulers are designed to operate in partially synchronous systems, systems with very high variability in communication costs will significantly increase the overheads of the algorithms and introduce greater variability in their performance. We thus expect the algorithm to be of value for those systems that have large execution times, high variability in section execution times and relatively stable (but still stochastic) communication costs.

Chapter 8

Experimental Results with Dependencies

8.1 Distributed Scheduling with Dependencies

In this chapter, we study the behavior of various distributed scheduling algorithms when dependencies can occur. In particular, we compare the behavior of our collaborative scheduling algorithm, DQBUA, to HUA, case two EDF with the PIP protocol and case two RMS with the PIP protocol.

The rest of this chapter describes the results we obtained from this comparison for different methods for deadline decomposition. As mentioned in Chapter 7, we do not consider the ultimate deadline decomposition method since, while it maximizes the difference between case two and case three schedulers, it leads to significant loss of deadlines by overestimating the amount of slack that earlier sections in a DT have. This is particularly true if the system consists of DTs that have significantly varying numbers of sections.

8.2 Worst Case Deadline Decomposition

In this section, we consider the worst case deadline decomposition technique mentioned in the previous chapter. We consider a task set that uses the Class A method for WCET assignment and hence favors collaborative scheduling. We do not consider task sets that use the Class B method for WCET assignment since we have already established, in Chapter 7, that such task sets are not suitable for collaborative scheduling.

We study the behavior of the algorithm under two different influences – different sizes of critical sections and different number of locks. We now describe the results obtained for each

of these conditions.

8.2.1 Single Lock

In this section, we consider the case of a single lock on the performance of the algorithm. Towards that end, each of the sections we consider have a single critical section that they enter towards the end of their execution. A single futex-based lock protects this critical section. In this set of experiments, distributed dependencies are not allowed — i.e., the lock is released before the thread makes a remote invocation to another node.

For the first experiment conducted, we considered the thread set described in Table 7.2. We set the length of the critical section in each section of the DT to 5% of the execution time of that section. We also consider negligible overhead failure handlers (in terms of the timescales we consider) whose only function is to free the locks and return immediately. The result for this first experiment is shown in Figures 8.1, 8.2, 8.3 and 8.4.

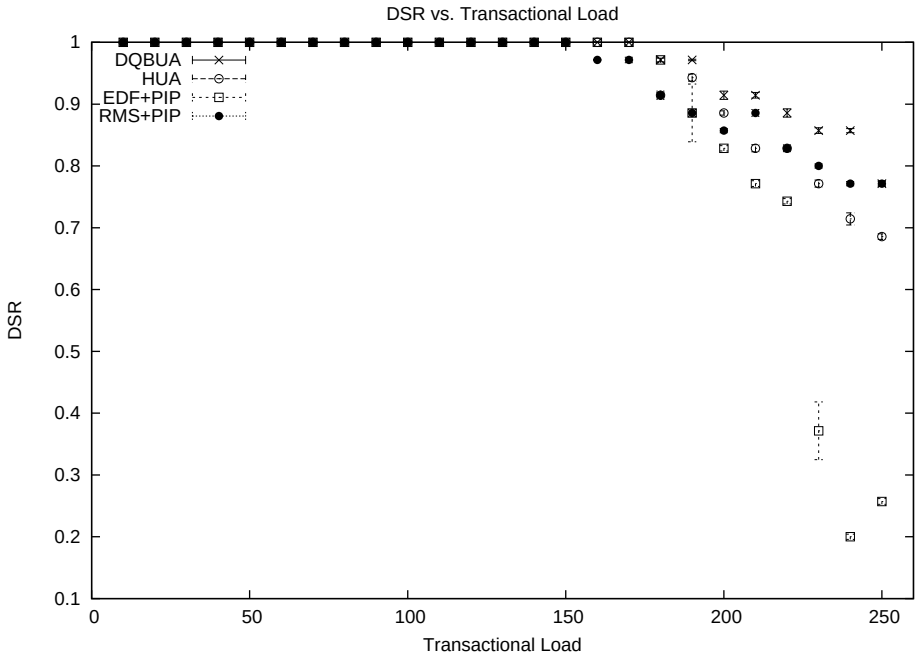

Figure 8.1: DSR vs. Transactional Load (WD, T1, CS=5%, NL=1)

As can be seen, DQBUA outperforms all the other algorithms. The maximum difference between DQBUA and all other algorithms occurs at the 240% transactional load point. It leads HUA with about 10%, EDF+PIP with about 73% and RMS+PIP with about 20% in AUR. Note that, like in Chapter 7, we report the differences in performance in terms of AUR since that is the most relevant to the best-effort property.

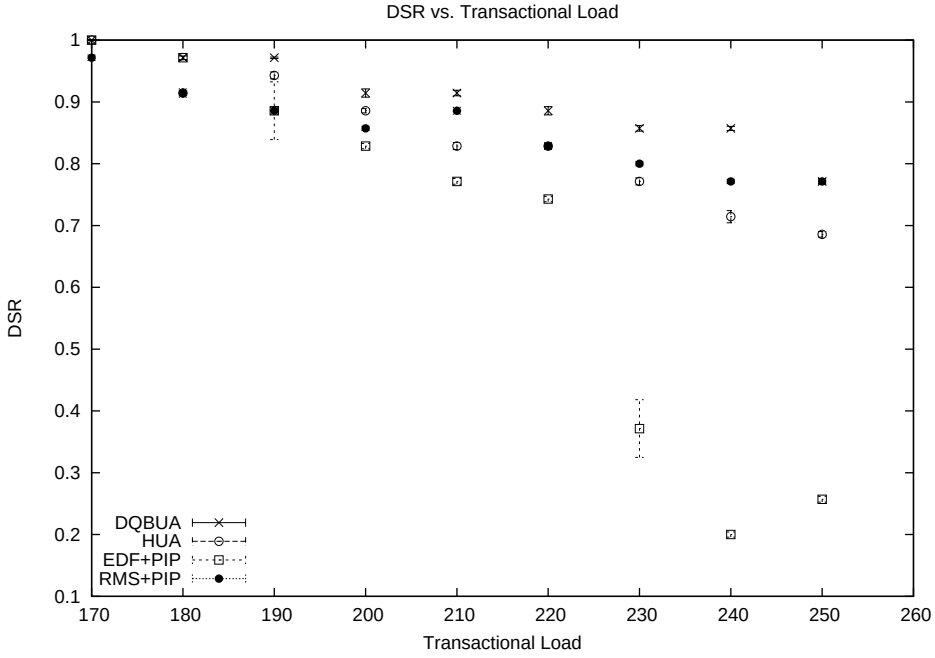

Figure 8.2: DSR vs. Transactional Load (OL, WD, T1, CS=5%, NL=1)

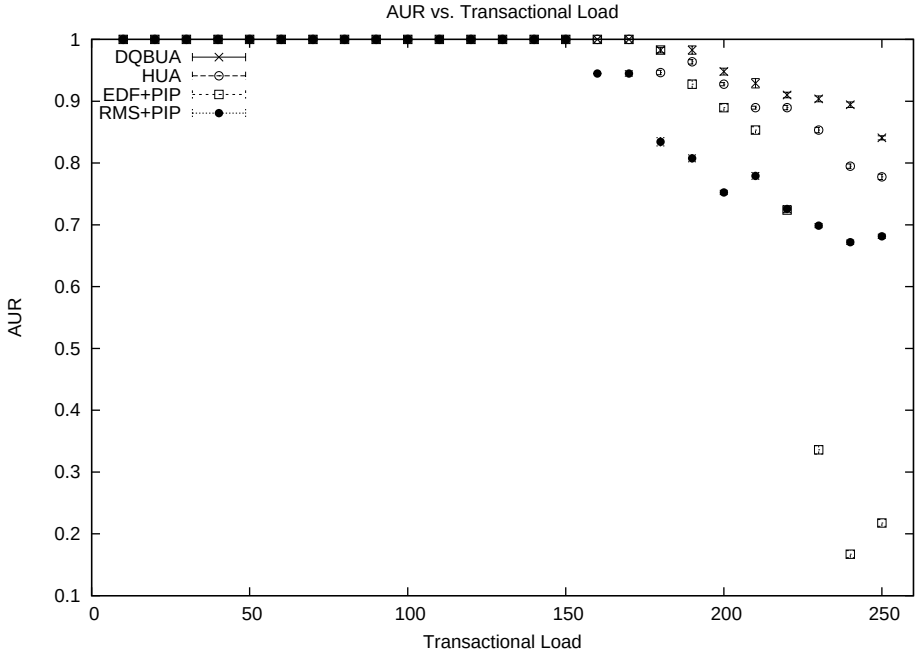

Figure 8.3: AUR vs. Transactional Load (WD, T1, CS=5%, NL=1)

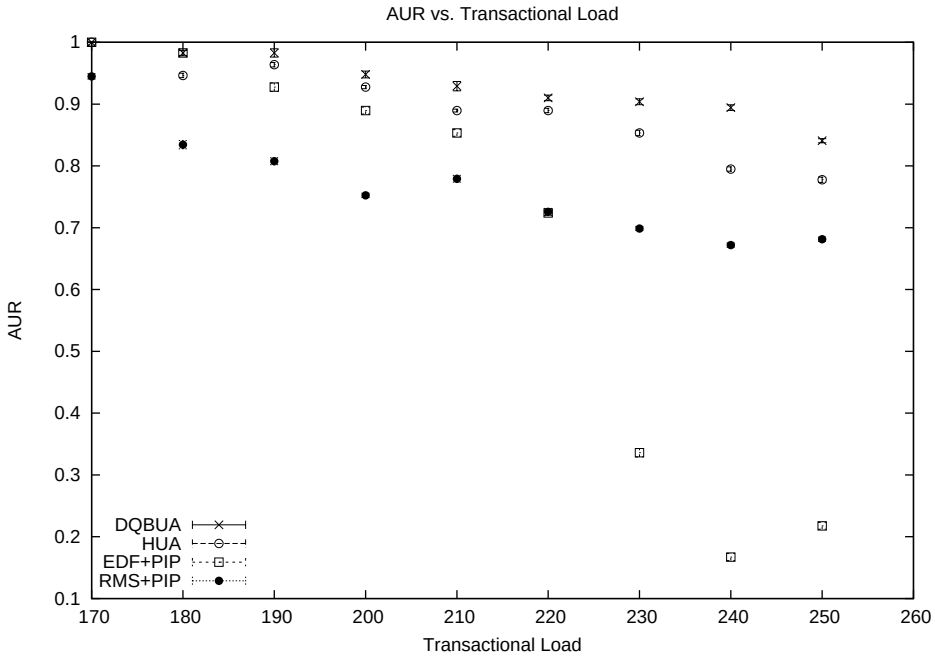

Figure 8.4: AUR vs. Transactional Load (OL, WD, T1, CS=5%, NL=1)

The very large difference in performance between EDF+PIP and the other algorithms is due to the domino effect that is famously common during overloads when EDF is used as the scheduling discipline. Because of this large difference in performance, including EDF+PIP reduces the visibility of the relative performance of the other scheduling algorithms. Therefore, we produce plots that have EDF+PIP removed in order to clarify the relative performance of these algorithms. These plots are presented in Figures 8.5 and 8.6.

RMS+PIP, while not optimal during underload (it begins to miss deadlines before the instantaneous load on any processor exceeds one), is still able to accrue quite a respectable amount of AUR during overloads. This occurs because RMS does not suffer from the domino effect of EDF, but continues to consistently favor threads with shorter periods. Of course, as seen in Section 6.5 of Chapter 6, it is possible to reduce RMS's AUR performance by selecting thread utilities in direct proportion to thread periods. This will cause the algorithm to be implicitly biased against high utility threads. Its DSR results, however, would remain unaffected as it would simply continue to meet the deadline of the same set of threads, but these threads will cease to be the "right" set of threads to favor.

As in Chapter 7, we repeat the experiment with 0.1 overhead failure handlers. The results are presented in Figures 8.7, 8.8, 8.9 and 8.10. As can be seen, DQBUA and HUA now begin to miss deadlines earlier, since they need to reserve scheduling space for the failure handlers – note that EDF+PIP and RMS+PIP do not exhibit this behavior since they only reserve space for the handlers if failures occur and they schedule them. It can also be seen that the larger number of deadline misses offers DQBUA an opportunity to highlight its better

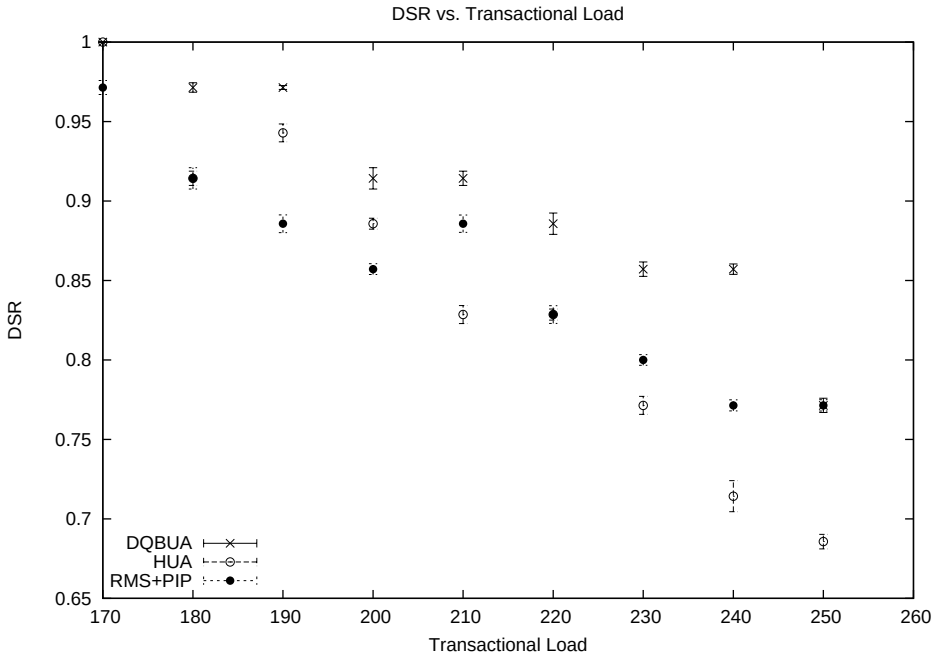

Figure 8.5: DSR vs. Transactional Load (OL, WD, T1, CS=5%, NL=1, no EDF)

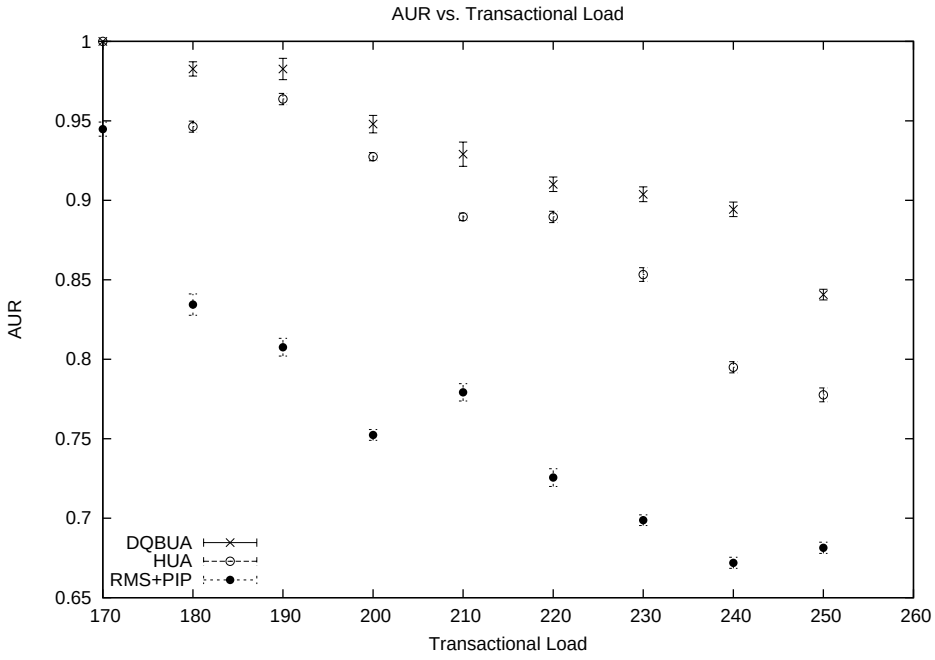

Figure 8.6: AUR vs. Transactional Load (OL, WD, T1, CS=5%, NL=1, no EDF)

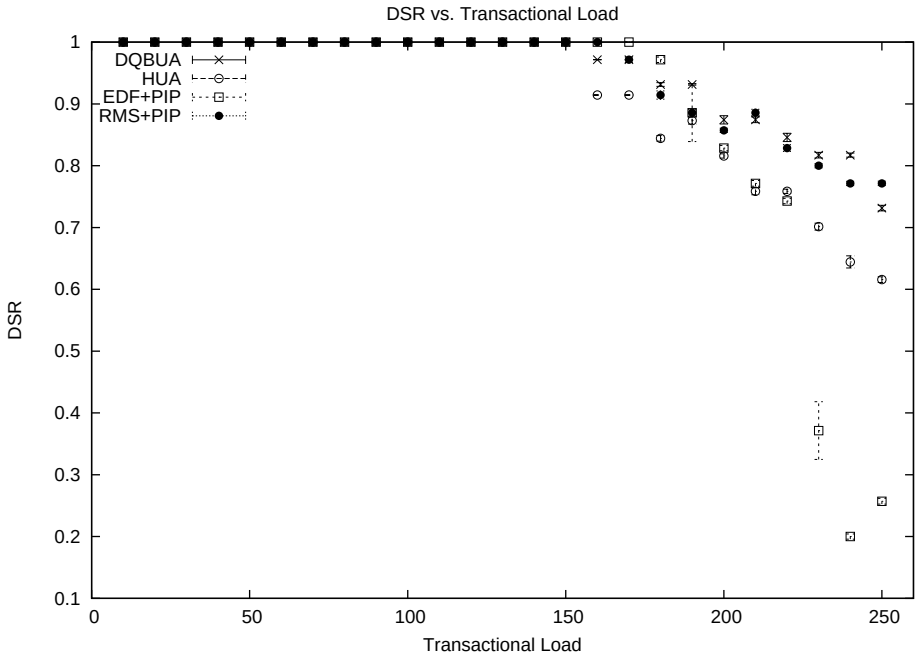

Figure 8.7: DSR vs. Transactional Load (WD, T1, CS=5%, NL=1, HL=0.1)

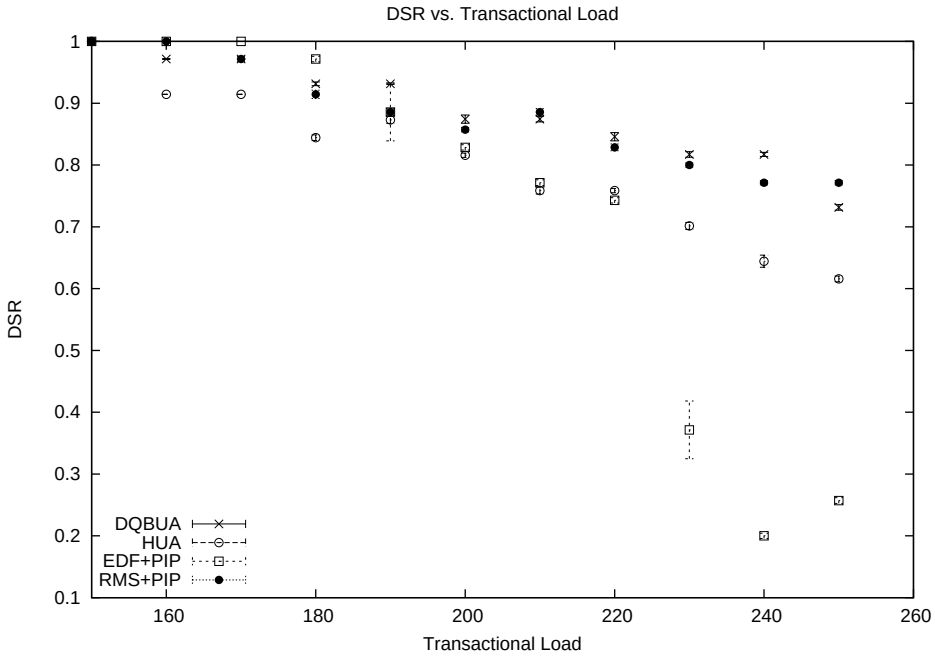

Figure 8.8: DSR vs. Transactional Load (OL, WD, T1, CS=5%, NL=1, HL=0.1)

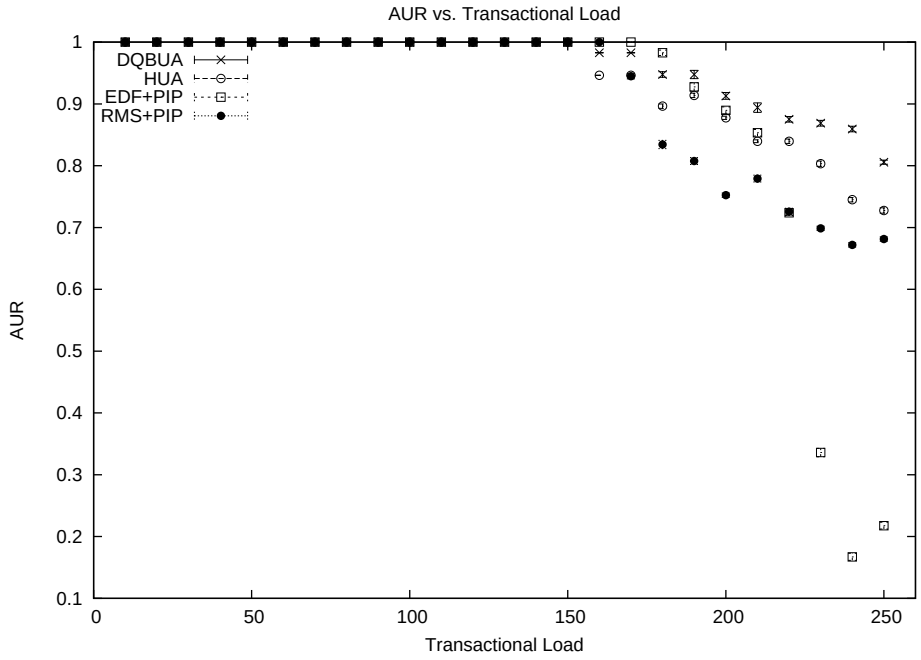

Figure 8.9: AUR vs. Transactional Load, (WD, T1, CS=5%, NL=1, HL=0.1)

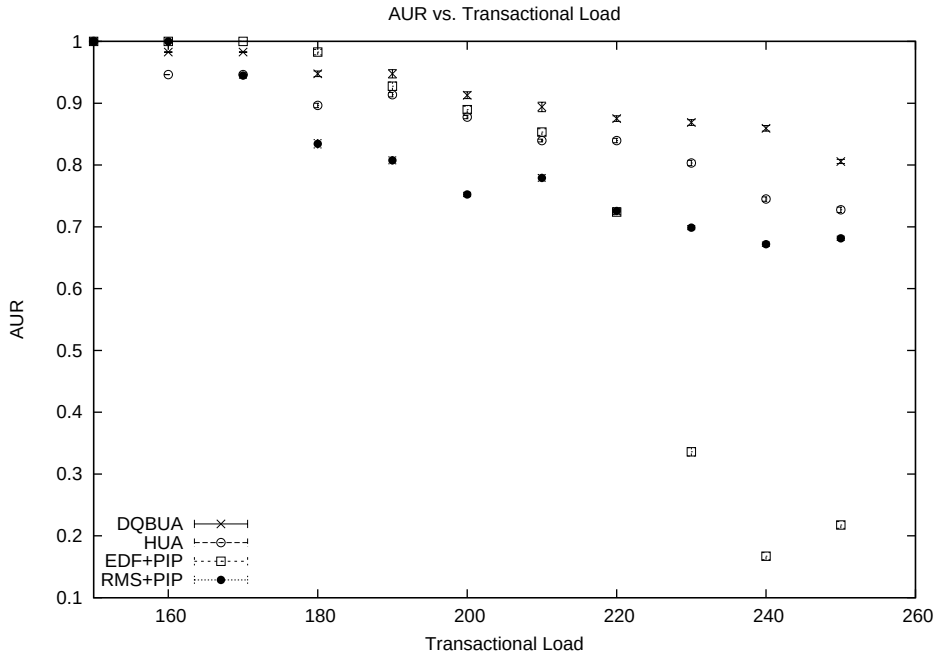

Figure 8.10: AUR vs. Transactional Load (OL, WD, T1, CS=5%, NL=1, HL=0.1)

best-effort property with respect to HUA (since both of them suffer from additional deadline misses in the presence of non-zero overhead handlers). It leads HUA with about 11% at the 240% transactional load point, EDF+PIP with about 69% at the 240% transactional load point and RMS+PIP with about 19% at the 240% transactional load point in AUR.

Now that we have seen the effect of non-zero overhead failure handlers, we do not repeat this experiment in the rest of this chapter. The behavior for other thread sets is the same, and it would needlessly clutter the dissertation. We now turn our attention to increasing the size of critical sections.

Will increasing the size of critical sections affect the results of the scheduling algorithm? Intuitively, increasing the size of critical sections would increase the contention among tasks for the lock held. This occurs because the period during which the lock is held becomes larger, thus making it more likely that another task will request that lock while it is still held. Having smaller critical sections reduces the likelihood of such a scenario occurring.

In order to investigate the effect of this on the scheduling algorithms, we conduct the same experiment above on the same task set twice. Once when the critical section in each section consumes 25% of its execution time and once when the critical section consumes 50% of the section's execution time. The results for the 25% critical sections can be seen in Figures 8.11, 8.12, 8.13 and 8.14.

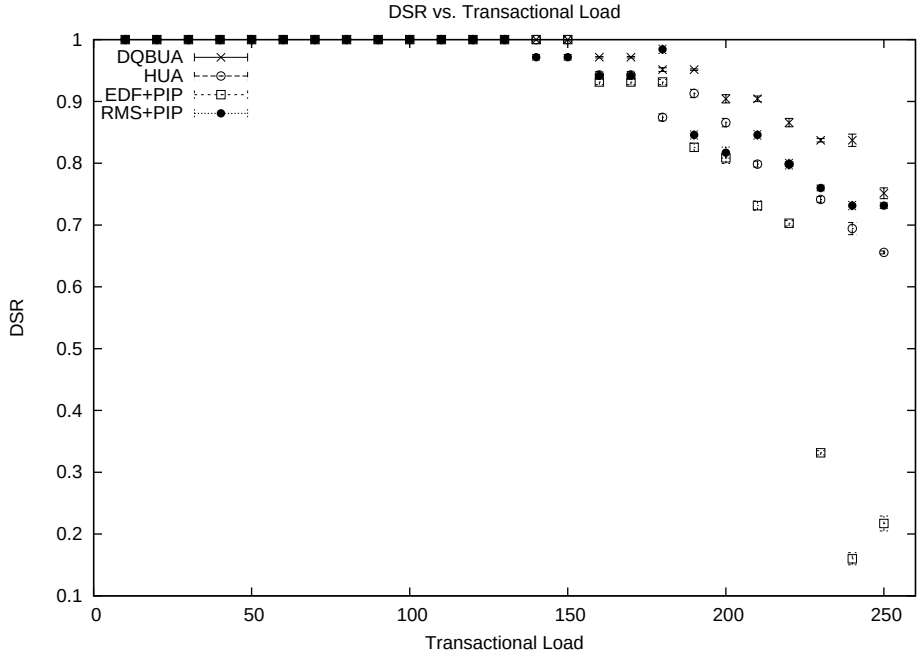

Figure 8.11: DSR vs. Transactional Load (WD, T1, CS=25%, NL=1)

As expected, increasing the "surface area" of the region of each section that can result in locking conflicts increases the contention on the lock and hence increases the number of

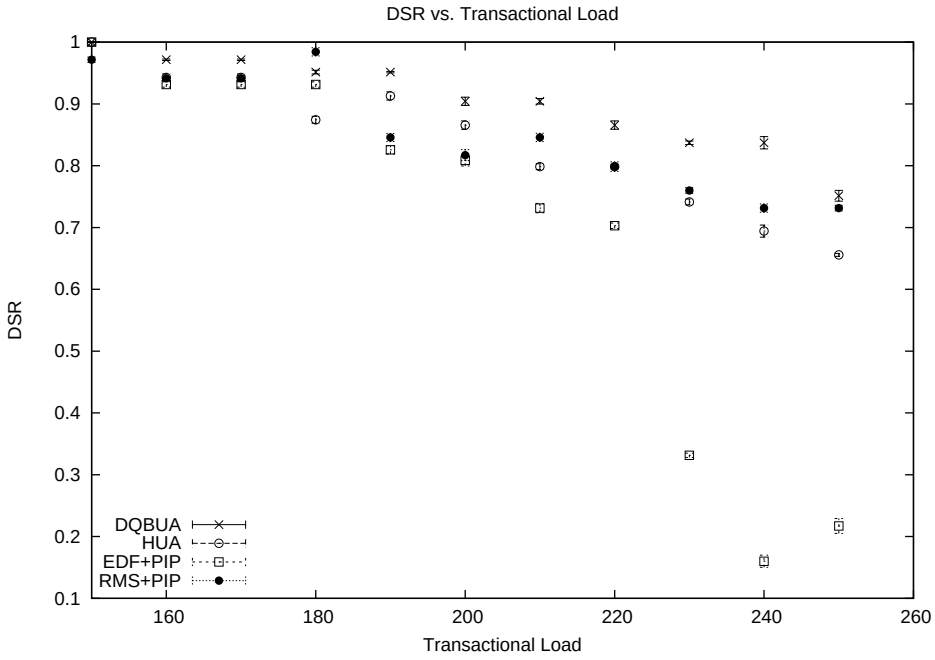

Figure 8.12: DSR vs. Transactional Load (OL, WD, T1, CS=25%, NL=1)

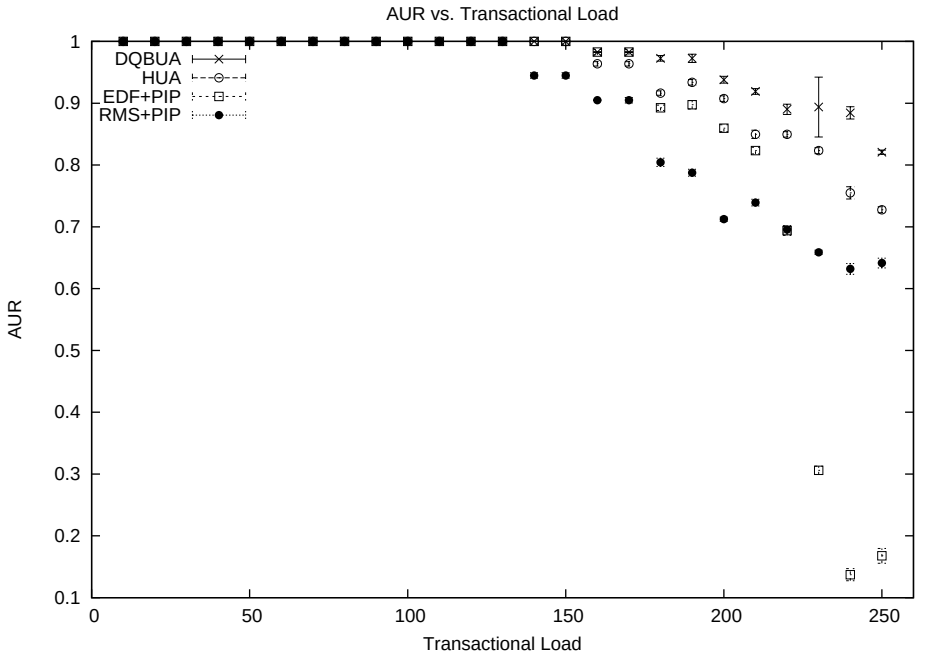

Figure 8.13: AUR vs. Transactional Load (WD, T1, CS=25%, NL=1)

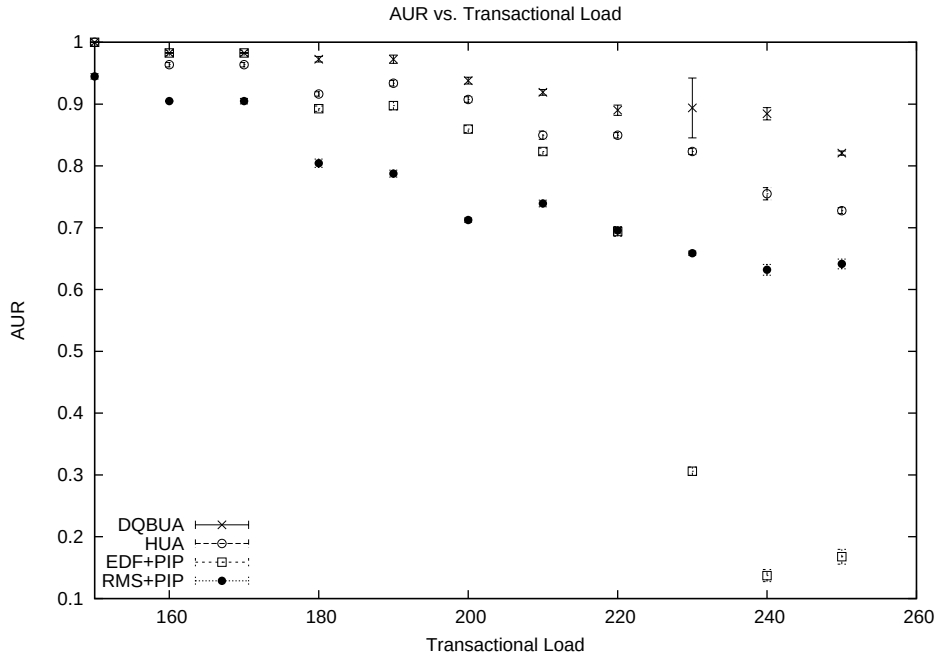

Figure 8.14: AUR vs. Transactional Load (OL, WD, T1, CS=25%, NL=1)

threads that miss their deadline. The maximum difference between the algorithms occurs at the 240% transactional load point. DQBUA leads HUA with 13% AUR, EDF+PIP with 75% AUR and RMS+PIP with 25% AUR.

In order to conserve space, we do not reproduce the entire plot(s) for the 50% critical section. Instead, we choose the 240% transactional load point and plot the AUR for that point for 5% WCET critical sections, 25% WCET critical sections and 50% WCET critical sections for algorithms we are comparing. The results are depicted in Figures 8.15 and 8.16. The maximum difference between the algorithms occur for the the 50% critical section, with AUR leads of 16%, 28% and 75% for HUA, RMS+PIP and EDF+PIP respectively.

As can be seen, the AUR drops as we increase the size of the critical section since this increases the contention for the lock and hence causes the system to shed more tasks. Note, again, that this offers DQBUA an opportunity for show-casing its better best-effort property.

By increasing the number of tasks being shed, this scenario allows DQBUA to differentiate itself from the other algorithms by choosing the "right" set of threads to remove from the system. Allowing it to gracefully degrade as more tasks are shed from the system. So, while all algorithms suffer a loss in AUR from the increased likelihood of contention, DQBUA suffers the least relative decline in AUR because it is able to choose the "right" set of tasks to shed from the system.

Note also that EDF+PIP has the worst performance in this scenario. As a matter of fact, EDF+PIP's small numbers in this plot distort the scale and minimize the visibility of the

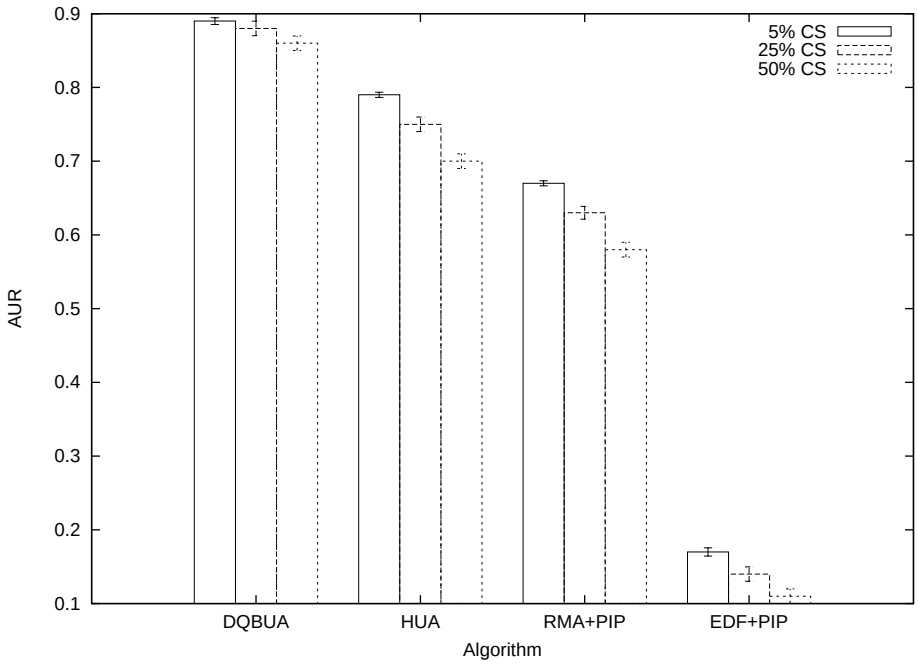

Figure 8.15: AUR vs. Size of CS (WD, 240% transactional load point)

differences between the other algorithms and the drops in AUR they accrue. Therefore, we produce another plot, Figure 8.16, without EDF+PIP to remove this distortion.

One interesting point is how the algorithms handle this additional pressure to shed threads. In most of the experiments we conducted, RMS and HUA shed tasks at a relatively similar rate. Of course, HUA attempts to maximize AUR while RMS doesn't so although they shed tasks at similar rates, HUA ends up with more AUR than RMS.

Interestingly, EDF+PIP sheds tasks, i.e., misses task deadlines, at a lower rate than HUA and RMS in most of the experiments we conducted. We believe that this is due to the fact that EDF's performance is already dominated by the domino effect during overloads and so the addition of more sources of overhead, such as increasing the duration of time when lock contention occurs, does not significantly contribute to the loss of deadlines.

DQBUA outperforms the rest of the algorithms by, as mentioned before, taking advantage of the requirement of shedding more tasks to handle overloads to select the "right" set of threads to shed on a system-wide basis. This allows it to come out of the process ahead of the other scheduling algorithms in terms of AUR accrued.

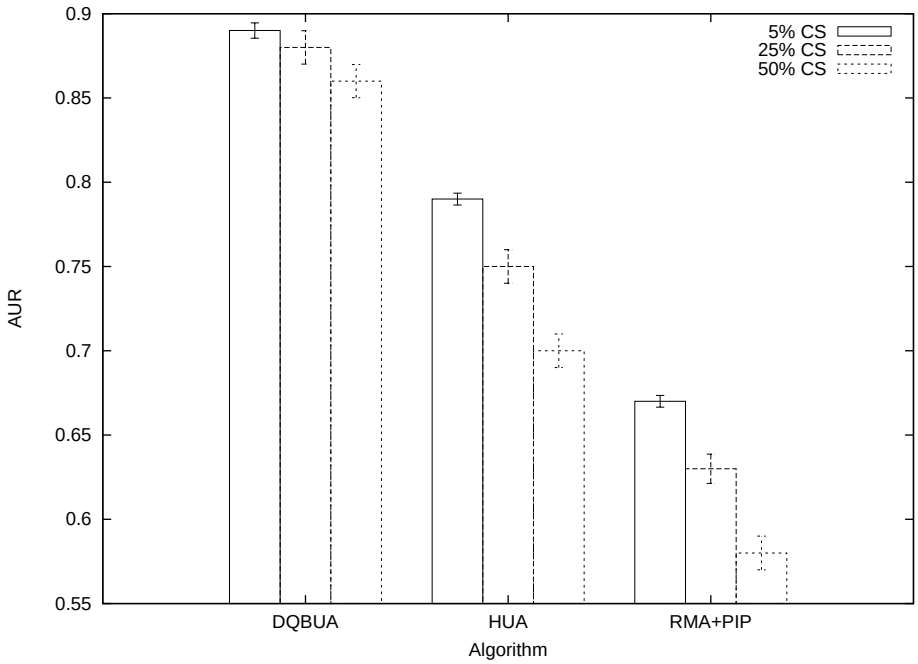

Figure 8.16: AUR vs. Size of CS (WD, 240% transactional load point, no EDF)

8.2.2 Multiple Locks

In this section, we investigate the effect of having more than one lock on the scheduling algorithms. Towards that end, we perform two experiments, one with 5 locks and the other with 10 locks. For these experiments, we fix the duration of the critical section to 40% of WCET of its section. The locks are nested. The duration of the critical section is divided equally among the levels of the nested critical sections. The order of lock acquisition is not pre-determined and so deadlocks are possible.

Intuitively, increasing the number of resources being requested increases the interaction of threads in the system which increases the overhead and distorts the schedule. Therefore, this results in more threads being shed from the system. Figures 8.17, 8.18, 8.19 and 8.20 show the results of this experiment. The maximum difference in AUR between DQBUA and HUA is now about 14% at the 230% transactional load point. This is also the same point that the maximum difference, 27%, between DQBUA and RMS occurs. The maximum difference between DQBUA and EDF+PIP occurs at the 240% transactional load point and is equal to 68%.

Next, we conduct the same experiment using 10 locks. Again, the duration of the entire critical section is distributed equally among the nested critical sections. Figures 8.21, 8.22, 8.23 and 8.24 depict the result of this experiment. As can be seen, increasing the number of locks per critical section increases the chance for lock contention and hence reduces both AUR

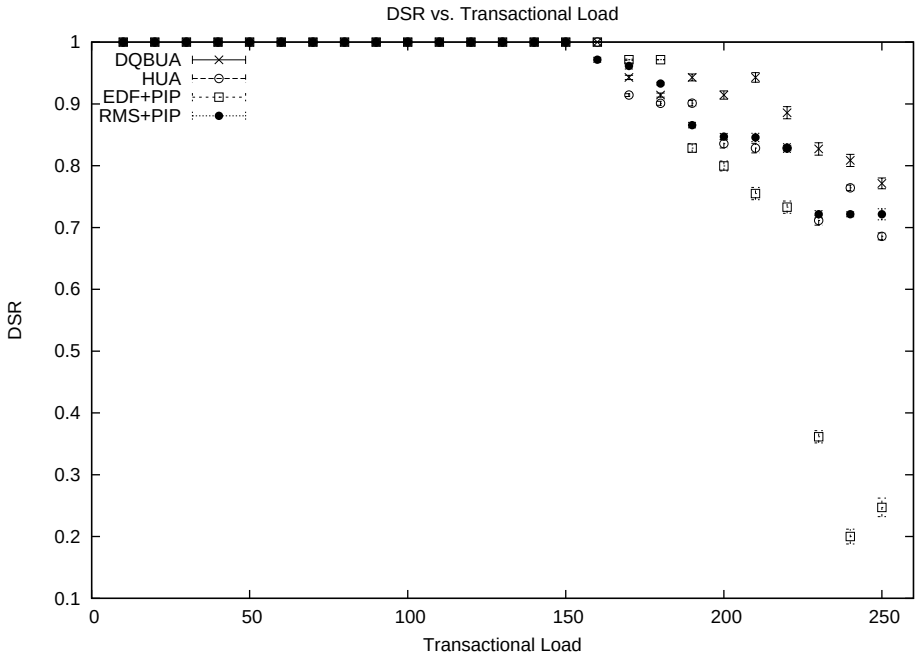

Figure 8.17: DSR vs. Transactional Load (WD, T1, CS=40%, NL=5)

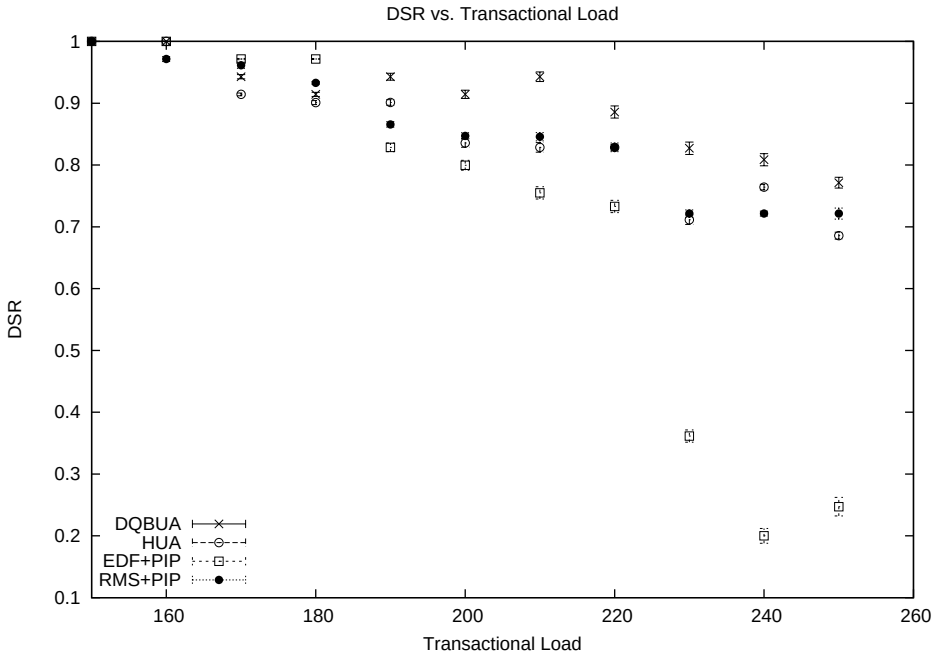

Figure 8.18: DSR vs. Transactional Load (OL, WD, T1, CS=40%, NL=5)

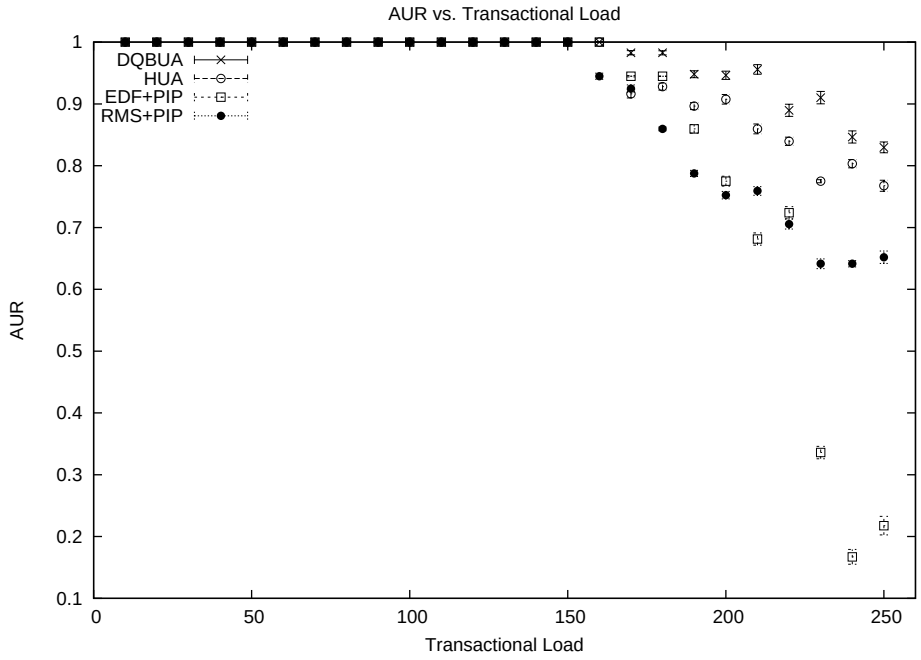

Figure 8.19: AUR vs. Transactional Load, (WD, T1, CS=40%, NL=5)

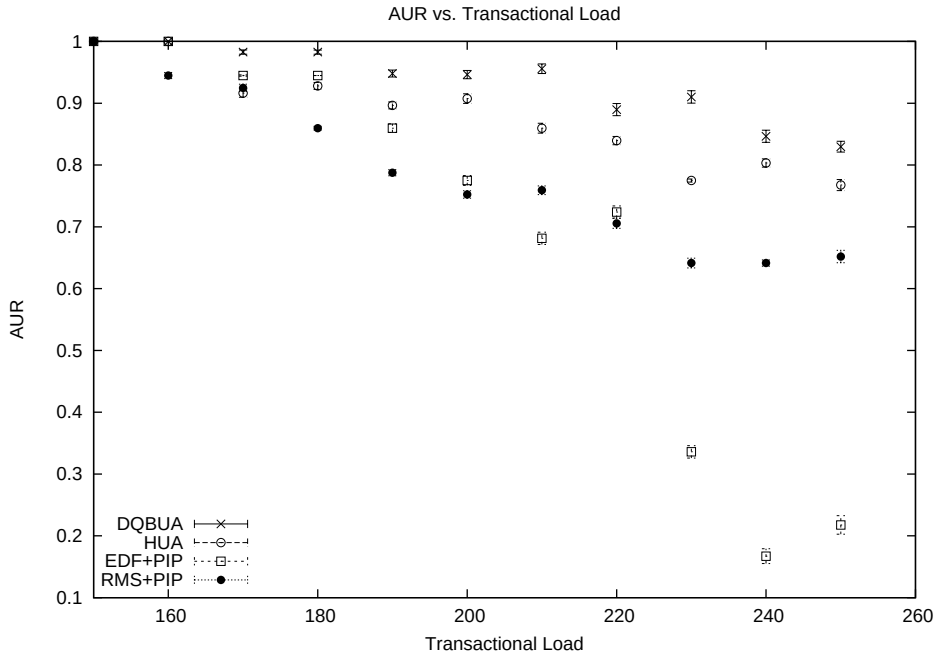

Figure 8.20: AUR vs. Transactional Load (OL, WD, T1, CS=40%, NL=5)

and DSR. It also causes the system to miss deadlines earlier on than when 5 locks are used. DQBUA performs better than HUA by a maximum of 14% at the 230% transactional load point, outperforms EDF+PIP by 71% at the 240% transactional load point and RMS+PIP by 24% at the 210% transactional load point.

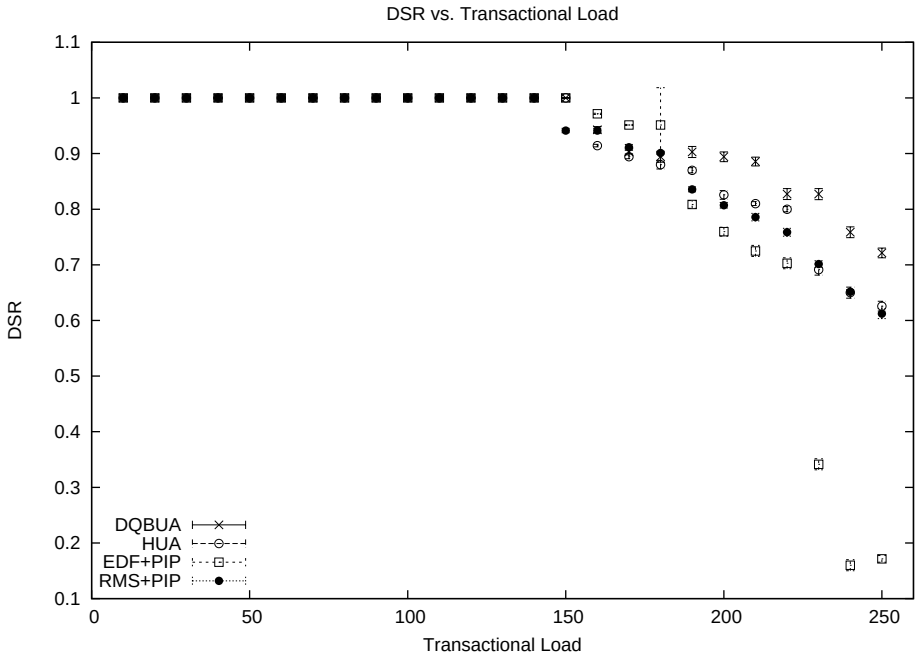

Figure 8.21: DSR vs. Transactional Load (WD, T1, CS=40%, NL=10)

8.2.3 Distributed Dependencies

In the previous experiments, each section released its locks before a remote invocation was made. Therefore, all lock contention was confined to local nodes. In this section, we investigate the effect of distributed dependencies on the scheduling algorithm. Toward that end, we consider the case of single lock critical sections that have 10% of WCET durations. Before a lock is released, a remote invocation is made to another node. Thus, the lock is held while the thread is executing remotely at another node.

We conduct this experiment on the thread set described in Table 7.2, with the Class A method of WCET assignment. The results are depicted in Figures 8.25, 8.26, 8.27 and 8.28. As can be seen, the difference in performance between DQBUA and all other algorithms becomes more pronounced along all transactional load points. This is due to the fact that it is the only algorithm that considers distributable dependencies. It can also be noted that deadlines are missed at an earlier transactional point (150% instead of 180%) this is because distributable dependencies cause much larger interaction between tasks than local dependencies.

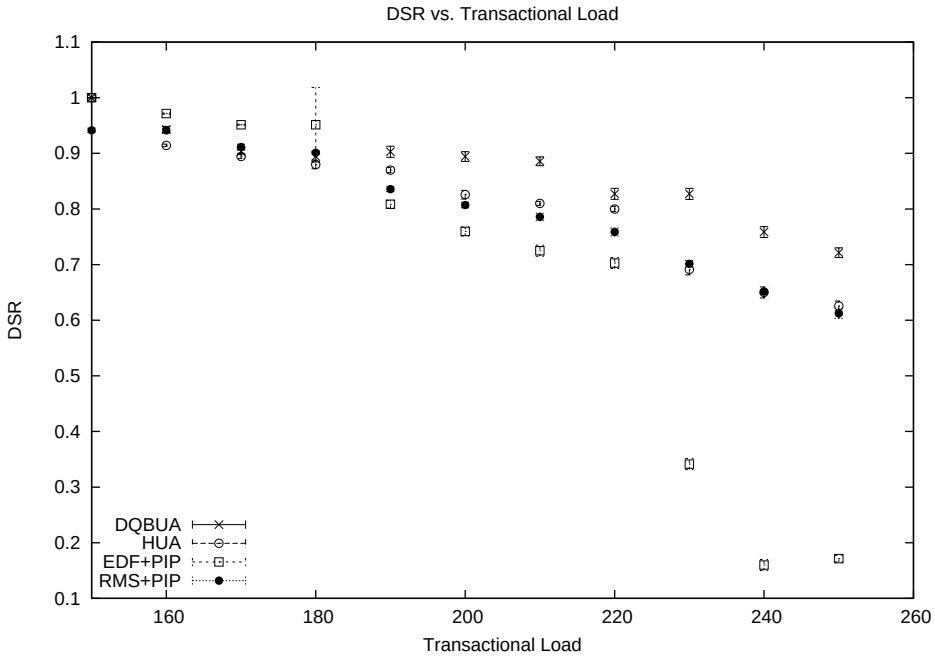

Figure 8.22: DSR vs. Transactional Load (OL, WD, T1, CS=40%, NL=10)

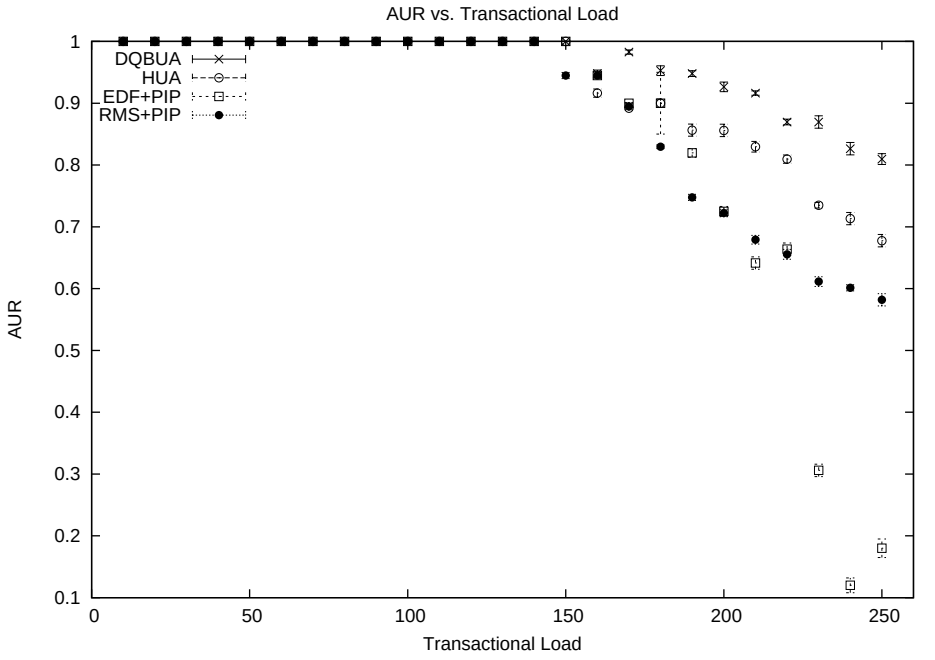

Figure 8.23: AUR vs. Transactional Load, (WD, T1, CS=40%, NL=10)

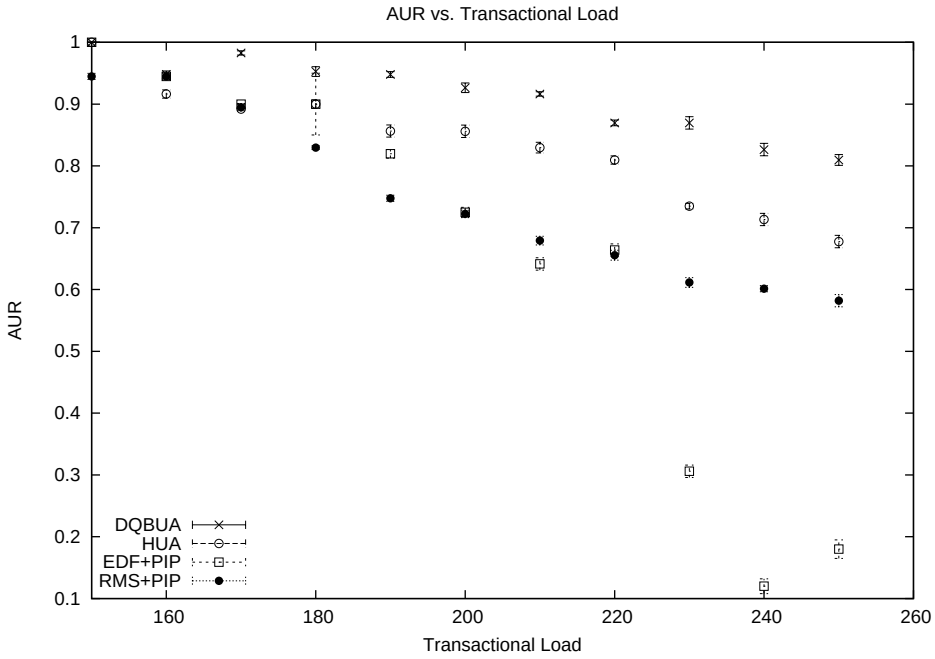

Figure 8.24: AUR vs. Transactional Load (OL, WD, T1, CS=40%, NL=10)

The maximum difference between DQBUA and HUA is 15% at the 240% transactional load point, the maximum different between DQBUA and EDF+PIP is 70% at the 240% transactional load point, while the maximum difference between HUA and RMS+PIP is 27% at the 230% transactional load point. Note that while all algorithms shed more deadlines, the least affected algorithm, at least at the upper range of transactional load points, is EDF+PIP. This occurs because its performance in that range is already dominated by the domino effect that causes it to loose a huge amount of deadlines during overloads. Therefore, adding another source of overhead at this stage does not significantly affect the algorithm's performance.

Next, we consider the case of distributed dependencies with more than one lock. We compare the results for the four competitor algorithms we consider for critical sections with 1 lock, 5 locks and 10 locks where distributed dependencies can occur (i.e., where we make remote invocations while insider critical sections). The result of this experiment is shown in Figures 8.29 and 8.30. As can be seen, while all algorithms shed more tasks as the increasing number of locks cause more interaction between the tasks, DQBUA looses less AUR because it selects the "right" set of tasks to eliminate from the system.

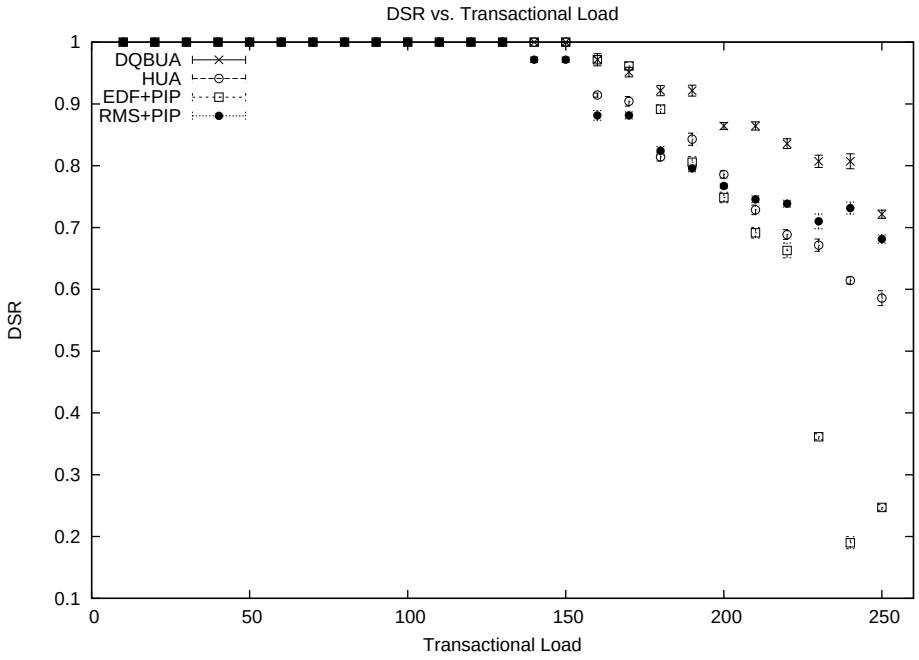

Figure 8.25: DSR vs. Transactional Load (WD, T1, CS=10%, NL=1, DD)

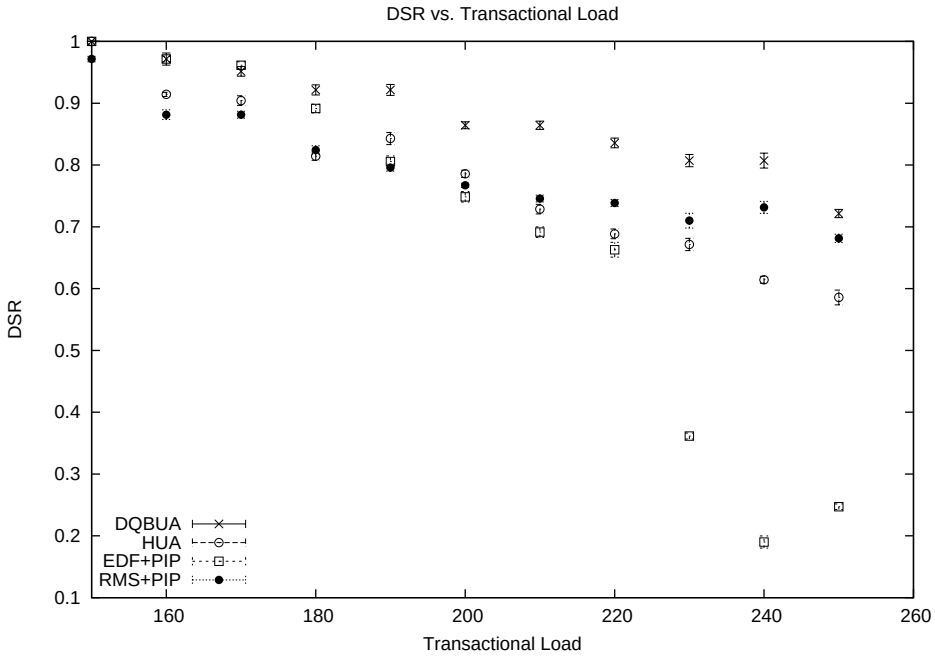

Figure 8.26: DSR vs. Transactional Load (OL, WD, T1, CS=10%, NL=1, DD)

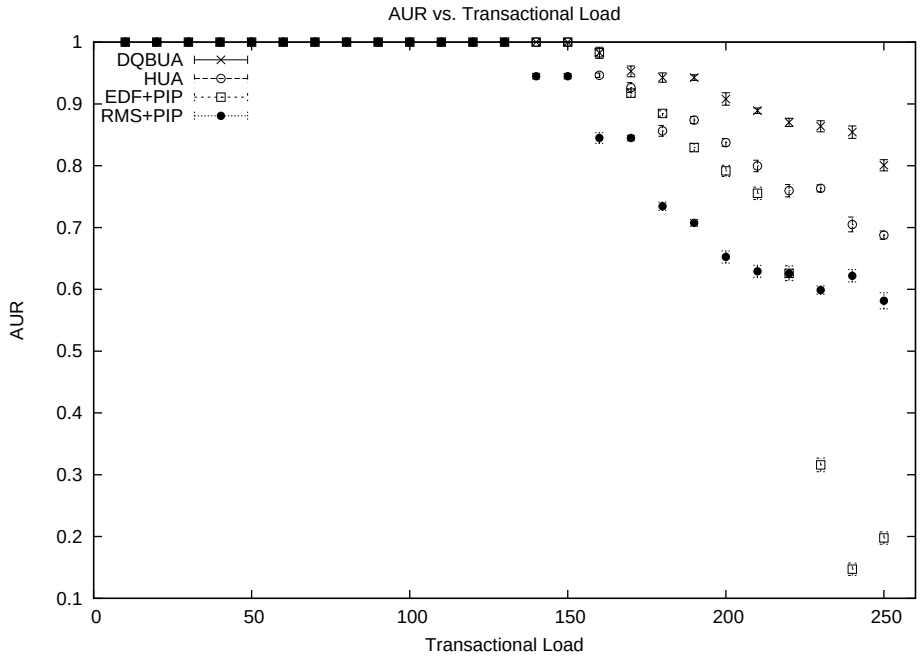

Figure 8.27: AUR vs. Transactional Load (WD, T1, CS=10%, NL=1, DD)

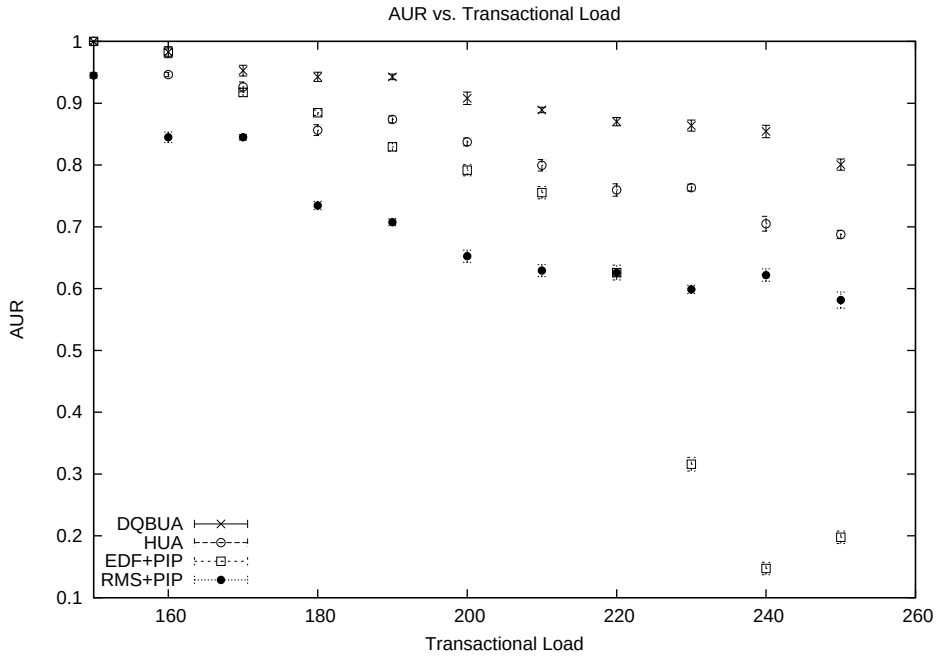

Figure 8.28: AUR vs. Transactional Load (OL, WD, T1, CS=10%, NL=1, DD)

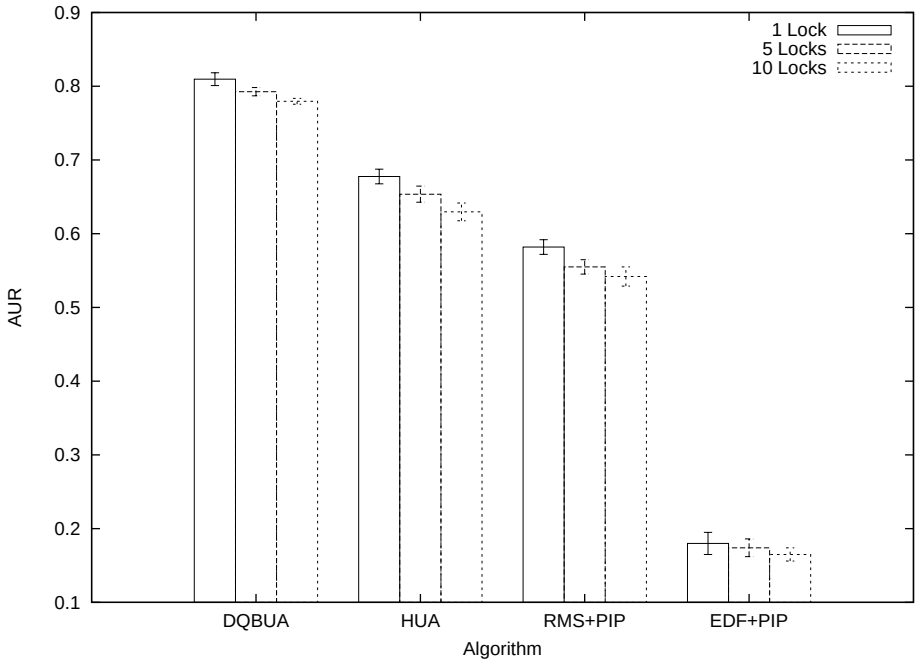

Figure 8.29: AUR vs. No. of Locks (WD, T1, CS=40%, DD)

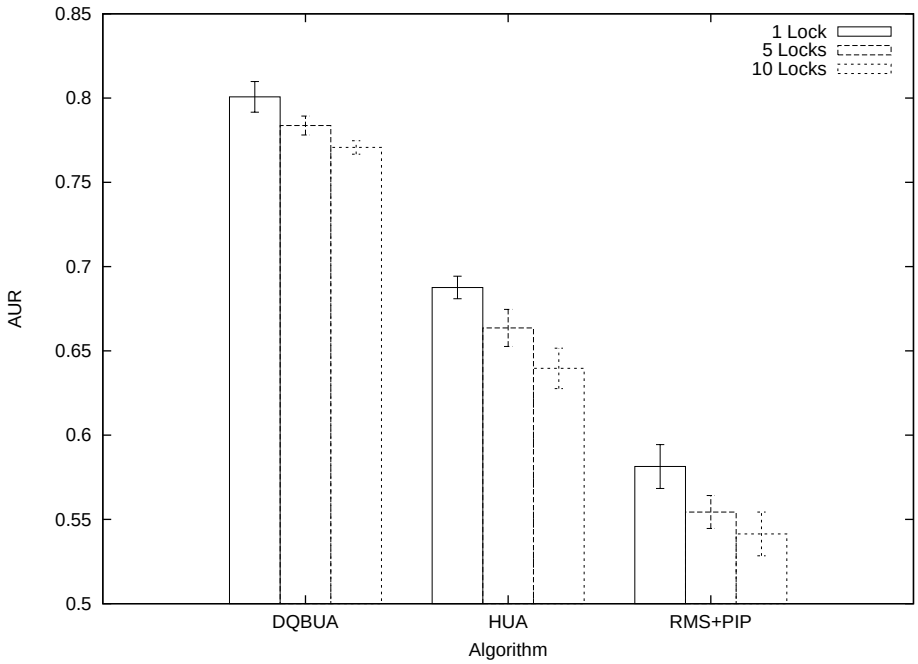

Figure 8.30: AUR vs. No. of Locks (WD, T1, CS=40%, DD, no EDF)

8.2.4 Comments on Deadlock Detection Overheads

Both the collaborative and independent scheduling algorithms we consider have deadlock detection and resolution capabilities. We provide this capability by detecting cycles in the resource graph and terminating the thread with the least PUD in such a cycle if it is detected.

The custom system call we wrote to allow threads to acquire locks invokes the scheduler (whether collaborative or independent) after indicating that it wants a specific lock. Starting from that lock, a dependency chain is constructed until either the next dependency is NULL or we arrive at the starting task. The former indicates that no deadlock has occurred, while the latter implies that a deadlock exists. At which point the task with the least PUD is terminated. To prevent reiteration over the dependency graph to choose this thread, a variable is used to keep track of the lowest PUD task as we traverse the resource graph to detect possible deadlock.

What this implies is that the overhead of deadlock detection is borne by all threads in the system, even those that are designed (by ordering lock acquires, for example) to prevent deadlock. The only difference between the case where deadlock is detected and deadlock is not detected is that the last step in the former is coming back to the starting task, while the last step of the latter is detecting NULL as the next task in the resource graph. Another difference is that when deadlock is detected, more tasks are shed in order to resolve the deadlock than is no deadlock is detected. The overhead of DQBUA, which includes the code path that checks for deadlocks, is described in Section 8.5.

8.3 Proportional Slack

8.3.1 Singe Lock

In this section, we report the results for the proportional slack method of deadline decomposition. We conduct the experiment on the thread set described in Table 7.2. Figures 8.31, 8.32, 8.33 and 8.34 depict the result.

As can be seen, the general trend in the relative behavior of the algorithms does not change for the proportional slack deadline decomposition method. There is, however, an improvement observed in the performance of the case two scheduling algorithms. This occurs because by using proportional slack decomposition, sections that cannot meet their deadlines are stopped earlier (as compared to the worst-case deadline decomposition method). This means that the system can detect whether a DT will be able to meet its end-to-end deadline earlier, remove threads that cannot meet their end-to-end sooner than the worst-case deadline decomposition method and thereby take advantage of the resulting freed processor time to service threads that can actually meet their end-to-end deadline.

For this method of deadline decomposition, DQBUA outperforms HUA by a maximum of 10% at the 240% transactional load point, outperforms EDF+PIP by a maximum of 42% at the 250% transactional load point and outperforms RMA+PIP by 18% at the 240% transactional load point. As can be seen, the difference between EDF and DQBUA narrows down to 42% from seventy something percent when moving to proportional slack from worst-case deadline decomposition. As mentioned previously, this is due to the fact that proportional slack gives earlier threads a proper idea of what slack is available for them so that they do not encroach on the slack of subsequent sections. When they discover that they will miss their assigned deadline, the thread is terminated early and thus the processing time freed up is used to execute other threads that can actually meet their end-to-end deadline.

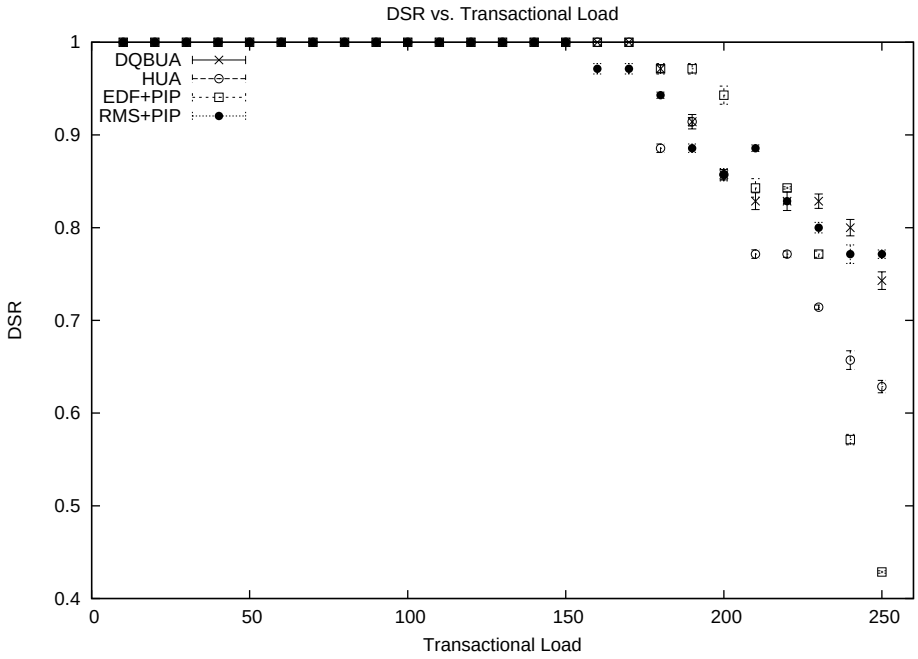

Figure 8.31: DSR vs. Transactional Load (PS, T1, CS=5%, NL=1)

Next, we investigate the performance of increasing critical section length on the performance of the algorithms. Towards that end, we plot the AUR accrued at the 240% transactional load point for all algorithms when critical section duration is 5%, 25% and 50% of WCET of its section. Figures 8.35 and 8.36 show the results for this experiment.

As can be seen, the AUR drops as we increase the size of the critical section since this increases the contention for the lock and hence causes the system to shed more tasks. Note, again, that this offers DQBUA an opportunity for show-casing its better best-effort property.

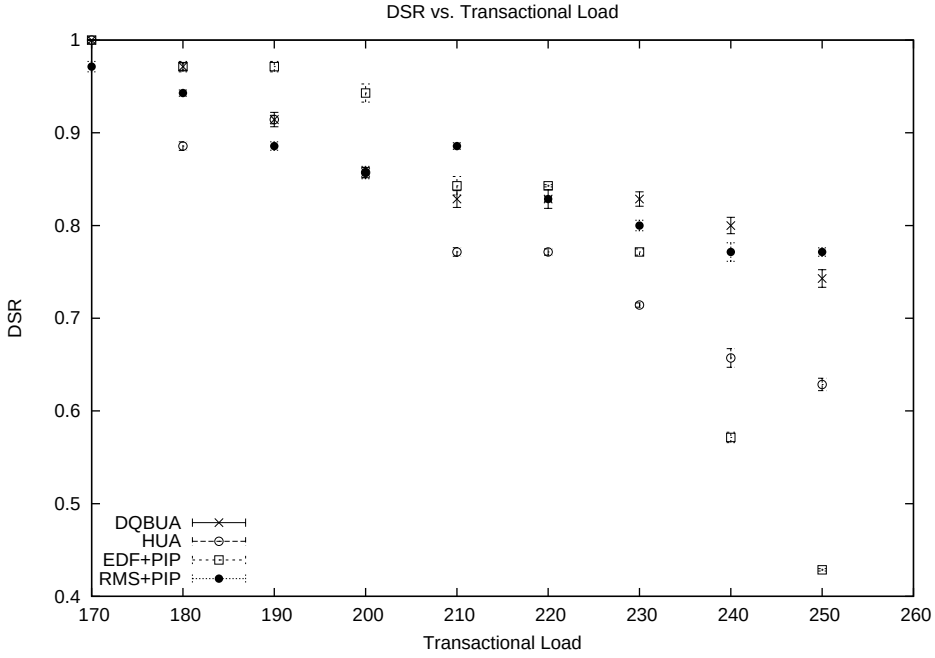

Figure 8.32: DSR vs. Transactional Load (OL, PS, T1, CS=5%, NL=1)

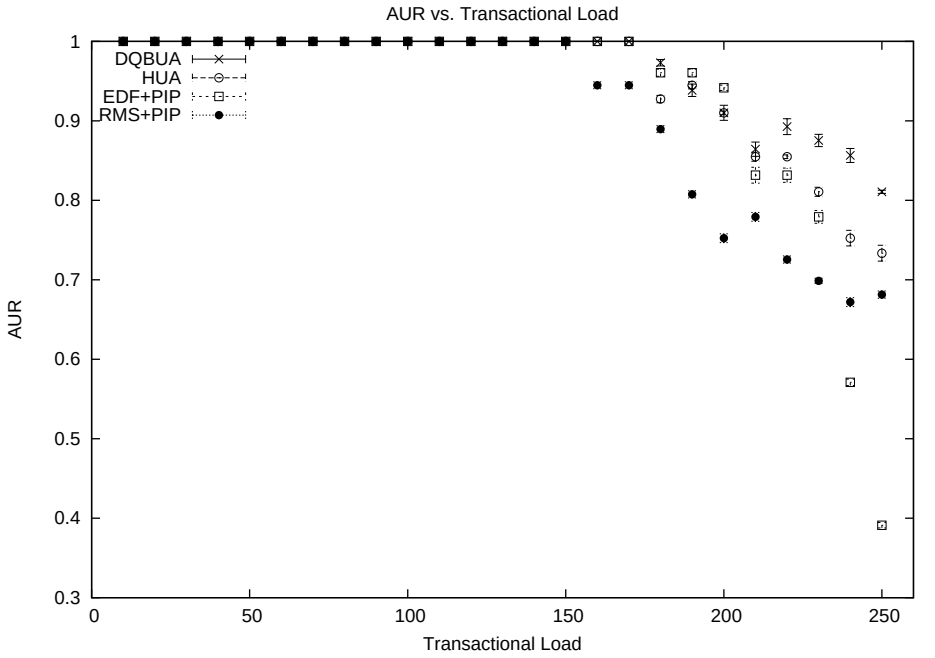

Figure 8.33: AUR vs. Transactional Load (PS, T1, CS=5%, NL=1)

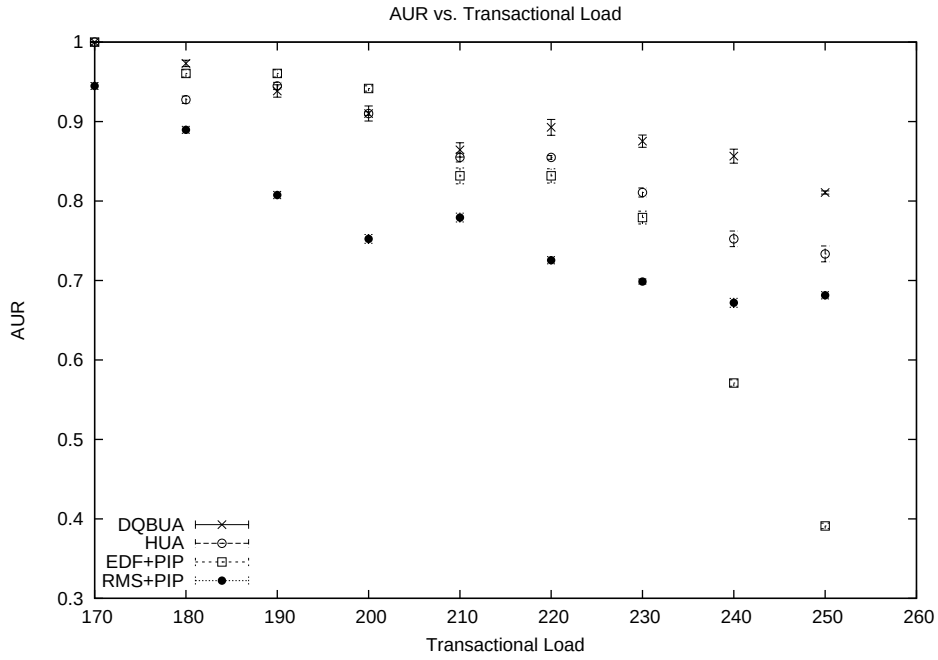

Figure 8.34: AUR vs. Transactional Load (OL, PS, T1, CS=5%, NL=1)

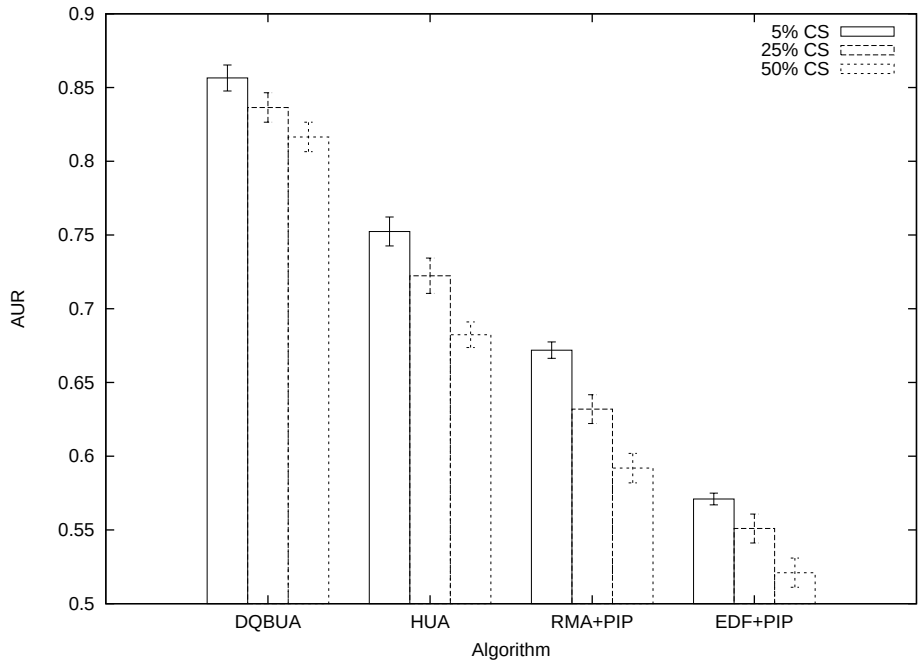

Figure 8.35: AUR vs. Size of CS (PS, 240% transactional load point)

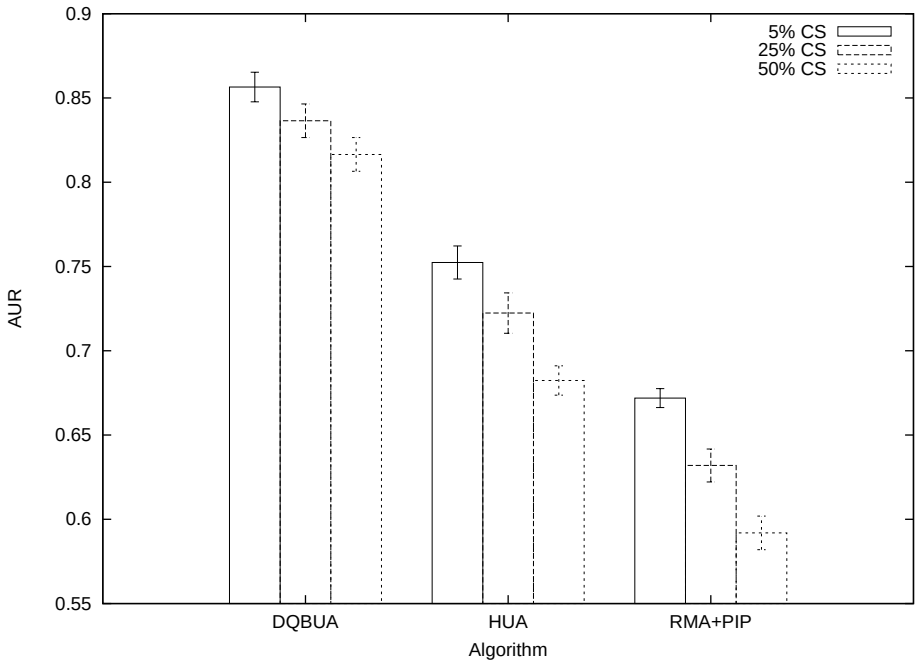

Figure 8.36: AUR vs. Size of CS (PS, 240% transactional load point, no EDF)

8.3.2 Multiple Locks

In this section, we perform multiple locks experiments on the proportional slack decomposition method. In this experiment, each thread has ten nested critical sections. The entire critical section is assigned 40% of the WCET of its section as its duration. This duration is equally divided among the nested critical sections. The experiment is performed using the thread set described in Table 7.2 using the Class A method for WCET assignment. The results are depicted in Figures 8.37, 8.38, 8.39 and 8.40.

In this experiment, the maximum difference between HUA and DQBUA is 15% at the 230% transactional load point, outperforms EDF+PIP by a maximum of 43% at the 250% transactional load point, and outperforms RMS+PIP at the 230% transactional load point by about 23%. All results are reported in terms of AUR.

8.3.3 Distributed Dependencies

In this section, we investigate the effect of distributed dependencies on scheduling performance. Towards this end, we perform an experiment using the task set described in Table 7.2. Each section has one critical section that consumes 10% of its WCET. While still holding the lock, a remote invocation is made, thus introducing remote dependencies. The results of this experiment are depicted in Figures 8.41, 8.42, 8.43 and 8.44.

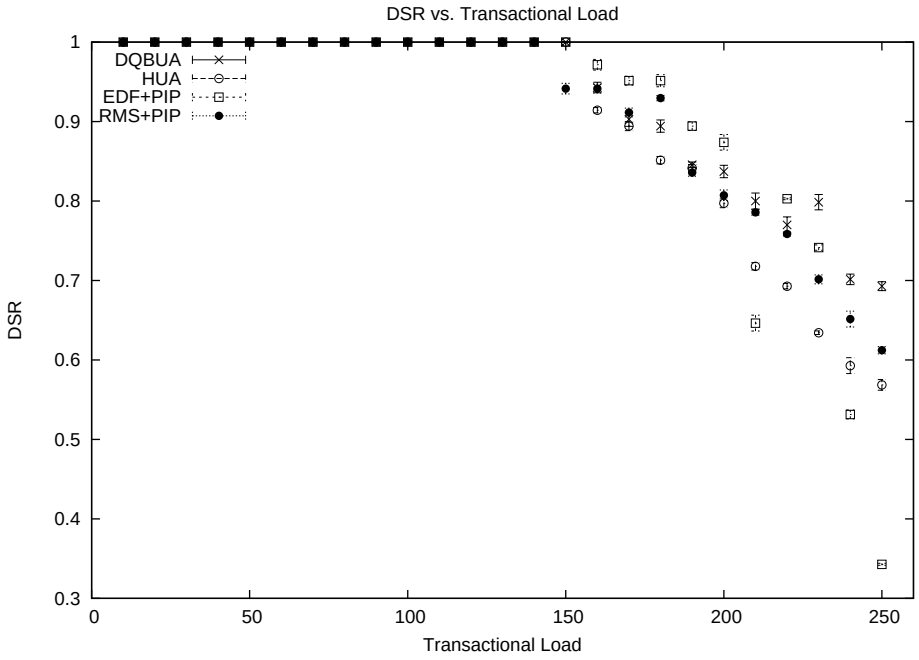

Figure 8.37: DSR vs. Transactional Load (PS, T1, CS=40%, NL=10)

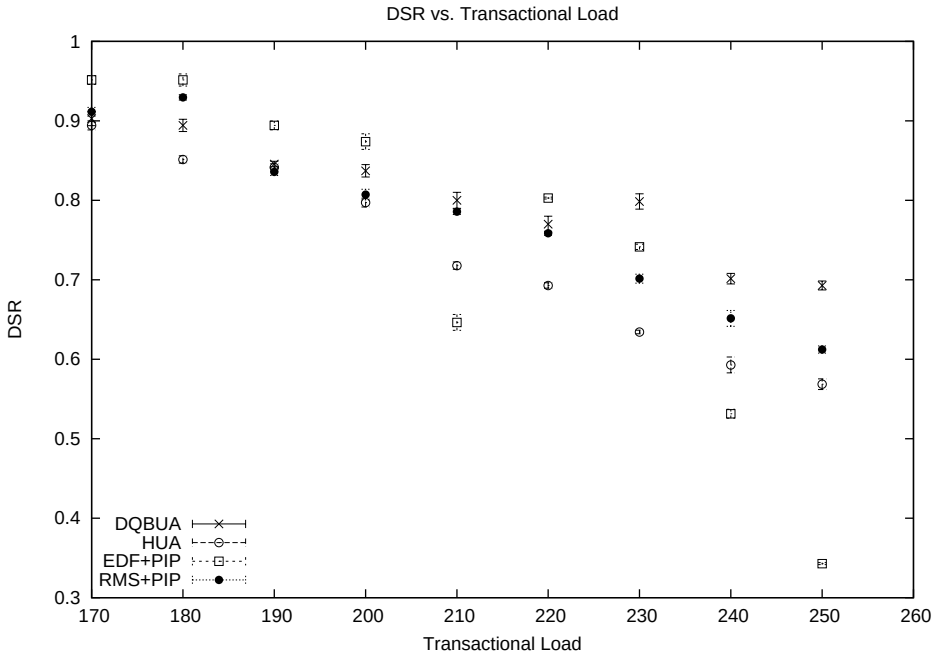

Figure 8.38: DSR vs. Transactional Load (OL, PS, T1, CS=40%, NL=10)

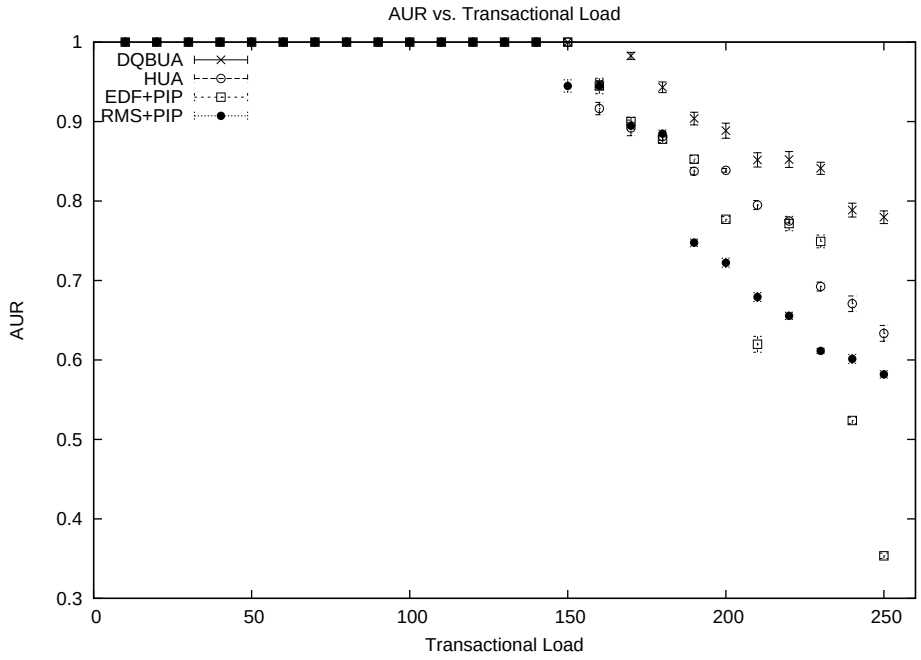

Figure 8.39: AUR vs. Transactional Load (PS, T1, CS=40%, NL=10)

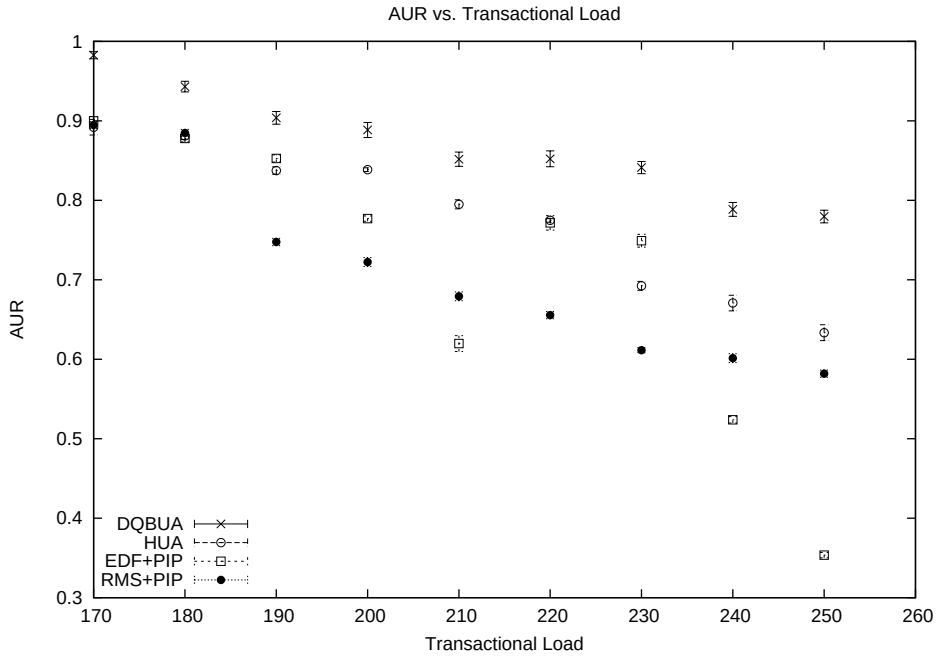

Figure 8.40: AUR vs. Transactional Load (OL, PS, T1, CS=40%, NL=10)

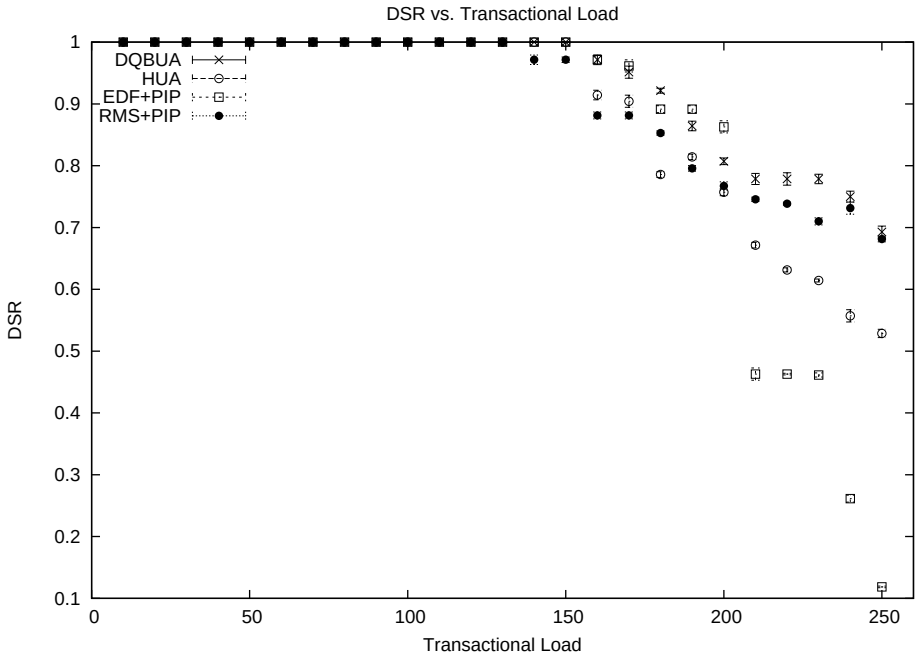

Figure 8.41: DSR vs. Transactional Load (PS, T1, CS=10%, NL=1, DD)

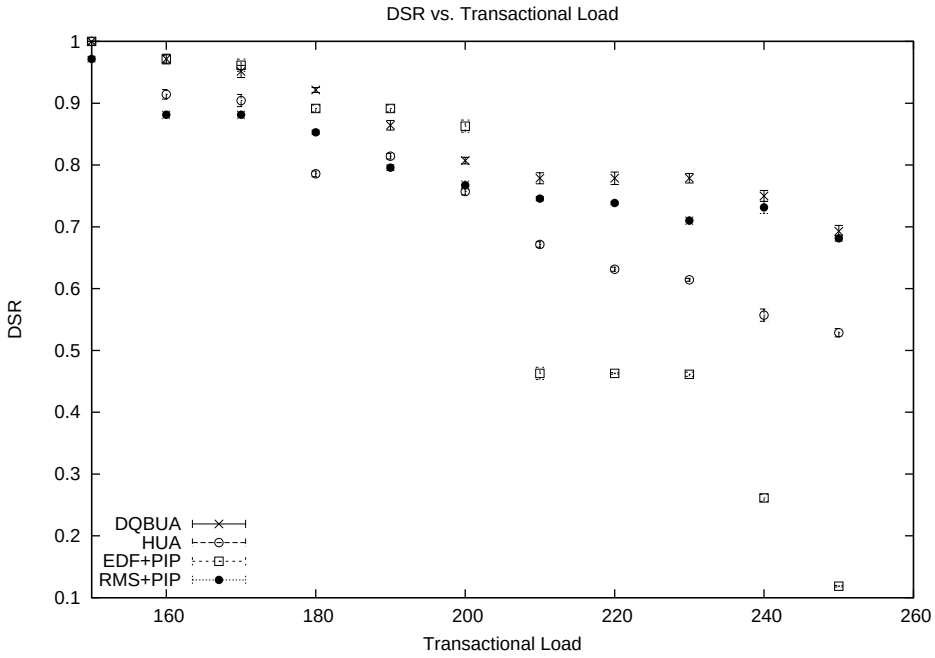

Figure 8.42: DSR vs. Transactional Load (OL, PS, T1, CS=10%, NL=1, DD)

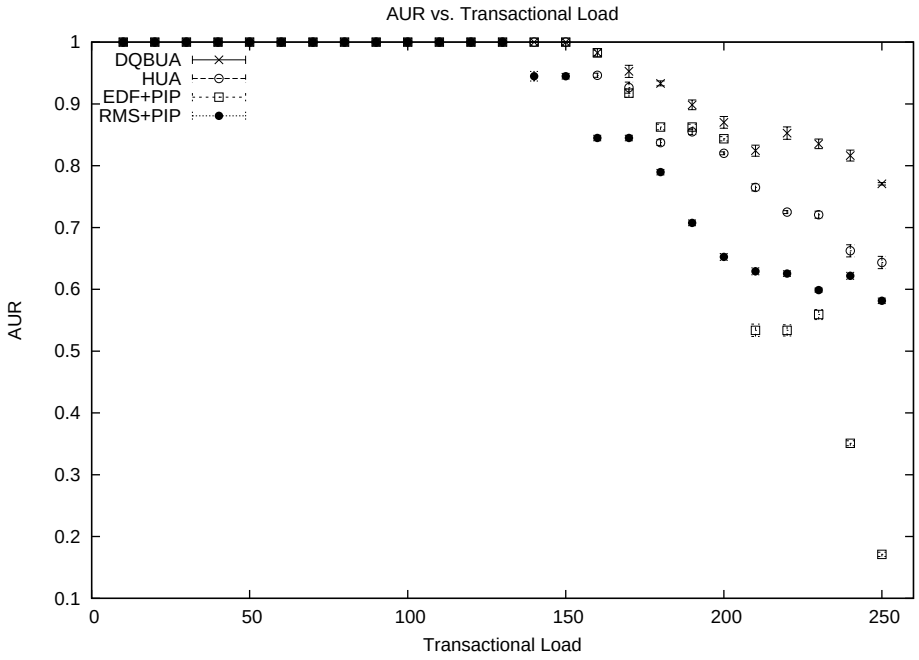

Figure 8.43: AUR vs. Transactional Load (PS, T1, CS=10%, NL=1, DD)

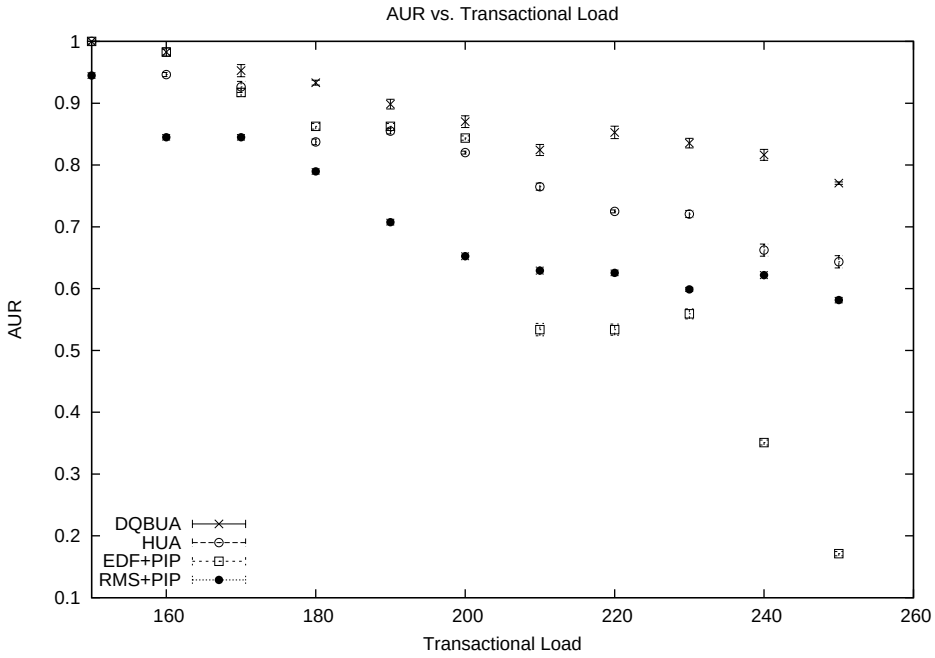

Figure 8.44: AUR vs. Transactional Load (OL, PS, T1, CS=10%, NL=1, DD)

The maximum difference, 15%, between the AUR performance of DQBUA and HUA occurs at the 240% transactional load point. The maximum difference between DQBUA and EDF+PIP occurs at the 250% transactional load point and is equal to 60%. The maximum difference between DQBUA and RMS+PIP is 24% at the 230% transactional load point.

In the next experiment, we investigate the effect of increasing number of distributed dependencies. Toward that end, we consider the case of a remote invocation made after 1, 5 and 10 nested levels of locks. We record the AUR at the 250% transactional load point for the thread set described in Table 7.2. The results are depicted in Figures 8.45 and 8.46. As can be seen, DQBUA suffers the most graceful degradation in the presence of increasing pressure to abort tasks as the number of distributed dependencies increases. This occurs because DQBUA can choose the "right" set of threads to keep based on system-wide notions of their importance.

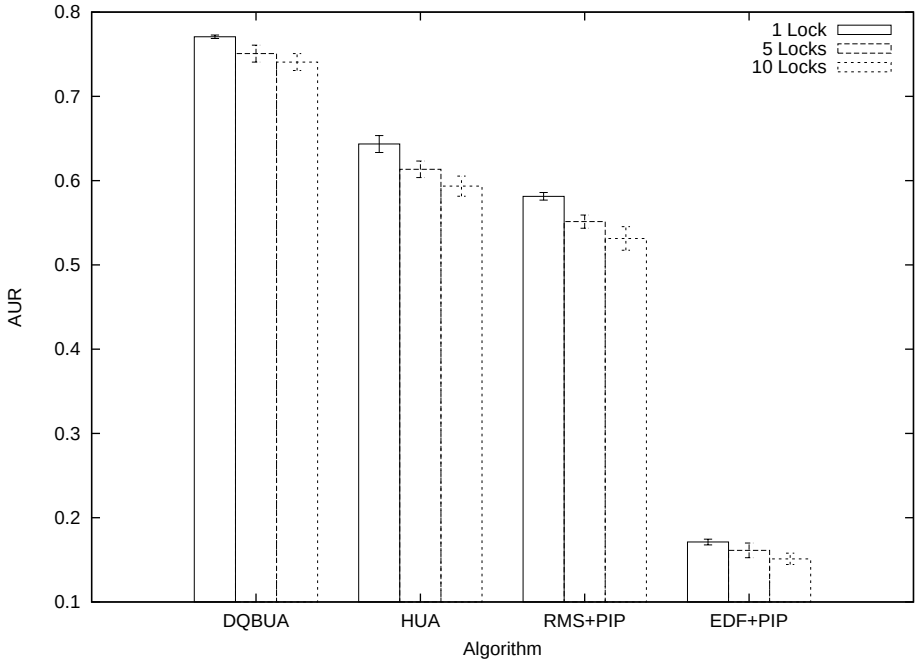

Figure 8.45: AUR vs. No. of Locks (PS, 250% transactional load point)

8.4 Failures

In this section, we study the effect of failures on the scheduling algorithm. Towards that end, we perform the same experiment as in Section 7.3 in Chapter 7, but using the proportional slack deadline decomposition method and considering distributed dependencies when 10 locks per critical section in 40% of execution time critical sections are considered. The experiment is conducted at the 250% transactional load point. The result is depicted in Figures 8.47

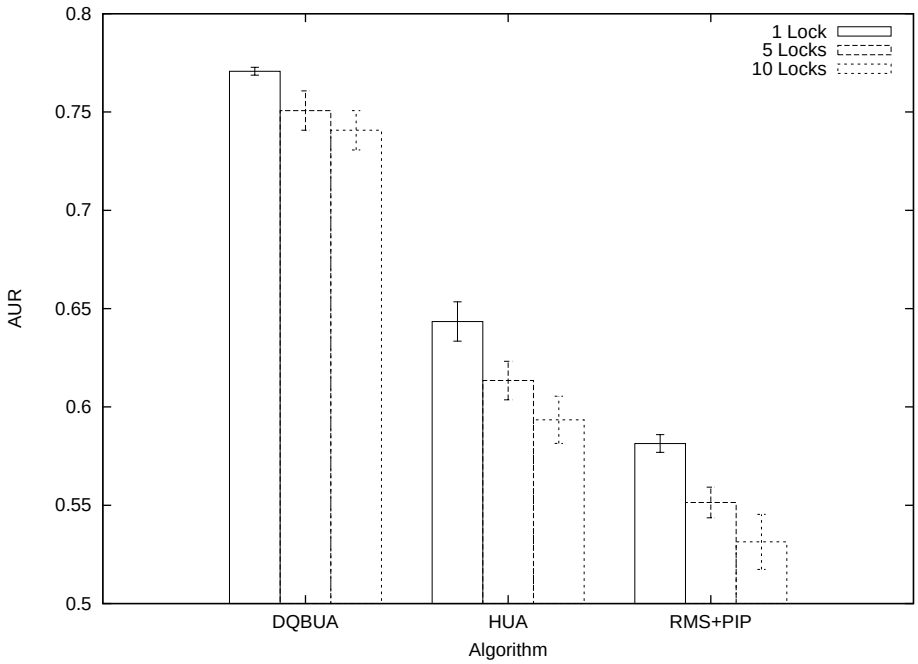

Figure 8.46: AUR vs. No. of Locks (PS, 250% transactional load point, no EDF)

and 8.48. The trends of this experiment is similar to those experienced by collaborative scheduling without dependencies in Chapter 7.

8.5 Overhead Measurements

8.5.1 Collaboration with Dependencies

In this section, we describe the overhead measurements we made on DQBUA. In order to measure the overhead of DQBUA, we instrument the kernel module with a call to **do_gettimeofday** right after the start of the part of the module that handles a collaborative scheduling event and right after the scheduling event is handled — note that the region being instrumented begins at the start of the quorum protocol and ends when a unicast message is sent to other nodes to notify them of changes in their schedule. **do_gettimeofday** allows us to reference the system time at sub-jiffies accuracy.

We perform the experiment for the thread set in Table 7.2. We record the average and standard deviations for 200 measurements of these overheads for each transactional load setting. Figure 8.49 shows the result of this experiment. For relatively low transactional loads, 180% to 210%, the overheads vary, relatively independent of transactional load, between 500μsecs and 600μsecs. Starting at the 220% transactional load point, the average overhead seems to

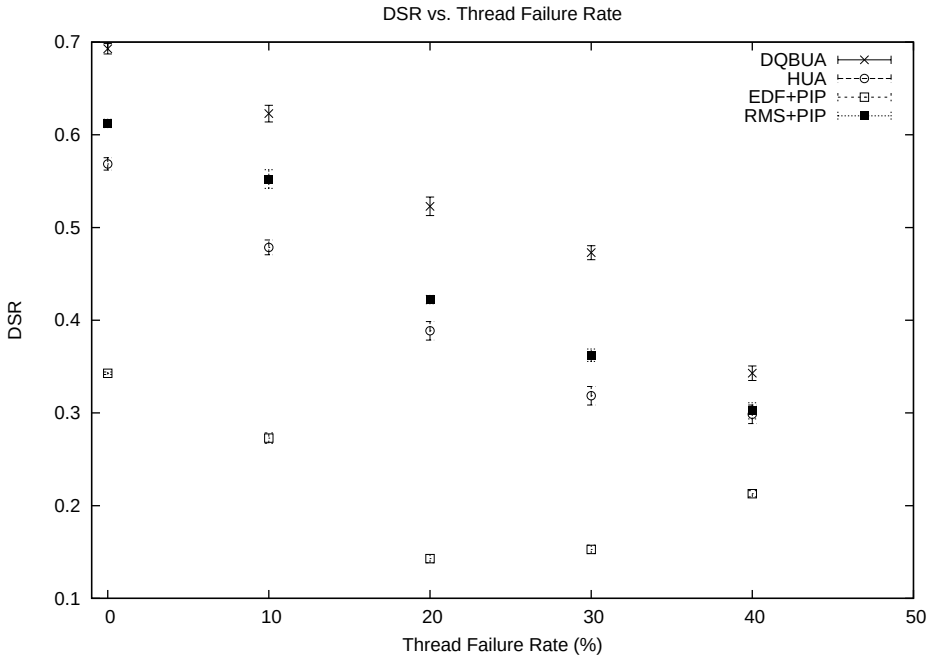

Figure 8.47: DSR vs. Failures (PS, 250% transactional load point)

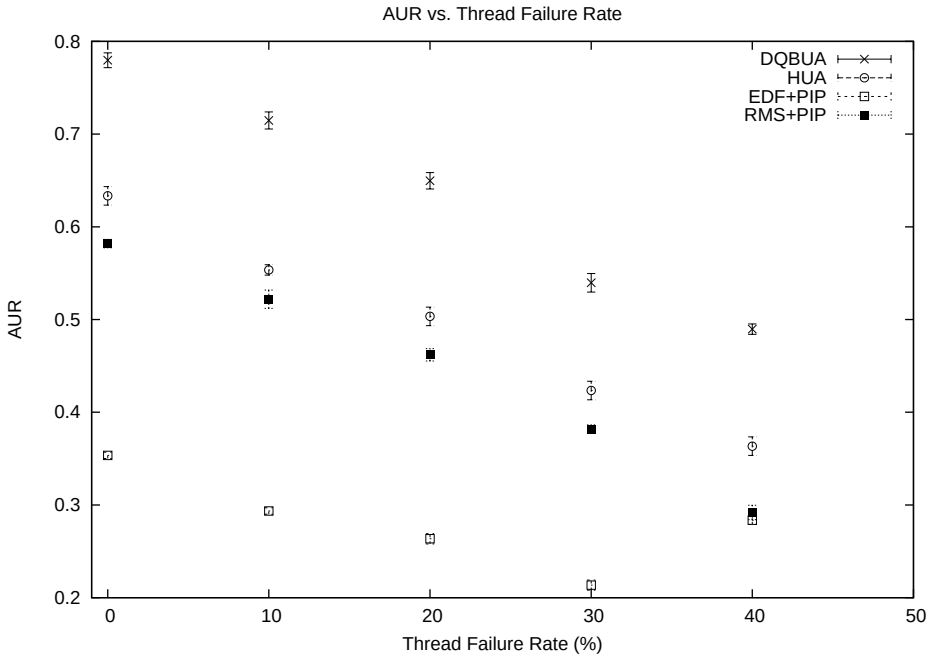

Figure 8.48: AUR vs. Failures (PS, 250% transactional load point)

take a linear relationship with transactional load.

In order to further depict the spread of overheads, we plotted a histogram of all the overhead measurements starting at 300μsecs and ending at 1050μsecs with bin sizes of 50μsecs. The result of this experiment is shown in Figure 8.50. The majority of the overheads are in the bins from 500μs to 800μs, with very few overhead measurements below or above that.

Generally speaking, the applications that can benefit from DQBUA need to tolerate its scheduling overheads. Therefore, their execution time magnitude scales should be large with respect to these scheduling overheads. As mentioned in Chapter 7, we consider tasks with periods generated from a uniform distribution with bounds of 10000000μs and 100000μs (i.e., 10s to 100ms) and therefore the tasks sets we generate can tolerate this overhead and benefit from collaboration. Note that the overhead measurements for collaborative scheduling algorithms are dominated by communication delay (including the delay suffered by the soft IRQ thread responsible for sending and receiving network messages – this is affected by system load) i.e., the scheduling overheads are affected to a greater extent by scheduling latencies than by computational overheads – although the later certainly play a part in overhead measurements.

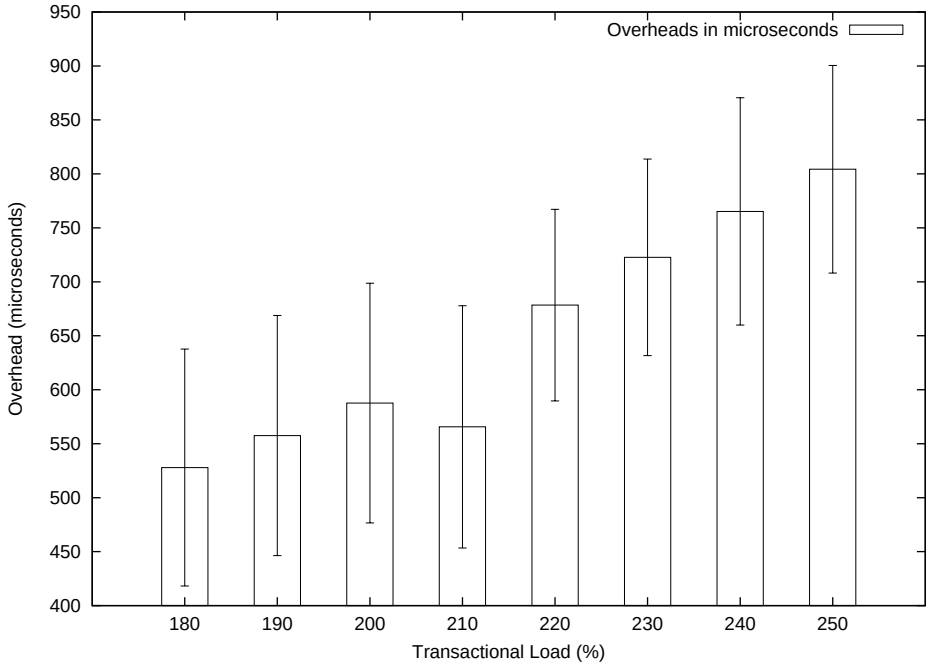

Figure 8.49: Overhead vs. Transactional load for DQBUA with T1

Similar overhead measurements where recorded for several different thread sets. We do not reproduce all of the resulting plots here to reduce clutter, but we do produce one more plot for another thread set. The overhead measurements for the thread set described in Table 8.1 are present in Figures 8.51 and 8.52.

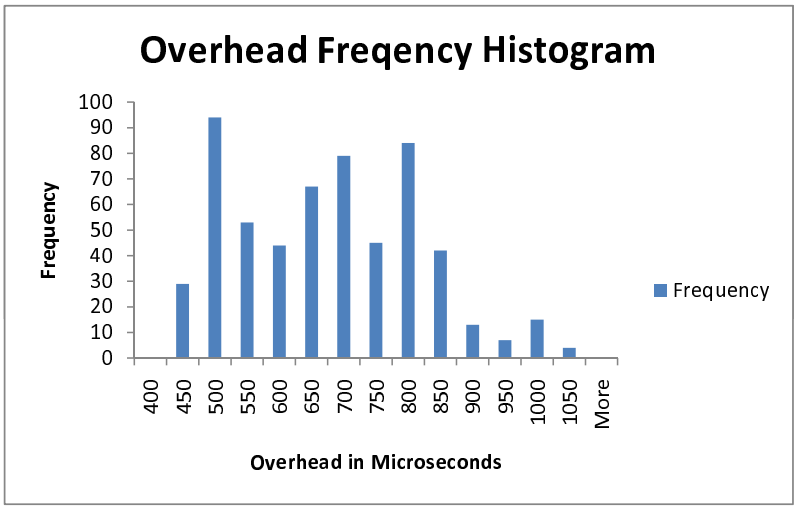

Figure 8.50: Overhead histogram for DQBUA with T1

Table 8.1: Task Set Parameters – Overhead Measurements

	Period (μsecs)	Utility
T1	1462000	76
T2	2090000	36
T3	1757000	30
T4	2199000	82
T5	2413000	85

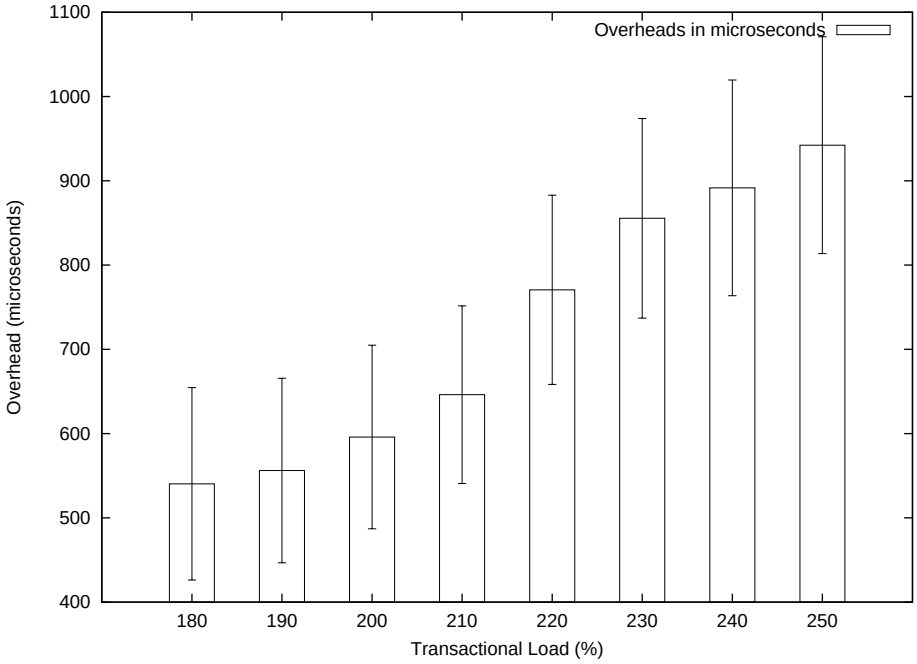

Figure 8.51: Overhead vs. Transactional load for DQBUA with T5

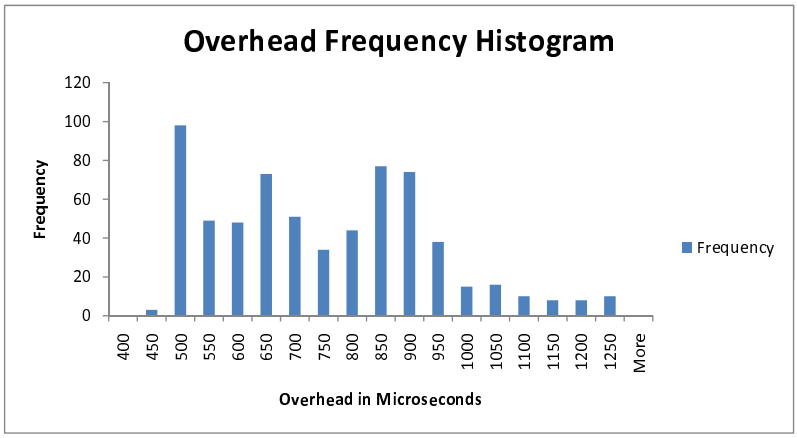

Figure 8.52: Overhead histogram for DQBUA with T5

As can be seen, the overhead behavior follows the same pattern for this thread set. As mentioned before, the overhead measurements we made for several different thread sets all followed this pattern. The spread of overhead values are clustered around a region starting at about 500μs and ending at about 900μs.

We conducted overhead measurements for twenty different thread sets. Each experiment consisted of 8 transactional load settings (from 190% to 250%), for each transactional load setting we ran 200 experiments. The average and standard deviation for all these overhead measurements are 690μs and 173μs to the nearest whole number. So, generally speaking, average task execution times must be larger than 690μs for DQBUA to be considered in the first place. We note that performance tends to be acceptable when the average execution times are at least 4 times that value, otherwise, the overhead of the algorithm is quite significant with respect to the execution times involved and a significant proportion of the application is spent performing scheduling operations instead of actual application code.

8.5.2 Collaboration without Dependencies

In this section, we describe the overhead measurements we perform for QBUA and ACUA. Toward that end, we instrument the kernel modules for both these algorithms with calls to **do_gettimeofday** before the start of a distributed scheduling event and after handling said distributed scheduling event. As in Section 8.5.1, we report the average and standard deviations of the overhead measurements we obtain. We perform the experiment on the thread set described in Table 7.2.

Before presenting the results for this experiment, it should be noted that QBUA offers performance benefits over ACUA in two ways 1) It uses multicast and unicast messages instead of broadcasts and 2) its communication overhead is constant in the presence of failure while ACUA's communication overhead scales linearly with the number of failed nodes. Since we consider a two-node testbed, the major source of benefit is 1) i.e., unicast and multicast messages instead of broadcast messages. Figure 8.53 depicts the result. As can be seen, the differences between ACUA and QBUA overheads are slight but noticeable. Also, note that the standard deviation for ACUA's overhead is larger than the standard deviations for QBUA's overhead. This occurs because the broadcast communication pattern used by ACUA introduces greater contention on the Ethernet communication medium used in the testbed.

Note that the slight difference in overhead head is matched by only slight improvements in performance over ACUA as described in Section 7.2 of Chapter 7. It is expected that this overhead difference will increase when more nodes are considered since the congestion caused by all those nodes broadcasting messages will increase message latencies and jitter in communication delay. Also, a larger testbed with a large number of node failures would also highlight the difference in performance as the overhead of the consensus protocol scales linearly with the number of failures while the overhead of the quorum algorithm remains

unchanged.

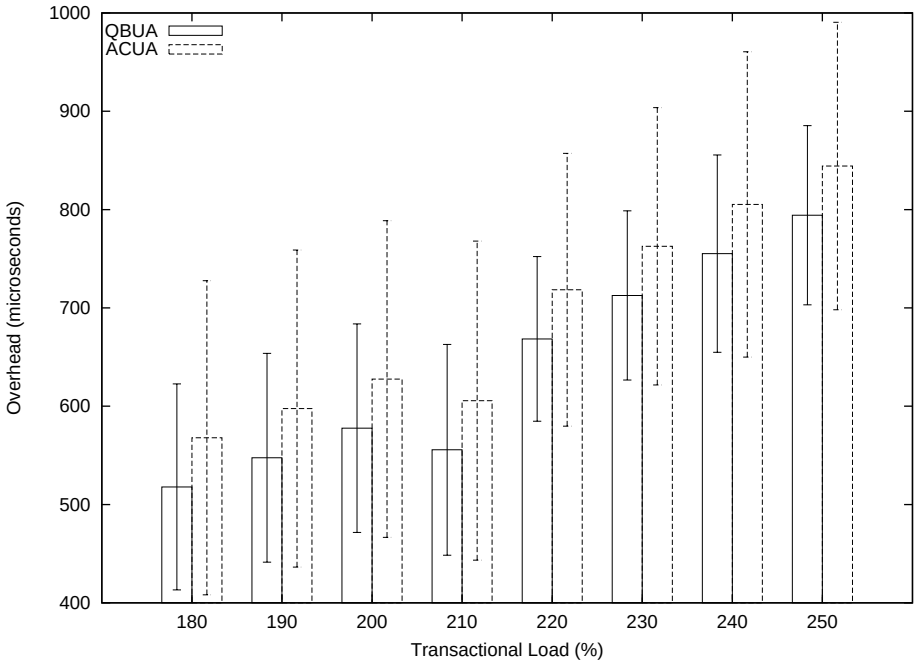

Figure 8.53: Overhead vs. Transactional load for ACUA and QBUA with T1

8.6 Conclusions

In this chapter, we performed a number of experiments to compare the performance of collaborative scheduling with independent scheduling in the presence of dependencies. The results indicate that for certain task sets, collaborative scheduling offers better timeliness performance since it can avoid decisions that are locally optimal but compromise system-wide optimality by taking into account information about all the nodes in the system when construction a schedule.

As in Chapter 7, collaborative scheduling with dependencies are best applied to thread sets with large execution times, high variability in section execution time, relatively stable communication delays and distributed dependencies. For such thread sets, collaborative scheduling outperforms independent scheduling. For example, for one such thread set, depicted in Figures 8.25, 8.26, 8.27 and 8.28, DQBUA outperforms, in terms of AUR, its independent scheduling competitors as follows; the average improvement over HUA during overloads is about 9% (with a standard deviation of about 4%) and a maximum improvement of about 15%. DQBUA outperforms EDF+PIP during overloads by an average of about 26% (with a standard deviation of about 26%) and a maximum of about 70% and outperforms RMS+PIP

by an average of about 22% (with a standard deviation of about 5%) and a maximum of about 27% – the results for other thread sets with similar properties follow the same trend.

The average overhead for the collaborative algorithms is about 690μs, with occasional excursions into the 1ms region. Therefore, the task sets that can take advantage of such scheduling algorithms must have average execution times at least as large as 1ms.

While the overhead of the quorum-based algorithm remains relatively constant as nodes are added and in the presence of failure, the consensus-based algorithms have overhead that scales linearly with the number of node failures in the system. Also, when the number of nodes increases, the broadcast behavior of the algorithm leads to higher congestion in the network and hence higher communication latencies and jitter.

Chapter 9

The Case for STM

9.1 Introduction

Recently, due to fundamental physical constraints such as heat emanations, the computer industry has undergone a paradigm shift: increasing computer performance is now done by increasing the number of cores on a chip rather than increasing clock speed [131]. Today, most machines produced are multi-core and the use of distributed systems is on the increase. Coinciding with this new direction of using concurrency to increase application throughput, is the discovery of a rich set of applications that are a natural fit for parallel and distributed architectures. From distributed databases to emerging distributed real-time systems [21], such emerging applications are only meaningful in a distributed system with multiple computing cores cooperating to execute the semantics of the application.

This parallelism offers a great opportunity for improving performance by increasing application concurrency. Unfortunately, this concurrency comes at a cost: programmers now need to design programs, using existing operating system and programming language features, to deal with shared access to serially reusable resources and program synchronization. The de facto standard for programming such systems is using threads, locks, and condition variables. Using these abstractions, programmers have been trying to write correct concurrent code ever since multitasking operating systems made such programs possible.

Unfortunately, the human brain does not seem to be well suited for reasoning about concurrency [85]. The history of the software industry contains numerous cases where the difficulty inherent in reasoning about concurrent code has resulted in costly software errors that are very difficult to reproduce and hence debug and fix. Among the more common errors encountered in lock-based software systems are deadlocks, livelocks, lock convoying, and, in systems where priority is important (e.g, embedded real-time systems), priority inversion. Such errors stem from the difficulty in reasoning about concurrent code.

Transactions have proven themselves to be a successful abstraction for handling concurrency in database systems. Due to this success, researchers have attempted to take advantage of their features for non-database systems. In particular, there has been significant recent efforts to apply the concepts of transactions to shared memory. Such an attempt originated as a purely hardware solution [66, 81] and was later extended to deal with systems where transactional support was migrated from the hardware domain to the software domain [124]. Software transactional memory (or STM) has, until recently, been an academic curiosity because of its high overhead. However, as the state-of-the-art improved and more efficient algorithms were devised, a number of commercial and non-commercial STM systems have been developed (see implementations section of [139]). In this chapter, we discuss the issues involved in implementing software transactional memory in distributed embedded real-time systems.

9.2 Motivation

Currently, the industry standard abstractions for programming distributed embedded systems include OMG/Real-Time CORBA's client/server paradigm and distributable threads [105] and OMG/DDS's publish/subscribe abstraction [110]. The client/server and distributable threads abstractions directly facilitate the programming of causally-dependent, multi-node application logic. In contrast, the publish/subscribe abstraction is a data distribution service for logically-single hop communications (i.e., from one publisher to one subscriber), and therefore, higher-level abstractions must be constructed – on an application-specific basis – to express causally-dependent, multi-node application logic (e.g., publication of topic A depends on subscription of topic B; B's publication, in turn, depends on subscription of topic C, and so on). All of these abstractions rely on lock-based mechanisms for concurrency control, and thus suffer from their previously mentioned inherent limitations.

In particular, lock-based concurrency control can easily result in local and distributed deadlocks, due to programming errors that occur as a result of the conceptual difficulty of the (lock-based) programming model. Detecting and resolving deadlocks, especially distributed deadlocks, that can potentially arise due to distributed dependencies is complex and expensive. Note that deadlocks can only be detected and resolved, as opposed to being avoided or prevented, in those distributed embedded systems where it is difficult to obtain a-priori knowledge of which activities need which resources and in what order. When a deadlock is detected in such systems, the usual method of resolving it is to break the cycle of the waiting processes by terminating one of them. Unfortunately, the choice of which process to terminate is not a simple one in real-time systems. By terminating one of the processes that are waiting in a cycle, we produce a chain of waiting processes. Depending on how, i.e., where, we break this cycle, it may or may not be feasible to meet the timing requirements of the remaining processes. Thus, we need to consider the structure of the dependency chain, after terminating a process to end the deadlock, in order to break the cycle in a way that

optimizes end-to-end timeliness objectives. Furthermore, a process's dependencies must be taken into account when making the choice about which process to terminate. For example, if a significant number of processes depend on the result of a process, terminating it to resolve a deadlock may not be in the best interest of the application. In addition, the cost of deadlock detection/resolution is exacerbated by the extra work necessary to restore the system to an acceptable state when failure occurs. Thus, deadlock resolution is a complex process.

The problem of distributed deadlock detection and resolution has been exhaustively studied, e.g., [36, 37, 42, 84, 102, 116, 125]. A number of these algorithms turned out to be incorrect by either detecting phantom deadlocks (false positives) or not detecting deadlocks when they do exist, e.g., [28, 37]. These errors occur because of the inherent difficulty of reasoning about distributed programs. This led to attempts at providing a formal method for analyzing such protocols to ensure correct behavior (e.g., [36]). Despite the difficulty of reasoning about distributed deadlock, solutions for this problem on synchronous distributed systems have been developed. Unfortunately, for asynchronous systems, errors in the deadlock detection process become inevitable. For real-time systems, these issues become more severe [125]. The semantic difficulty of thread and lock based concurrency control and the high overhead associated with detecting and resolving distributed deadlock, as indicated above, are the driving motivations for finding different programming abstractions for distributed embedded real-time systems. Chapter 2 contains a review of the literature regarding this matter and contains our reasoning for believing that STM is a promising solution to this problem.

9.3 STM for distributed embedded systems

There are a number of competing abstractions for implementing STM in distributed embedded real-time systems. An interesting abstraction is the notion of real-time distributed transactional objects, where code is immobile and objects migrate between nodes to provide a transactional memory abstraction. Another alternative is to allow remote invocations to occur within a transaction, spawning sub-transactions on each node (where they are executed using STM), and using a distributed commit protocol to ensure atomicity. A third alternative is to provide a hybrid model, where both data and code are mobile and the decision of which is moved is heuristically decided either dynamically or statically. Finally, there is the scenario of only considering local transactions and ensuring that all sections have finished executing their transactional portion before making a remote invocation. It is the last abstraction that we consider in this dissertation since it is a natural first step towards creating a distributed software transactional memory platform. Ongoing research in our group, based on the ideas gleaned from our initial investigation into the domain, is being conducted on the other approaches.

Several key issues need to be studied in order to use STM in distributed embedded systems, these are:

- Choosing an appropriate abstraction for including STMs in distributed embedded systems;
- Designing the necessary protocols and algorithms to support these abstractions;
- Implementing these abstractions in a programming language by making necessary changes to its syntax and in its run-time environment – or, alternatively, creating a library that implements STM semantics. The two approaches trade-off ease of implementation with efficiency; and
- Designing scheduling algorithms to provide end-to-end timeliness using these new programming abstractions.

In the rest of this chapter, we shall only discuss the first and last point, since it is there that we have actually contributed material in this dissertation. The other two points are discussed in greater detail in the Future Work section of Chapter 13.

9.3.1 Choosing an appropriate abstraction.

STM is a technology for multiprocessor systems, to use it in a multicomputer environment, we need to develop appropriate abstractions. We have identified four competing programming abstractions into which to incorporate STM:

- A model where cross-node transactions are not allowed and a task can only make a remote invocation from outside a transactional context.
- A model where cross-node transactions are permitted using remote invocations and atomicity is enforced using an atomic commit protocol;
- A model where a distributed cache coherence protocol is used to implement an abstraction of shared memory on top of which we can build STM; and
- A hybrid model where code or data is migrated depending on a number of heuristics such as size and locality.

In the first approach, no cross-node transactions are allowed. The research problems for this approach include deriving suitable schedulability tests for systems using this abstraction and improving the timeliness of STM by including real-time information in its operation. Towards that end, we have developed suitable schedulability analysis for this approach (Chapters 10 and 11) and have designed real-time contention managers to improve the timeliness of STM (Chapter 12). This is the *only* abstraction that we have considered in this dissertation. We have, however, identified some interesting research directions for realizing the other

abstractions for STM in a distributed system. We briefly describe the other approaches in this chapter and elaborate on the research points that need to be address for these abstractions in the Future Work section of Chapter 13.

In the second approach, we manage concurrency control on each node using STM, but allow remote invocations to occur within a transaction. Thus we allow a transaction to span multiple nodes. At the conclusion of the transaction, the last node on which transactional code is executed acts as a coordinator in a distributed commit protocol to ensure an atomic commitment decision. Our preliminary research, we suggest that this be further elaborated on in future work, indicates that such an approach may be prone to "retry thrashing" especially when the STM implemented on each node is lock-free.

Since lock-free STM is an optimistic concurrency control mechanism, extending the duration of a transaction by allowing it to sequentially extend across nodes results in a significantly higher probability of conflicts among transactions. Such conflicts lead to aborted transactions that are later retried. Retrying is antagonistic to real-time systems since it degrades one of the most important features of real-time systems: predictability. Lock-based STM tends to reduce some of this "thrashing" behavior since it eliminates part of the "optimism" of the approach. However, long transactions are still more susceptible to retries and introducing locks into the STM implementation necessitates a deadlock detection and resolution solution. Fortunately such a solution does not need to be distributed since it only needs to resolve local deadlocks.

Implementing STM on top of a distributed cache coherence protocol has been investigated in [67, 96]. In this approach, code is immobile, but data objects move among nodes as required. The approach uses a distributed cache coherence protocol to find and move objects. We propose that future work deals with the design of real-time cache coherence protocols, where timeliness is an integral part of the algorithm. STM can then be designed on top of these protocols and its performance compared to the flow control abstraction. An important advantage of this approach is that it eliminates the need for a distributed commit protocol. Since distributed commit protocols are a major source of inefficiency in real-time systems [60], such an approach is expected to yield better performance.

The last approach we identify is touched upon in [19]. This is a hybrid approach where either data objects or code can migrate while still retaining the semantics of STM. By allowing either code or data to migrate, we can choose a migration scenario that results in the least amount of communication overhead. For example, suppose we have a simple transactional program that increments the value of a shared variable X and stores the new value in the transactional store. Assume further that X is remote, using a data flow abstraction would necessitate two communication delays; one to fetch X from its remote location and the other to send it back once it has been incremented. Using a control flow abstraction in this case may be more efficient since it will only involve a single communication delay.

On the other hand, assume that several processes need access to a small data structure and that these processes are in roughly the same location and are far away from the data they

need. Since communication delay depends on distances, it may make sense to migrate the data to the processes in this case rather than incur several long communication delays by moving the code to the data. In short, the choice of whether to migrate code or data can have a significant effect on performance. In [19], this is accomplished under programmer control by allowing an *on* construct which a programmer can use to demarcate code that should be migrated. Interesting directions for research include coming up with solutions that would use static analysis at compile-time (or dynamically at run-time) to make decisions about which part of the application to move using a number of heuristics such as, for example, size of code/data and locality considerations.

As previously stated, we limit ourselves to the first abstraction in this dissertation. We have, however, identified a number of interesting research points that need to be addressed in order to incorporate the other abstractions for STM into a distributed system. We discuss elaborate on these research directions in Chapter 13.

9.3.2 Scheduling algorithms and analysis.

For STM to be useful in real-time distributed systems, designers need to determine the schedulability of their code before deploying it on production systems. Therefore, the design of suitable schedulability analysis is an important step in including STM in the repertoire of real-time programmers.

Unfortunately, this is a challenge due to the fact that the retry behavior of STM is antagonistic to predictability. There have been several attempts at providing timing assurances when STM is used in real-time systems or when lock-free data structures are used in real-time systems [6–8, 97]. These approaches only consider uni-processor systems and use the periodic task arrival model to bound retries.

Some of the approaches are fairly sophisticated and use, for example, linear programming [6] to derive schedulability criteria for lock-free code. The basic idea of these approaches is that, on a uni-processor system, the number of retries is bounded by the number of task preemptions that occur. This bound exists because a uni-processor can only execute one process at a time. Since it is not possible for a process to perform conflicting operations on shared memory, and hence cause the retry of another process, unless it is running, the number of preemptions naturally bounds the number of retries on uni-processors. Given this premise, the analysis performed in [6–8, 97] bounds the number of retries by bounding the number of times a process can be preempted under different scheduling algorithms. This analysis allows the authors to derive schedulability criteria for different scheduling algorithms based on information about process execution times, execution times of the retried code sections, process periods, etc.

More recently, attempts have been made at providing timeliness guarantees for lock-free data structures built on multiprocessor systems [72]. The approach used in [72] is suitable for

Pfair-scheduled systems and other multiprocessor systems where quantum-based scheduling is employed. The most restrictive assumption made in this approach is that access to a shared lock-free object takes at most two quanta of processor time. Using this assumption, the authors go on to bound the number of retries by determining the worst-case number of accesses that can occur to a shared object during the quanta in which it is being accessed. For an M processor system, the worst-case number of processes that can interfere with access to a particular shared object is $M-1$. Given an upper bound on the number of times a process can access a shared object within a quanta, it is possible to derive an upper bound on the number of retries in such a system. The authors also go on to describe how it is possible to use the concept of a "supertask", basically a single unit that is composed of several tasks that are to be scheduled as one unit, to reduce the worst-case number of retries and hence improve system performance.

We have developed, in Chapters 10 and 11, response time computation techniques for distributed systems programmed using the first approach to incorporating STM into distributed systems (i.e., there are no cross-transactional node). We propose that future work look into the possibility of coming up with suitable response time analysis techniques for the other abstractions considered.

9.4 Conclusions

Programming distributed systems using lock-based concurrency control is semantically difficult and computationally expensive. In order to alleviate some of these problems, we propose the use of STM for concurrency control. In order to achieve this goal, a number of issues need to be addressed. This chapter outlines these some issues, provides a roadmap for the problems that we have already solves and proposes a number of directions for future work for the issues that we do not address in this dissertation.

Chapter 10

Schedulability of Distributed Uniprocessor Systems with STM

10.1 Introduction

As mentioned in Chapter 9, STM is a promising alternative to traditional lock-based concurrency control. However, while STM has many promising features, it is not a silver bullet. Some problems associated with STM include handling irrevocable instructions such as I/O, the weak atomic semantics of some of the current implementations [46] and the overhead of retries. Despite these disadvantages, its semantic simplicity makes it a very promising alternative to lock-based concurrency control.

In this chapter, we consider using STM as the concurrency control mechanism for (non-I/O code in) distributed real-time systems. Toward this, we propose a method for computing an upper bound on the worst-case response time of periodic tasks, programmed using STM, running in a distributed real-time system that employs Earliest Deadline First (EDF) scheduling.

10.2 General Framework

There are different ways of incorporating STM into distributed systems [46]. In this chapter, we consider a model where a distributed application consists of several pieces of code, which we will refer to as tasks, executing on single nodes that are subject to crash failures. Concurrency control on each node is managed using STM. A task makes an invocation (a procedure call or an RPC, depending on whether the successor resides on the same node) after it has finished execution (at which point all its STM transactions should have been committed). We do not allow cross-node critical sections (i.e. cross node transactions).

We present a method for analyzing the worst-case response times in such a system. In order to do this, we first show how to extend Spuri's response time analysis technique [126] to include the overhead associated with the retry behavior of STM on a single machine without considering offsets in Section 10.3.2. We then extend this analysis to include offsets in Section 10.4.2. Including offsets is necessary since it enables us to handle the precedence constraints of a distributed system where tasks make RPC calls to remote nodes and therefore some tasks cannot start before their predecessor makes an invocation.

The ability to use offsets and jitters to represent these precedence constraints is discussed in Section 10.6. In Section 10.5.1, we show how failures can be considered in the analysis by ensuring that exception handlers can be executed if necessary. Our goal is to prove that it is possible to provide real-time assurances for distributed systems where concurrency control is managed using STM. Introducing STM to the repertoire of programming tools available for the real-time programmer can considerably improve the quality of distributed concurrent real-time programs and reduce the software development time by reducing the complexity of the programming environment. As such, this chapter is our first step towards achieving the goal of studying STM for distributed real-time systems in all its possible varieties as outlined in [46].

10.3 Tasks with Jitter

Given a set of periodic, independent tasks scheduled by EDF on a single processor, Spuri, in [126], proposed an algorithm for computing an upper bound on the worst-case response time for a task. In this section we extend the algorithm to consider tasks with mutually exclusive resource access requirements that are programmed using software transactional memory. We extend Spuri's analysis to consider these tasks and compare the tightness of the bound we obtain by comparing it to the utilization based schedulability analysis of lock-free code proposed in [7].

Spuri's idea for computing an upper bound on worst-case response time is based on finding a "critical instant", the release time of a task, τ_a, that would cause it to experience maximal interference from other tasks. This critical instant is found in a busy period, which is defined as a period of time during which a processor is busy executing tasks in the system. The following theorem is helpful in finding that critical instant:

Theorem 54 (Spuri [126])**.** *The worst-case response time of a task τ_a is found in a busy period in which all other tasks are released simultaneously at the beginning of the busy period, after having experienced their maximum jitter.*

The proof of this theorem, presented in [126], rests on the fact that the conditions in Theorem 54 result in the largest number of interferences from other tasks. The same condition holds for tasks using STM, since the largest number of interferences will result in the largest

number of transaction retries and hence the worst-case response time. Note that we make the assumption that each interference by a task results in a transactional retry. This is a pessimistic assumption since the task may not be executing its transactional code at this point in time, or the transactional operations may not conflict before the transaction commits. However, we make this assumption to give our analysis the nature of an upper bound. In Section 10.3.2, we present the modified analysis for independent tasks on a single processor.

10.3.1 Task model

We consider a task model with periodic tasks scheduled with the EDF discipline. The system is composed of a set of N periodic tasks executing in a single processor. Each task τ_i is activated periodically with a period of T_i, has a computation time of $C_i + m_i s$, where C_i is the computation time of the task instance without considering the transactional part of the code, m_i is the number of times transactional code is executed in the task activation and s is the computation cost of the transactional part of the code. In the rest of the chapter, we shall refer to a task activation as *job*. Each job has a relative deadline d_i and a release jitter bounded by J_i. We refer to the absolute deadline of a job as D_i.

10.3.2 Analysis

In this section, we compute the worst-case contribution of a task τ_i to the response time of the task under analysis τ_a. Specifically, we compute the worst-case contribution of task τ_i during a busy period of duration t when the deadline of τ_a is D. Without loss of generality, we label the time at which the busy period starts as t_0, and measure the duration of the busy period, t, and the deadline of τ_a, D, from t_0.

As per Theorem 54, the worst-case contribution of a task, τ_i, occurs when it is released at the start of the busy period after having experienced its maximum jitter. This scenario is chosen so as to maximize the number of instances of τ_i that occur during the busy period in order to maximize interference.

Only jobs with a deadline less than or equal to D can interfere with τ_a, also, jobs that start outside the busy period, t, do not contribute to the interferences that occur within that period. Using this information, we can compute the maximum number of jobs of τ_i that can interfere with τ_a.

From [108], we know that the number of jobs of τ_i within the busy period, p_t, is:

$$p_t = \left\lceil \frac{t + J_i}{T_i} \right\rceil \tag{10.1}$$

and the number of task instances with deadlines at or before D is:

$$p_D = \left\lfloor \frac{J_i + D - d_i}{T_i} \right\rfloor + 1 \tag{10.2}$$

Since both conditions, p_D and p_t, must be satisfied, the maximum number of interferences of the jobs of τ_i in τ_a is:

$$n_i = min\left(\left\lceil \frac{t + J_i}{T_i} \right\rceil, \left\lfloor \frac{J_i + D - d_i}{T_i} \right\rfloor + 1 \right)_0 \tag{10.3}$$

The zero that appears as a subscript in the equation above causes the result of the bracketed expression to be zero if its value is negative. This is necessary because if $d_i > D$, no activation of τ_i will interfere with τ_a. Thus, the worst-case contribution of τ_i to the response time of τ_a is:

$$W_i(t, D) = n_i(C_i + m_i s) \tag{10.4}$$

Given this equation, it is possible to compute the response time of τ_a after having determined the critical instant. Unfortunately, we do not know which instant in the busy period is the critical instant. However, it is known that the critical instant can be found either at the beginning of the busy period, or at an instant of time such that the deadline of the analyzed job of τ_a coincides with the deadline of a task τ_i's job. Otherwise, it would be possible to make the activation time of τ_a earlier, without changing the schedule, to increase the response time. The set of instants, Ψ, at which the deadline of τ_a's job coincides with the deadline of the job of some other task in the busy period, is:

$$\Psi = \bigcup \{(p-1)T_i - J_i + d_i\} \tag{10.5}$$
$$\forall p = 1 \cdots \left\lceil \frac{L + J_i}{T_i} \right\rceil, \ \forall i$$

In the above equation, L is the worst-case length of a busy period. The following recurrence relation can be used in computing L:

$$L = \sum_{\forall i} \left\lceil \frac{L + J_i}{T_i} \right\rceil C_i^{exe} \tag{10.6}$$

where C_i^{exe} is an estimate of the processing load placed on the processor by a job during the busy period.

Equation (10.6), one of the many recurrence equations that can be found in response time analysis [80], can be solved by starting with a small initial value for L and then iterating

until the equation converges. The equation is guaranteed to converge if the system is not over-utilized (i.e., the load is under 100%). The execution time of a job, without considering any interference, is $C_i + m_i s$. To this execution time, we need to add the load that a job can place on the processor during the busy period if it interferes with another job. For uni-processor systems scheduled using EDF an interference can only occur when a new job arrives with a lower deadline than those already executing causing a context switch. Thus, a job can cause, at most, one retry in some other job. Therefore, the load a job places on the processor during the busy period is $C_i + m_i s + s$, giving us:

$$L = \sum_{\forall i} \left\lceil \frac{L + J_i}{T_i} \right\rceil (C_i + m_i s + s) \tag{10.7}$$

Thus, we can compute the set of critical instants by subtracting d_a from each element in Ψ. We then consider all the critical instants in our analysis to find the critical instant that gives us the worst-case response time.

There may be several jobs of τ_a in the busy period, therefore we need to examine all of these jobs in order to determine which one of them results in the worst-case response time. Assuming that the first instance of τ_a occurs A time units after the start of the busy period, the completion time of job p of τ_a, $w_a^A(p)$, can be computed as:

$$w_a^A(p) = p(C_a + m_a s) + \sum_{\forall i \neq a} W_i(w_a^A(p), D^A(p)) + Is \tag{10.8}$$

where I is the maximum number of interferences that can occur in jobs of higher priority (lower deadline) than the instant of τ_a being studied as expressed in Equation (10.9).

$$I = \sum_{\forall i} min \left(\left\lceil \frac{t + J_i}{T_i} \right\rceil, \left\lfloor \frac{J_i + D - d_i}{T_i} \right\rfloor + 1 \right)_0 \tag{10.9}$$

In Equation (10.8), the term $D^A(p)$ is the deadline of job p when the first job of τ_a occurs A time units after the start of the busy period and can be computed as:

$$D^A(p) = A - J_a + (p-1)T_a + d_a \tag{10.10}$$

The response time is obtained by subtracting the activation time from the completion time of each task:

$$R_a^A(p) = w_a^A(p) - A + J_a - (p-1)T_a \tag{10.11}$$

Next we need to determine the set of values we will use for A. Note that we have already established that the critical instant can only be in the set Ψ computed in Equation (10.5).

We now further narrow down the set of values that need to be considered. Naturally, for each value of p we only need to check values of A within one period. Thus, we can further narrow down the critical instants we should consider to:

$$\Psi^* = \{\Psi_x \in \Psi \mid (p-1)T_a - J_a + d_a \leq \Psi_x < pT_a - J_a + d_a\} \tag{10.12}$$

For each of the values, Ψ_x, in Ψ^*, we need to check the values $A(\Psi_x) = \Psi_x - [(p-1)T_a - J_a + d_a]$.

Finally, we determine the worst-case response time by examining all instants of τ_a within the busy period and taking the maximum response time as our result:

$$\begin{gathered} R_a = \max R_a^A(p) \\ \forall p = 1 \cdots \left\lceil \frac{L - J_a}{T_a} \right\rceil, \forall A(\Psi_x) \mid \Psi_x \in \Psi^* \end{gathered} \tag{10.13}$$

In Sections 10.4.2 and 10.6, this analysis is extended to allow us to handled distributed systems (while Section 10.5.1 shows how we can include exception handlers in the analysis).

10.4 Tasks with Jitter and Offsets

In this section, we study a system where groups of tasks are grouped together into logical entities which we shall call "transactions". Each transaction is activated by a periodic external event. Once this external event has arrived, the tasks in each transaction begin execution after a certain time period, which we refer to as the offset, has passed. This model can be used to model distributed systems. Already, several papers have been published showing how to obtain an upper bound on the response time of distributed systems programmed using this task model, e.g., [62, 108, 112]. Those attempts are based on the Holistic analysis first proposed by Tindell and Clark [136]. In this section, we extend such analysis to deal with STM based concurrency control.

10.4.1 Task model

We consider a system composed of a set of tasks executing on the same processor. These tasks are grouped into logical entities referred to as transactions. Each transaction, Γ_i, is activated by a periodic external event with period T_i. A transaction consists of n_i tasks (not to be confused with the n_i used in the analysis of Section 10.3.2). We designate the tasks τ_{ij}, with the first subscript, i, identifying the transaction the task belongs to, and the second

subscript, j, specifying the order of the task within the transaction in non-decreasing order of offsets.

Each of these tasks has an execution time of $C_{ij} + m_{ij}s$, where C_{ij} and s are the execution times of the transactional and non-transactional part of a task, respectively, and m_{ij} is the number of times transactional code is invoked in a task. Tasks are activated after a certain time, which we shall refer to as the offset, elapses from the arrival of the external event that triggered the transaction. For each τ_{ij}, its offset is designated ϕ_{ij}. We also allow a task to suffer release jitter which is bounded by the term J_{ij}. Both offsets and jitter can be larger than the period of their transaction.

10.4.2 Analysis

In this section, we compute the worst-case response time for task τ_{ab}. In order to do this, we must determine the worst-case contribution of each transaction to the response time of the task under analysis. The theorem below can be used for this purpose:

Theorem 55 (Palencia and Harbour [108])**.** *The worst-case contribution of transaction Γ_i to the response time of a task τ_{ab} is obtained when the first activation of some task τ_{ik} that occurs within the busy period coincides with the beginning of the busy period, after having experienced the maximum possible delay, i.e., the maximum jitter, J_{ik}.*

Again, this theorem is based on the fact that the worst-case contribution will occur when the most number of tasks are released within the busy period. For the sake of being concise, we will not reproduce the whole derivation of the analysis, which can be found in [108], but will only include the final results and the modifications necessary for accommodating STM.

The worst-case contribution of a task, τ_{ij}, to the response time of the task under analysis, τ_{ab}, during a busy period of duration t and deadline D, when the task whose activation time coincides with the start of the busy period is τ_{ik}, is:

$$W_{ijk}(t,D) = \left(\left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + \min\left(\left\lceil \frac{t - \varphi_{ijk}}{T_i} \right\rceil, \left\lfloor \frac{D - \varphi_{ijk} - d_{ij}}{T_i} + 1 \right\rfloor \right) \right)_0 (C_{ij} + S) \tag{10.14}$$

where $\varphi_{ijk} = T_i - (\phi_{ik} + J_{ik} - \phi_{ij}) \mod T_i$. Thus, the contribution of transaction Γ_i is the summation of the contribution of all its tasks:

$$W_{ik}(t,D) = \sum W_{ijk}, \quad \forall j \in \Gamma_i \tag{10.15}$$

In order to make the analysis tractable, the worst-case contribution of a transaction, Γ_i, is considered to be the maximum of all possible contributions that could have been caused by considering each of the tasks of Γ_i as the start of the busy period:

$$W_i^*(t,D) = \max\left(W_{ik}(t,D)\right), \quad \forall k \in \Gamma_i \tag{10.16}$$

We number the activations of a job within the busy period using the index p, and consider the first activation to start within the busy period to have an index of $p = 1$. The activations that start before the busy period and suffer their maximum jitter to start at the beginning of the busy period have indices $p \leq 0$. Thus, we can determine the index of the first, $P_{0,ijk}$, and last, $P_{L,ijk}$, activations to contribute to the busy period as follows:

$$P_{0,ijk} = -\left\lfloor \frac{J_{ij}+\varphi_{ijk}}{T_i} \right\rfloor + 1 \qquad (10.17)$$

$$,P_{L,ijk} = \left\lceil \frac{L-\varphi_{ijk}}{T_i} \right\rceil \qquad (10.18)$$

where L is the maximum length of the busy period as computed in Equation (10.7).

Thus, like in Section 10.3.2, the set of values to analyze is:

$$\Psi = \bigcup \{\varphi_{ijk} + (p-1)T_i + d_{ij}\} \tag{10.19}$$
$$\forall p = P_{0,ijk} \cdots P_{L,ijk}, \quad \forall j,k \in \Gamma_i$$

Thus, if the first activation of τ_{ab} occurs after A time units from the start of the busy period, the worst-case completion time of activation p of task τ_{ab} can be computed as:

$$W_{abc}^A(p) = (p - P_{0,ijk} + 1)(C_{ab} + s) \tag{10.20}$$
$$+W_{ac}^-(W_{abc}^A(p), D_{abc}^A(p)) + \textstyle\sum_{\forall i \neq a} W_i(W_{abc}^A(p), D_{abc}^A(p))$$

where W_{ac}^- is the result of Equation (10.15) without considering the contribution of τ_{ab} and $D_{abc}^A(p)$ is the deadline of activation p when the first one occurs at time A:

$$D_{abc}^A(p) = A + \varphi_{abc} + (p-1)T_a + d_{ab} \tag{10.21}$$

Thus, we can obtain the response time of a task by subtracting from the completion time the arrival time of the external event:

$$R_{abc}^A(p) = W_{abc}^A(p) - A - \varphi_{abc} - (p-1)T_a + \phi_{ab} \tag{10.22}$$

Also, as in Section 10.3.2, we only need to check the value of A within one period, thus, the points we need to check are:

$$\Psi^* = \{\Psi_x \in \Psi \mid \varphi_{abc} + (p-1)T_a + d_{ab} \leq \Psi_x < \varphi_{abc} + pT_a + d_{ab}\} \tag{10.23}$$

For each value of Ψ_x above, we check $A = \Psi_x - [\varphi_{ijk} + (p-1)T_a + d_{ab}]$. Naturally, the worst-case response time is the maximum response time obtained from the analysis, i.e.,

$$\begin{gathered} R_{ab} = \max(R^A_{abc}(p)) \\ \forall p = P_{0,abc} \cdots P_{L,abc} \quad \forall c \in \Gamma_A, \quad \forall A \in \Psi^* \end{gathered} \tag{10.24}$$

10.5 Handling Failures

In this section, we extend our analysis to take failures into account. In some distributed systems, failures are the norm rather than the exception. Therefore, it is necessary to provide some form of assurance on system performance in their presence. We assume that each task, τ_{ij}, has an exception handler that can be used to restore the system to a safe state in case of failure, and that this exception handler has an execution time C^h_{ij} and relative deadline d^h_{ij}. The absolute deadline of the handler is relative to the time that failure is detected, t_f, i.e., $D^h_{ij} = t_f + d^h_{ij}$.

When a node fails, all the jobs executing on that node cease to exist. Since we are considering "transactions" where a sequence of consecutive jobs execute within one logical computational context, it is necessary to understand the effect of failures on this abstraction. Naturally, a failure may fragment a transaction leading to several orphan jobs (i.e., jobs that have been disconnected from their downstream predecessor due to node failure). These jobs need to be identified and their exception handlers need to be executed in order to restore the system to a safe state.

Therefore, in order to have a fault-tolerant system, it must be possible to execute the exception handlers before their deadlines when failure occurs. In this section, we show how we can take the execution time of the exception handlers into account when computing response times in order to ensure safe execution of the system in the presence of failures.

10.5.1 Analysis

We need to determine the maximum number of exception handlers that can execute within a busy period in order to take their overhead into account. As in Sections 10.3.2 and 10.4.2, there are two conditions that determine the number of jobs, in this case exception handlers, that can contribute to the worst-case response time of an instance of a task, τ_{ab}, within busy period of duration t; 1) The number of exception handlers that execute within the duration

of the busy period (including exception handlers that were released before the busy period but whose activation time is delayed until the start of the busy period), and 2) the number of handlers with deadline less than or equal to the deadline of the job being analyzed.

From [108], we know that the number of interferences from the jobs of task τ_{ij} that occur from jobs that start at the beginning of a busy period, after suffering some jitter, is:

$$x_i = \left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor \tag{10.25}$$

Each of these activations has an associated exception handler. In the worst-case, each job executes to completion, thus, using up as much processor time as possible, and then an error occurs that causes the trigger of its exception handler, thus, using up more processor time to execute the handler. It is this worst-case scenario from which we derive the deadline of the exception handlers. Note that since we are considering activations of the same task, and all of these activations start at the beginning of the busy period, their exception handlers have the same deadline which is:

$$D_{ij}^{h_{first}} = d_{ij} + d_{ij}^{h} \tag{10.26}$$

If we consider, without loss of generality, that the beginning of the busy period t_B is time zero, the contribution of these exception handlers is $x_i C_{ij}^{h}$ if $D_{ij}^{h_{first}} \leq D$ and zero otherwise.

We now turn our attention to determining the number of interferences that occur from job instances started within the busy period. Below, is the equation that determines the number of instances that can occur within a busy period of duration t:

$$n_{inst} = \left\lceil \frac{t - \varphi_{ijk}}{T_i} \right\rceil \tag{10.27}$$

For each of these activations, the deadline of their handler can be computed as:

$$De = \varphi_{ijk} + (p-1)T_i + d_{ij} + d_{ij}^{h} \qquad \forall p = 1 \cdots n_{inst} \tag{10.28}$$

and they each contribute a factor of C_{ij}^{h} to the worst-case response time if their deadline is less than or equal to D.

At this point we have computed the contribution of the execution time of handlers of the activations of τ_{ij} that start at or after the beginning of the busy period t_B. Figure 10.1 depicts the types of scenarios we will be considering. The term in Equation (10.25) represents the

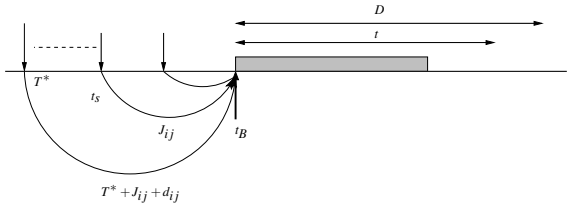

Figure 10.1: Uniprocessor: Scenario for calculating worst-case contribution

number of jobs that can be delayed at most J_{ij} so that their activation starts at t_B. In Figure 10.1, the first such job starts at t_s; other jobs that follow it will be delayed an amount of time less than J_{ij} in order to start at the beginning of the busy period. Equation (10.27) computes the number of jobs that will start after the beginning of the busy period, i.e., after t_B. However, if failures are not considered, jobs that start before t_s, such as the one depicted as starting at time T^* in Figure 10.1, do not contribute to the busy period. This lack of contribution occurs because even if these jobs were delayed J_{ij}, their activation time would fall before t_s. However, now that we consider failures, it is possible for these tasks to contribute to the worst-case response time if their exception handlers start within the busy period. Using similar reasoning as in Section 10.4.2, we compute the worst-case contribution of these exception handlers by considering how many can start at the beginning of the busy period. The latest start time of a handler whose job starts at T^* is:

$$S = T^* + d_{ij} \tag{10.29}$$

If this start time, S, is greater than or equal to t_B then the handler will contribute to the busy period. In other words:

$$T^* + d_{ij} \geq T^* + nT_i + J_{ij} \tag{10.30}$$

which gives us:

$$n < \frac{d_{ij} - J_{ij}}{T_i} \tag{10.31}$$

Since n is an integer, Equation (10.31) resolves to:

$$n = \left(\left\lceil \frac{d_{ij} - J_{ij}}{T_i} \right\rceil - 1 \right)_0 \tag{10.32}$$

As before, the zero subscript indicates that negative values are considered zero (in this case, such an event indicates that none of the jobs starting before t_s can contribute to the busy period). Thus, the contribution of these jobs is nC_{ij}^h if $d_{ij}^h \leq D$ and zero otherwise. We can

now compute the contribution of the exception handlers of τ_{ij} to the response time of a job of τ_{ab} in a busy period of duration t and deadline D, when the task whose activation time coincides with the start of the busy period is τ_{ik}, using the following function:

Thus, we can modify Equation (10.14) from Section 10.4.2 to:

$$W_{ijk}(t,D) = \left(\left\lfloor \frac{J_{ij}+\varphi_{ijk}}{T_i} \right\rfloor + \min\left(\left\lceil \frac{t-\varphi_{ijk}}{T_i} \right\rceil, \left\lfloor \frac{D-\varphi_{ijk}-d_{ij}}{T_i}+1 \right\rfloor\right)\right)_0 (C_{ij}+S) + W^h_{ijk}(t,D) \tag{10.33}$$

Also, we need to modify the critical instants to be examined in order to accommodate the inclusion of the exception handlers in the busy period, thus, Equation (10.19) becomes:

$$\Psi = \bigcup\{\varphi_{ijk}+(p-1)T_i+d_{ij}\}\bigcup\{\varphi_{ijk}+(p-1)T_i+d_{ij}+d^h_{ij}\} \quad \forall p = P_{0,ijk}\cdots P_{L,ijk}, \quad \forall j,k \in \Gamma_i \tag{10.34}$$

Algorithm 15: $W^h_{ijk}(t,D)$

```
1:  sum=0;
2:  if d^h_ij ≤ D then
3:      n = (⌈(d_ij − J_ij)/T_i⌉ − 1)_0;  sum ← sum + nC^h_ij;
4:  if d_ij + d^h_ij ≤ D then
5:      x_i ← ⌊(J_ij + φ_ijk)/T_i⌋;
6:      sum ← sum + x_i C^h_ij;  n_inst = ⌈(t − φ_ijk)/T_i⌉;
7:      for 1 ≤ p ≤ n_inst do
8:          De = φ_ijk + (p − 1)T_i + d_ij + d^h_ij;
9:          if De ≤ D then sum ← sum + C^h_ij;
10: return sum;
```

Similarly, Equation (10.7) needs to be modified to:

$$L = \sum_{\forall i,j} \left\lceil \frac{L+J_{ij}}{T_i} \right\rceil (C_i + m_i s + s + C^h_{ij}) \tag{10.35}$$

Essentially extending the execution time of a job by C^h_{ij} because, in the worst-case, the exception handler is triggered at the last instant of time in the execution of the job, thus, placing a demand on the processor equal to the total time of the job and the handler. The rest of the analysis remains unchanged.

10.6 Dynamic Jitter and Offsets

In this section we briefly indicate how the iterative techniques first developed by Palencia and Harbour in [109], based on Tindell and Clark's Holistic analysis [136], and later improved in [62, 108, 112] can be used to provide response time analysis of distributed systems programmed using STM. From the analysis in Sections 10.4.2 and 10.5.1, we can perform response time analysis of systems where concurrency control is managed using STM, tasks have offsets and jitters, and failures are possible. Initially, the offset of each task is set to the minimum possible completion time of its predecessor, i.e.,

$$\phi_{ij} = \sum_{i \leq k \leq j} (\delta_{ik} + C_{ij}) + \delta_{ij} \quad \forall 1 \leq j \leq N_i \tag{10.36}$$

where δ_{ij} is the communication delay between nodes i and j. The jitters are all set to zero and the response time, R_{ij}, is computed using either the analysis in Section 10.4.2, if failures are not being considered, or Section 10.5.1, if we wish to consider failures. Then, jitters are modified as follows:

$$J_{i1} = 0 \tag{10.37}$$

$$J_{ij} = R_{ij-1} + \delta_{ij} - \phi_{ij} \quad \forall 1 < j \leq N_i \tag{10.38}$$

Offsets remain unchanged. Essentially, this means that the jitters are modified so that each task, τ_{ij}, is released at most δ_{ij}, the communication delay, time units after the completion of its predecessor τ_{ij-1}. We then compute the response times again using either the analysis in Section 10.4.2 or 10.5.1. This process is repeated until the result of two successive iterations are the same, at which point we have obtained the response time for each task. If the response times do not diverge, the process above is guaranteed to converge to the solution since the process is monotonic in its parameters. Naturally, during the computation only the contribution of the tasks running on the same processor is taken into account when computing the response times.

10.7 Experiments

In this section, we experimentally evaluate the performance of the proposed algorithm against a system simulated using RTNS [107]. In the first set of experiments, we measure the average ratio, R_{ana}/R_{sim}, between the response time of the analysis to the response time of the simulation. Execution times and periods are randomly generated as is the number of STM transactions in each task. Figure 10.2 shows the result of our first set of experiments.

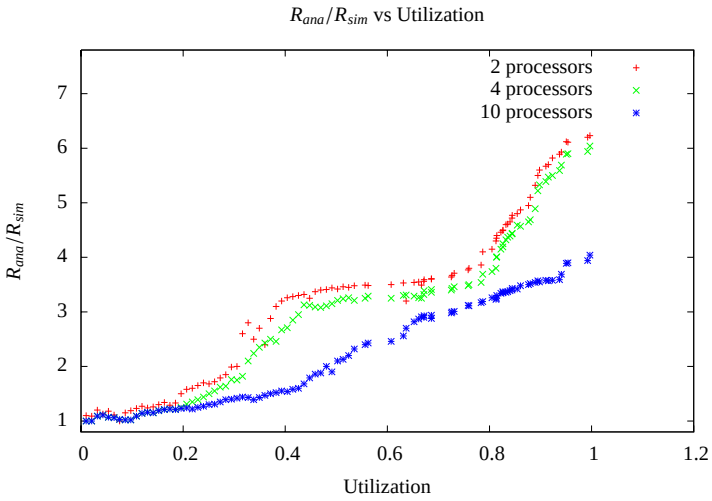

Figure 10.2: Uniprocessor: Ratio vs. Utilization

The utilization depicted on the x-axis is derived from $\sum_{\forall i} \frac{C_i+m_i s+s}{T_i} \leq 1$ [7]. Therefore, ideally, all tasks should meet their deadline for all utilizations at or below one. In this experiment, we studied three different systems. In the first system, only two processors exist and the tasks make remote invocations to either one at random. The utilization, in all three case we studied, is defined as the *maximum* utilization experienced by any node in the system. The other systems have four and ten processors respectively.

In all our experiments, the response time derived from the proposed analysis is higher than that of the simulation. This can be seen from the fact that the ratio R_{ana}/R_{sim} never falls below one in Figure 10.2. Also, the ratio becomes worse as system load increases. This occurs because as the system gets more loaded, the number of interruptions increases and hence the pessimism of the analysis increases (since we assume each interruption will result in a retry). Also, as the number of processors in the system increases, the ratio becomes better. This occurs because the utilization measured on the x-axis is the *maximum* utilization experienced by any node. Therefore, it is possible for other nodes in the system to be lightly loaded, leading to less interferences and, thus, less pessimism in the analysis.

Also, the pessimism of the analysis depends on the cost of the transactional part of the code, s, relative to the execution time of the non-transactional part of the code C_{ij}. The larger the ratio of s to C_{ij}, the more pessimistic the analysis becomes, because the pessimism in the proposed analysis is in the number of retries. Increasing the weight of the retries in the analysis by increasing the cost of the transactional component of the code results in larger estimates of worst-case response times. Figure 10.3 shows the result of an experiment where we compare the performance of a system to other systems that have a value of s twice as large and half as large as the base system.

It can be seen that the larger values of s cause greater divergence from the simulation results.

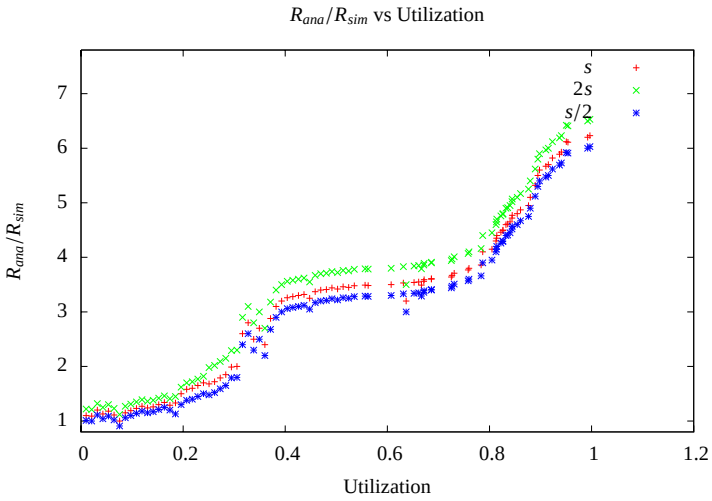

Figure 10.3: Uniprocessor: Ratio vs. Utilization

10.8 Conclusions

In this chapter we presented an algorithm for computing an upper bound on the response time of tasks in a distributed real-time system where concurrency control is managed using STM and nodes are subject to crash failures. We compared the result of our analysis to a simulation of the system in order to determine the efficacy of the proposed solution.

The result of this study indicates that it is possible to provide timeliness assurances for distributed systems programmed using STM. This allows for the first time, the usage of STM as a concurrency control mechanism (among others) for programming distributed real-time systems. Future research includes studying the different approaches for incorporating STM as outlined in [46] and dealing with some of its shortcomings. For example, we can consider whether buffered I/O can be used in STM code blocks and whether it is possible to eliminate the problem of weak atomicity using declarative languages or placing restrictions on the use of variables in imperative programming languages. Other areas of research include reducing the overhead of STM using software-hardware hybrid techniques and implementation optimizations. We also plan to consider aperiodic tasks and overload scheduling (i.e., providing assurances during overload conditions).

Chapter 11

Schedulability of Distributed Multiprocessor Systems with STM

11.1 Introduction

In this chapter, we present an algorithm for computing a worst-case bound on the response time of tasks in a real-time distributed multiprocessor system (we define a distributed multiprocessor system as a distributed system where each node is a multiprocessor), where failures may occur and concurrency control is managed using STM.

We consider using STM as the concurrency control mechanism for (non-I/O code in) distributed real-time systems. Toward this, we propose a method for computing an upper bound on the worst-case response time of periodic tasks, programmed using STM, running in a distributed real-time system that employs PFair [9].

11.2 Roadmap

In this chapter, we provide timeliness assurances for multiprocessor distributed systems programmed using STM. Pfair scheduling is an optimal scheduling algorithm for multiprocessor real-time systems during underload conditions [9]. Therefore, we propose an algorithm for computing an upper bound on the worst-case response times of tasks running on distributed multiprocessor systems scheduled using the Pfair discipline. Towards that end, we first show how Pfair scheduling on a single processor can be represented as EDF scheduling of the transactional model proposed in [108]. We then show how an upper bound can be placed on Pfair scheduled systems on a multiprocessor and extend this result using holistic analysis to deal with distributed systems. We then show how to incorporate the retry overhead of STM into the analysis. Finally, we show how our analysis can be extended to deal with crash

failures.

11.3 Pfair Scheduling on a Single Processor

We consider periodic tasks scheduled using the Pfair discipline on a single processor. Each task, τ_i, is characterized by its period, T_i, and its execution time C_i. We assume that the deadline of each task is equal to its period. The idea of Pfair scheduling [9] is to divide the processor time among the tasks in proportion to their rates (defined as $wt(\tau_i) = C_i/T_i$). To do this, a task, τ_i, is subdivided into several quanta sized subtasks, τ_{ij}, pseudo-release times, $r(\tau_{ij})$, and pseudo-deadlines, $d(\tau_{ij})$, are derived for these subtasks and then they are scheduled using the EDF discipline (ties are broken using tie breaking rules [9]).

In the rest of the chapter, we refer to pseudo-deadlines and pseudo-release times as simply deadlines and release times for simplicity. For synchronous tasks, the deadlines and release times of subtasks can be derived using the following equations:

$$r(\tau_{ij}) = \left\lfloor \frac{j-1}{wt(\tau_i)} \right\rfloor \quad (11.1) \ , \quad d(\tau_{ij}) = \left\lceil \frac{j}{wt(\tau_i)} \right\rceil \quad (11.2)$$

For asynchronous tasks, assume that task τ_i releases its first subtask at time r and let τ_{ij} $(j \geq 1)$ be this task. The release time and deadline of each subtask τ_{ik} $(k \geq j)$ can be obtained by computing the term $\Delta(\tau_i) = r - \lfloor (j-1)/wt(\tau_i) \rfloor$ and adding it to Equations (11.1) and (11.2).

In [108], the authors show how it is possible to perform schedulability analysis for EDF systems programmed using "transactions". A transaction, as used in [108], is a sequence of tasks that belong to a single programming context. A sort of precedence constraint is placed on the relative execution times of these tasks by using offsets and jitters. It can be easily shown that this transactional model can be used to represent tasks scheduled using the Pfair discipline.

Specifically, since each task in Pfair scheduling is subdivided into subtasks and these subtasks belong to the same execution context, we can represent each task as a transaction. It now becomes necessary to obtain values for the jitter and offsets to specify the precedence constraints of the subtasks (i.e, τ_{ij} can only start executing after τ_{ij-1} has completed executing). In Section 11.3.1 we show how this is performed.

11.3.1 Application to Pfair scheduling

In this Section, we show how the analysis of [108] can be applied to Pfair scheduled systems. As mentioned before, in Pfair scheduling, each task, τ_i, is subdivided into several quantum length subtasks, τ_{ij}. Therefore, we first need to determine the scheduling parameters of these subtasks. The execution time of each subtask is one quantum (i.e., $C_{ij} = Q$, where Q is the

duration of a scheduling quantum). To keep the analysis simple, we shall assume $Q = 1$, it is trivial to extend this analysis for $Q \geq 1$. Each subtask retains the period of its parent task, T_i, and d_{ij} is set to the value of Equation (11.2). We now turn our attention to the offset, ϕ_{ij}, and jitter, J_{ij} of our subtasks.

The initial values of the offsets, ϕ_{ij} are set as follows:

$$\phi_{ij} = \begin{cases} \sum_{\forall k<j} C_{ik} & \text{if } j > 1 \\ 0 & \text{if } j = 1 \end{cases} \tag{11.3}$$

and all jitters are set to the pseudo-release times of each subtask as computed in Equation (11.1), i.e., $J_{ij} = r(\tau_{ij})$. We then perform the analysis in [108] and update jitters, after the analysis, as follows:

$$J_{ij} = \begin{cases} R_{ij-1} - \phi_{ij} & \text{if } j > 1 \\ 0 & \text{if } j = 1 \end{cases} \tag{11.4}$$

where R_{ij} is the response time of τ_{ij}. This process is repeated until two successive iterations produce the same response time, at which point we have the worst-case response time of the tasks in the system.

11.4 Multi-processors

Now that we have established that Pfair scheduling can be represented as EDF scheduling using a transactional model, we turn our attention to providing an upper bound on the worst-case response time of Pfair scheduled multi-processor systems. The following theorem shows how this can be done

Theorem 56 (Theorem 6 in [16])**.** *An upper bound on the response time of a task τ_k in an EDF-scheduled multiprocessor system can be derived by the fixed point iteration on the value R_k^{ub} of the following expression, starting with $R_k^{ub} = C_k$:*

$$R_k^{ub} \leftarrow C_k + \left\lfloor \frac{1}{m} \sum_{i \neq k} I_k^i(R_k^{ub}) \right\rfloor \tag{11.5}$$

with $I_k^i(R_k^{ub}) = \min(W_i(R_k^{ub}), J_k^i(D_k), R_k^{ub} - C_k + 1)$

The proof of Theorem 56 can be found in [16]. The term $W_i(R_k^{ub})$ is the maximum workload offered by τ_i during a period of duration R_k^{ub}, $J_k^i(D_k)$ is the maximum number of interferences

by τ_i that can occur before the deadline of τ_k, and $R_k^{ub} - C_k + 1$ is a natural upper bound on the interference of any task on τ_k (because the response time, R_k^{ub}, is naturally composed of a period of time during which τ_k executes, C_k, and some interferences $R_k^{ub} - C_k + 1$). In order to take advantage of Equation (11.5), we need to derive expressions for these terms using our transactional model.

First, let us consider the term $W_i(R_k^{up})$. Before presenting the analysis we change the notation to $W_i(R_{ab}^{up})$, since we will be analyzing τ_{ab}. The maximum workload offered by τ_i during a period of length R_{ab}^{up} is equal to the maximum number of jobs of τ_i that can execute during that period . Remember, however, that, in Pfair scheduling, each task, τ_i, is divided into several subtasks to create a transaction Γ_i. So, in essence, when computing the term $W_i(R_{ab}^{up})$, we are computing the worst-case contribution of transaction Γ_i. From the analysis in [108], we know that the worst-case contribution of Γ_i to the response time of a task being analyzed during a period of length R_{ab}^{up} occurs when one of its tasks τ_{ik} coincides with the start of the period. We also know that when τ_{ik} coincides with the beginning of the period, the worst-case contribution of a task τ_{ij} during a period of length R_{ab}^{up} is:

$$W_{ijk}(R_{ab}^{up}) = \left(\left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + \left\lceil \frac{R_{ab}^{up} - \varphi_{ijk}}{T_i} \right\rceil \right)_0 C_{ij} \tag{11.6}$$

where $\varphi_{ijk} = T_i - (\phi_{ik} + J_{ik} - \phi_{ij}) \mod T_i$. Thus, the worst-case contribution, workload, of a transaction Γ_i to the response time of a task τ_{ab} when τ_{ik} coincides with the beginning of the period is:

$$W_{ik}(R_{ab}^{up}) = \sum W_{ijk}(R_{ab}^{up}), \quad \forall j \in \Gamma_i \tag{11.7}$$

and the upper bound on the contribution of Γ_i is:

$$W_i^*(R_{ab}^{up}) = \max\left(W_{ik}(R_{ab}^{up})\right), \quad \forall k \in \Gamma_i \tag{11.8}$$

Likewise, we can determine a value for $J_k^i(D_k)$ from the analysis in [108]. Specifically:

$$J_{ijk}(D_{ab}) = \left(\left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + \left\lfloor \frac{D_{ab} - \varphi_{ijk} - d_{ij}}{T_i} \right\rfloor + 1 \right)_0 C_{ij} \tag{11.9}$$

$$J_{ik}(D_{ab}) = \sum J_{ijk}(D_{ab}), \quad \forall j \in \Gamma_i \tag{11.10}$$

$$J_k^i(D_k) = J_{ab}^i(D_{ab}) = J_i^*(D_{ab}) = \max\left(J_{ik}(D_{ab})\right), \quad \forall k \in \Gamma_i \tag{11.11}$$

Unlike in [108], we do not have to take into account the interference of tasks belonging to the transaction being analyzed for two reasons. First, by definition of Pfair scheduling,

our subtasks have precedence constraints. Thus, when analyzing a subtask we are sure that all previous subtasks have finished executing and all subsequent subtasks are yet to start. Second, since we assume that deadlines are equal to periods for transactions (see Section 11.3), we are sure that while analyzing a particular transaction its predecessor's deadline has already passed (and so it is no longer in the system) and its successor is yet to start (otherwise the period would have been over and the current transaction's deadline would have already passed).

Finally, we need to set the value of the offset and jitter for the subtasks in our analysis. We take a pessimistic approach and set the jitter of each subtask to one quanta after the deadline of its preceding subtask i.e.,

$$J_{ij} = \begin{cases} d_{ij-1} + 1 & \text{if } j > 1 \\ 0 & \text{if } j = 1 \end{cases} \tag{11.12}$$

and the offset of each subtask is set to the best case completion time of its predecessor as in Equation (11.3).

11.5 Distributed Multiprocessor Systems

Now that we have shown, in Section 11.4, how to obtain an upper bound on the response time of tasks scheduled using the Pfair discipline on a multiprocessor, we can extend our analysis to a distributed system. Specifically, we can use the variant of holistic analysis developed in [108], i.e., task offsets are initially set to the best-case completion time of their predecessor:

$$\phi_{ij} = \sum_{i \le k \le j} (\delta_{ik} + C_{ij}) + \delta_{ij} \quad \forall 1 \le j \le N_i \tag{11.13}$$

where δ_{ij} is the communication delay between nodes i and j. The jitters are all set to zero and the response time, R_{ij}, is computed using the analysis in Section 11.4. Then, jitters are modified as follows:

$$J_{i1} = 0 \tag{11.14}$$

$$J_{ij} = R_{ij-1} + \delta_{ij} - \phi_{ij} \quad \forall 1 < j \le N_i \tag{11.15}$$

Offsets remain unchanged. Essentially, this means that the jitters are modified so that each task, τ_{ij}, is released at most δ_{ij}, the communication delay, time units after the completion of its predecessor τ_{ij-1}. We then compute the response times again using the analysis in

Section 11.4. This process is repeated until the result of two successive iterations are the same, at which point we have obtained the response time for each task. If the response times do not diverge, the process above is guaranteed to converge to the solution since the process is monotonic in its parameters. Naturally, during the computation only the contribution of the tasks running on the same processor is taken into account when computing the response times.

11.6 Considering STM

For the sake of this analysis, we consider atomic regions programmed using STM (e.g, [63]). We assume that each atomic region is kept short and that all atomic regions have computation cost at most s. We now show how the retry overhead of these atomic regions can be incorporated into our analysis. In [72], Anderson *et. al.* show how the overhead of lock-free code can be incorporated into Pfair scheduled systems.

Here, we show how this analysis can be applied to atomic regions programmed using STM. We assume that any two atomic regions that execute concurrently can interfere with each other. We further assume that a retry can only occur at the completion of an atomic region (i.e., validation of the transaction is performed before it commits). The second assumption implies that the number of retries of an atomic region is at most the number of concurrent accesses to atomic regions by other tasks. Finally, we assume that each atomic region is small and spans, at most, two quanta. The last assumption makes sense since atomic regions are usually designed to be small in order to reduce the likelihood of interferences and hence retries.

The idea behind the analysis is to compute the worst-case overhead introduced by the retry behavior of atomic regions. This overhead is then added to the execution time of each task to compute its worst-case demand on the processor. Using these new execution times, the analysis in Sections 11.4 and 11.5 can be used to compute the response time on a distributed system.

We assume that each task, τ_i, has N_i atomic regions and accesses atomic regions at most AA_i times during each quantum. Thus, the maximum number of interferences that can occur to τ_a in a single quantum is:

$$I_a = \mathbf{maxsum}_{M-1}\{AA_i | i \neq a\} \tag{11.16}$$

where M is the number of processors. Also, since we assume that an atomic region can span at most two quanta, and hence can only be preempted once, the overhead introduced by a single access to an atomic region in τ_a can be computed as in Equation (11.17).

$$O_a^{one} = s + (2I_a + 1)s \qquad (11.17)\ , \quad O_a = O_a^{one} \times N_a \qquad (11.18)$$

The term $(2I_a+1)s$ in Equation (11.17), represents the overhead of retries. The $2I_a$ represents the maximum number of retries that may occur in the two quanta that the operation spans, and the 1 added to $2I_a$ represents the retry that may occur due to interference that occurred while the task was preempted in the middle of its atomic region (note that by our assumption this can only occur once). Now that we have computed the overhead of one atomic region, we can compute the overhead of the N_a atomic regions using Equation (11.18).

Thus, we can set the new execution time of task τ_a to $C_a = O_a + C_a$ and then perform the analysis of Sections 11.4 and 11.5 using this new value.

11.7 Handling Failures

In this section we show how the analysis in Section 11.4 can be extended to take failures into account. We assume that each quantum-sized subtask in the system has an associated exception handler that can be used to restore the system to a safe state in case of failure, and that this exception handler has execution time C^h_{ji} and relative deadline d^h_{ij}. The absolute deadline of the handler is relative to the time failure occurs, t_f, thus the absolute deadline of the handler is $D^h_{ij} = t_f + d^h_{ij}$. Since we cannot determine t_f a priori, we assume a worst-case scenario where each job executes to completion, using up as much processor time as possible, and then an error occurs that triggers its exception handler. It is this worst-case scenario from which we derive the deadline of the exception handlers.

11.7.1 Analysis

In order to incorporate exception handlers in the analysis, it is necessary to extend Equations (11.6) and (11.9) to take their overhead into account.

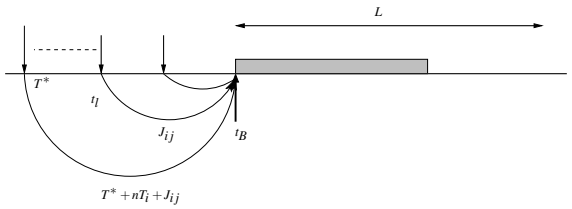

Figure 11.1: Scenario for calculating worst-case contribution

We assume that a critical instant occurs at time t_B (note that for this analysis we do not need to know the value of t_B, we just assume that it exists) and compute the worst-case contribution of the exception handlers during a period of length L starting at t_B. From [108], we know that there are

$$n_1 = \left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor \tag{11.19}$$

jobs of τ_{ij} that can start at the beginning of the period being studied after suffering some jitter. Each of these activations has an associated exception handler that can start, at most, at time $t_B + d_{ij}$. Thus, when computing the maximum number of interferences that can occur during a period of length L, we only consider these exception handlers if $d_{ij} < L$. There are

$$n_2 = \left\lceil \frac{L - \varphi_{ijk}}{T_i} \right\rceil \tag{11.20}$$

activations that will occur within a period of duration L. The latest start time of the exception handlers of these activations are

$$S = \varphi_{ijk} + (p-1)T_i + d_{ij} \qquad \forall p = 1 \cdots n_2 \tag{11.21}$$

We only consider exception handlers for which $S < L$. We also need to compute the overhead of exception handlers whose activations do not contribute to the overhead. This may occur when an activation finishes before the critical instant t_B, but its exception handlers execute after t_B. In Figure 11.1, the first activation that can be delayed to start at the beginning of the period being studied is depicted as starting at t_l.

However, if failures are not considered, jobs that start before t_l, such as the one depicted as starting at time T^* in Figure 11.1, do not contribute to the analysis because even if these jobs were delayed J_{ij}, their activation time would fall before t_B. It is, now, possible for these tasks to contribute to the worst-case response time if their exception handlers start after t_B. The latest start time of a handler whose job arrives at T^* is:

$$S = T^* + d_{ij} \tag{11.22}$$

If S is greater than or equal to t_B then the handler will contribute to the busy period. In other words:

$$T^* + d_{ij} \geq T^* + nT_i + J_{ij} \tag{11.23}$$

which gives us:

$$n < \frac{d_{ij} - J_{ij}}{T_i} \tag{11.24}$$

Since n is an integer, Equation (11.24) resolves to:

$$n = \left(\left\lceil \frac{d_{ij} - J_{ij}}{T_i} \right\rceil - 1 \right)_0 \tag{11.25}$$

The zero subscript indicates that negative values are considered zero (in this case, such an event indicates that none of the jobs starting before t_l can contribute to the busy period). Thus we can compute the contribution of the exception handlers of τ_{ij} to the response time of τ_{ab} in a period of duration L, when the task whose activation time coincides with t_B is τ_{ik} using Algorithm 16. Thus, Equation (11.6) becomes:

$$W_{ijk}(R_{ab}^{up}) = \left(\left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + \left\lceil \frac{R_{ab}^{up} - \varphi_{ijk}}{T_i} \right\rceil \right)_0 C_{ij} + W_{ijk}^h(R_{ab}^{up}) \tag{11.26}$$

and Equation (11.9) becomes:

$$\begin{aligned} J_{ijk}(D_{ab}) = & \left(\left\lfloor \frac{J_{ij} + \varphi_{ijk}}{T_i} \right\rfloor + \left\lfloor \frac{D_{ab} - \varphi_{ijk} - d_{ij}}{T_i} \right\rfloor + 1 \right)_0 C_{ij} \\ & + W_{ijk}^h(D_{ab} - d_{ij}^h) \end{aligned} \tag{11.27}$$

Algorithm 16: $W_{ijk}^h(L)$

```
1: sum = (⌈(d_ij − J_ij)/T_i⌉ − 1)_0 C^h_ij;
2: if d_ij < L then
3:     sum ← sum + ⌊(J_ij + φ_ijk)/T_i⌋ C^h_ij;
4: n_2 ← ⌈(L − φ_ijk)/T_i⌉;
5: for 1 ≤ p ≤ n_2 do
6:     S ← φ_ijk + (p − 1)T_i + d_ij;
7:     if S < L then
8:         sum ← sum + C^h_ij;
9: return sum;
```

The rest of the analysis remains unchanged.

11.8 Experiments

In this section, we perform a number of experiments to verify the validity of the analysis presented. In our first set of experiments, we determine whether or not the analysis presented in Section 11.4 can be used to derive suitable upper bounds for the response times of Pfair

scheduled tasks. Toward that goal, we conducted a number of experiments to determine the Deadline Satisfaction Ratio (or DSR) of the analysis in Section 11.4.

We perform the analysis for ten tasks on multiprocessors that contain 4, 6 and 8 processors. For each of these systems we fix the execution time of the tasks and vary the periods to obtain utilizations between 0 and m, where m is the number of processors. We ran our experiment 700 times and recorded the average DSR for each utilization. Figure 11.2 depicts the result of the experiments. The utilization on the x-axis is normalized with respect to the number of processors used.

The results indicate that the analysis is tighter for a smaller number of processors, but that it provides a good bound for response time in most cases (for example, the average DSR in our experiments does not drop below 0.8 until close to the 0.8, normalized, system utilization point). In the next set of experiments, we compute the ratio of the response time obtained using the proposed analysis to a system simulated using [107].

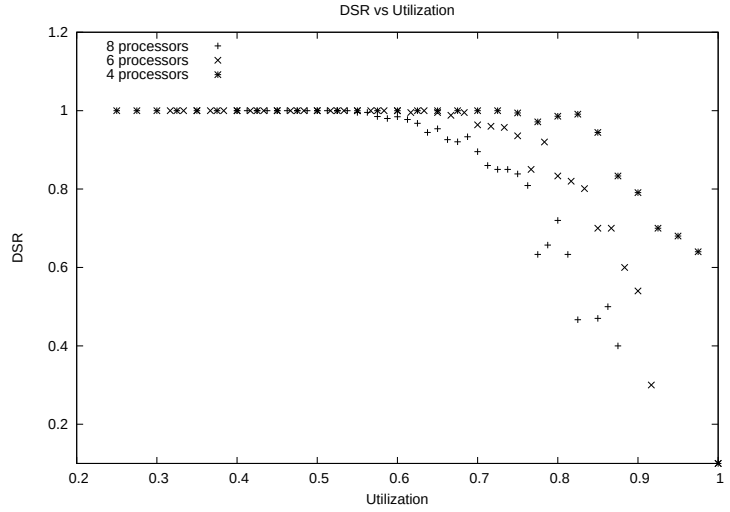

Figure 11.2: DSR vs. Utilization

For this experiment, we fixed the number of processors and nodes, fixed the execution times and varied the periods to obtain utilizations between 0 and m. Figure 11.3 depicts the result of our experiments. As can be seen, the response time analysis becomes more pessimistic as the value of s increases due to the dependence of the analysis on Equation (11.18). Other sources of pessimism include Equations (11.5) and (11.12).

11.9 Conclusions

We presented an algorithm for computing an upper bound on the worst-case response time for tasks on a multiprocessor distributed real-time system where concurrency control is pro-

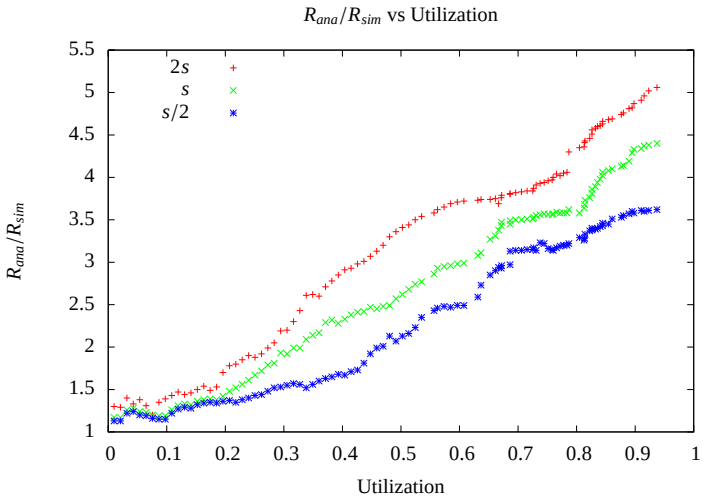

Figure 11.3: R_{ana}/R_{sim} vs. Utilization

grammed using STM.

With this result, it is now possible to include STM in the repertoire of real-time programming tools on such architectures. Future work includes tightening the analysis by considering slack (as in [16]), and considering non-periodic tasks and overload scheduling. Other directions include investigating other methods [46] for incorporating STM in distributed real-time systems.

Chapter 12

Contention Management in STM

12.1 Introduction

One of the important features of STM is that a base implementation need not provide any progress guarantee stronger than obstruction freedom — that a single thread executing in isolation will complete its execution within a bounded period of time. Obstruction freedom is more of a correctness guarantee than a progress guarantee since it specifies that the implementation should not hinder the progress of the thread when it is executing on its own, but says nothing about the progress behavior of a thread in the presence of interference.

This allows STM implementations to be relatively simple, although some authors have suggested that it is possible to further weaken obstruction freedom in order to improve performance [43]. In addition, it opens the door for users of the implementation to design orthogonal contention management policies that can be used to provide the progress guarantees required by their applications. In this chapter we do just that. We will show how contention managers can be designed to take real-time criteria into account in order to make the technology suitable for real-time systems and describe the properties of such contention managers, something that, to the best of our knowledge, has not been done before.

12.2 Previous work

Several authors have tackled the issue of using non-lock based concurrency control for real-time systems. Some of the most prominent works in this category include [6–8, 72, 97]. In [97], the authors show how it is possible to implemented a scaled down version of STM for a uniprocessor system where priority scheduling is employed. In [6–8], the authors develop a utilization based schedulability analysis for uniprocessor systems programmed using lock-free concurrency control. In [72], lock-free concurrency control is studied for a multiprocessor

machine where the PFair scheduling algorithm is used. More recently, there have been attempts to provide methods for bounding response times on distributed real-time systems programmed using STM [49,50]. These attempts are related to our work since they indicate that STM is a viable technology for real-time systems.

Perhaps the closest work to ours is that presented in [97]. The paper addresses the implementation of a scaled down version of STM for a uniprocessor system. The scaled down nature of the STM and the uniprocessor environment negates the need for contention management, and so that issue is not addressed in that paper. In this chapter, we attempt to address the issue of designing real-time contention managers for fully fledged implementations of STM on multi-core platforms. We do not implement an STM from scratch, but rather implement plug-in contention managers for the DSTM2 STM implementation from Sun.

There have also been several studies [58,59,119–122,141] performed on the issue of using appropriate contention managers, although none of these studies have attempted to investigate the possibility of using real-time criteria for managing contention. A particularly interesting approach is that taken in [13, 59]. These two works lay down the basis for a theoretical framework for CM. Specifically, they show that any deterministic contention manager that has the pending commit property (i.e that there is always a transaction that will not be aborted by others once it starts and will continue executing until it commits) can theoretically guarantee that all transactions finish execution within a bounded period of time and achieve a polynomial competitive ratio in terms of makespan when compared to an optimal offline scheduler.

This result is interesting for a number of reasons. Chief among them is that it lays the ground work for theoretically proving non-trivial properties for contention managers. This would be beneficial for real-time systems since it would introduce formal determinism into the analysis. Unfortunately, both the theoretical framework introduced and the example CM that is shown to have these properties are not applicable directly to real-time systems. The pending commit property effectively excludes all forms of dynamic deadline scheduling. In addition, makespan is not the metric to optimize in real-time systems, rather timeliness properties are the metric to optimize. In this chapter, we attempt to provide some theoretical basis for the real-time properties of the CMs we design. Specifically, we show how to upper bound the number of transactional retries when the EDF CM is used. Future work will try to lay a stronger framework for the theoretical properties of real-time CMs.

One other work we build on is [65] which describes Sun's DSTM2 STM implementation. DSTM2 is a flexible framework for STM that allows programmers to implement different contention managers and different transactional factories (the mechanism DSTM2 uses to provide atomic access to memory). This flexibility allows us to implement our real-time contention managers within the framework.

12.3 The Contention Managers

We develop a number of contention managers (CMs) that take real-time properties into account. The ideas behind these contention managers are simple, yet they provide better performance for real-time systems since they specifically take real-time constraints into account. In the rest of this section, we provide a brief overview of the contention managers we design.

12.3.1 Plain EDF CM

This is perhaps the simplest contention manager we design. As its name suggests, the EDF contention manager resolves contention in favor of transactions belonging to threads with the earliest deadline. Algorithm 17 depicts this contention manager.

Algorithm 17: EDFresolveConflict(Transaction me, Transaction other)

```
1: if other.isActive() then
2:     if me.Deadline ≤ other.Deadline then
3:         other.abort();
4:     else
5:         other.waitWhileActive();
6: else
7:     return;
```

Algorithm 18: TUFresolveConflict(Transaction me, Transaction other)

```
1: if other.isActive() then
2:     if me.PUD > other.PUD then
3:         other.abort();
4:     else
5:         other.waitWhileActive();
6: else
7:     return;
```

As can be seen, if a conflict occurs with an active transaction the deadline of both transactions are used to determine which transaction gets aborted. If the deadline of the current transaction is closer than the deadline of the conflicting transaction, the conflicting transaction is aborted. Otherwise the calling transaction (me) waits while the conflicting transaction (other) is active. Once the conflicting transaction (other) becomes inactive, (me) continues execution.

12.3.2 Plain TUF CM

The TUF contention manager attempts to accrue the most utility to the system by favoring high importance (or utility) transactions. The remainder of this section describes how the TUF scheduler attempts to maximize accrued utility.

Thread schedulers that use TUFs to schedule are referred to as Utility Accrual (UA) schedulers. UA schedulers attempt to schedule threads by employing heuristic criteria whose purpose is to maximize the utility accrued to the system. The most popular heuristic for unit-step TUFs is that of potential utility density (PUD) [31]. PUD is essentially the ratio between the remaining execution time for a thread and the utility that would be accrued to the system if it where to finish on time.

This ratio is an indication of the "return on investment" for each unit of computation time assigned to that particular thread. UA schedulers favor threads with high PUDs since it is assumed that their completion will result in the most benefit to the system (i.e. they are the threads that would offer the largest accrual in utility for the effort expended in executing them). We emulate these thread schedulers by employing PUD as the heuristics in the TUF contention manager developed in this chapter. When two or more transactions attempt to access the same memory location, the contention is resolved in favor of the thread with the highest PUD.

Algorithm 18 depicts the plain TUF contention manager used in this chapter. It is virtually identical to the plain EDF contention manager, but the conflict is resolved using PUD rather than deadlines. PUD is computed as described in this section.

12.3.3 Exponential Backoff/EDF Hybrid CM

We designed a hybrid contention manager that uses both the exponential backoff algorithm (ala the CDMA scheme of Ethernet) and the ideas of EDF to manage contention. Algorithm 19 depicts the algorithm.

As can be seen, when two transactions conflict, the contention manager first checks if the current transaction has exhausted its backoff attempts. If this is true, then the conflicting transaction is aborted regardless of deadline. If the current transaction has not exhausted its back off attempts, it first checks to see if its deadline is earlier than that of the conflicting transaction. If so, it backs-off for a short period of time (lines 4-6).

Notice that this is in contrast to the two previous algorithms which automatically aborted any conflicting transaction if their deadlines were earlier. If the deadline of the conflicting transaction is earlier, the current transaction backs-off according to the standard CDMA-like exponential backoff algorithm (lines 7-9). Note that the backoff period in this case becomes progressively higher on each attempt until it reaches an upper limit (lines 10-11).

Algorithm 19: EDFBackresolveConflict(Transaction me, Transaction other)

```
1:  if currentAttempts ≤ MAX_RETRIES then
2:      if other.isActive()=false then
3:          return;
4:      if me.Deadline ≤ other.Deadline then
5:          logBackoff ← min(HIGH_PRIO_BACKOFF,
6:          currentAttempts−2+MIN_LOG_BACKOFF);
7:      else
8:          logBackoff←currentAttempts−2+MIN_
9:          LOG_BACKOFF;
10:     if logBackoff>MAX_LOG_BACKOFF then
11:         logBackoff←MAX_LOG_BACKOFF;
12:     time←Random(1..logBackoff);
13:     me.Sleep(2^time);
14:     currentAttempts++;
15: else
16:     other.Abort();
```

Algorithm 20: EDFEruptresolveConflict(Transaction me, Transaction other)

```
1:  transferred← 0;
2:  attempts← 0;
3:  while true do
4:      attempts++;
5:      Δ ← other.getPriority() − me.getPriority();
6:      if Δ < 0 or attempts>attemptsLimit then
7:          transferred← 0;
8:          other.Abort();
9:          return;
10:     if me.getPriority()>transfered then
11:         other.setPriority(other.getPriority()+
12:         me.getPriority()-transferred);
13:         transferred←me.getPriority();
14:     if attempts < attemptsLimit then
15:         me.Sleep(SLEEPTIME);
```

This algorithm allows us to favor high priority threads (those with earlier deadlines) while at the same time allowing us to curtail the thrashing that occurs when lower priority transactions are always aborted with no backoff interval used by making threads with deadlines further away more "polite". This "politeness" is manifest in the fact that transactions belonging to less urgent threads will differ to a conflicting transaction for a longer period of time. However, as their deadlines draw near, these threads become "impatient" and therefore differ for shorter periods of time.

12.3.4 Exponential Backoff/TUF Hybrid CM

We also designed a hybrid exponential backoff/TUF contention manager. It is essentially the same as Algorithm 19 but in line 4, PUD rather than deadlines is used to manage the behavior of the contention manager. Specifically, transactions belonging to higher PUD threads are less "polite" than transactions that belong to threads with lower PUD.

12.3.5 Eruption/EDF Hybrid CM

In this section we describe the idea behind a hybrid Eruption/EDF contention manager. The main idea behind the Eruption contention manager [119] is that each transaction starts with some base priority. Each time the transaction opens an object, its priority is incremented. Thus allowing transactions holding a large number of system resources to gain higher priority. Conflicts are resolved in favor of the highest priority transaction. When a low priority conflicting transaction arrives, it propagates its priority to the transaction it is waiting for.

This allows the transaction being waited on to gain the priority of all the transactions that are waiting for it, thus allowing it to "erupt" past the transaction blocking it. A low priority thread only does this for a limited number of times (attemptsLimit in line 6 of Algorithm 20), after that it just aborts the conflicting transaction. We keep the same semantics in the hybrid CM we design, but the base priorities are assigned in inverse proportion to deadlines (i.e. earlier deadlines are associated with larger priorities). Algorithm 20 depicts this algorithm.

12.4 Algorithm Properties

Here we state, in a semi-formal fashion, some of the properties of the EDF CM, with particular focus on real-time properties. It should be noted that we consider deadlines to equal periods in this analysis.

Claim 57. *In a hard real-time system with the plain EDF CM, for a thread T_i, the worst-case number of times that its transactions can be interfered with, and hence cause retries,*

is $\sum_{T_j \in \gamma_i} \left(\left\lfloor \frac{t(T_i)}{t(T_j)} \right\rfloor + 1 \right) \beta_{i,j}$*, where* $t(T_i)$ *is the period of thread* T_i*,* γ_i *is the set of threads that share an STM object with thread* T_i *and* $\beta_{i,j}$ *is the number of accesses to transactional objects in* T_i *that are shared with* T_j*.*

Proof. Assume we are dealing with the time interval $[t, t+\Delta)$. We need to determine the number of invocations, p, belonging to thread T_j that can cause a retry in an invocation of thread T_i during that period. In hard real-time systems, an invocation executes for at most $t(T_i)$. Therefore, the interval we wish to consider is $[t, t+t(T_i))$.

The deadline of the earliest invocation of T_j that can execute in $[t, t+t(T_i))$ is just after t (otherwise the invocation would have left the system by the time the interval we are considering arrives), and hence its release time is after $t - t(T_j)$. Therefore, the interval we wish to consider is $[t-t(T_j), t+t(T_i))$ which is equivalent to $[t, t+t(T_i)+t(T_j))$. Without loss of generality, let us assume that $t = 0$, giving us the inverval $[0, t(T_i)+t(T_j))$.

We now need to compute the number of invocations of thread T_j that can start within this interval such that their deadline is less than the upper limit of the interval $t(T_i)+t(T_j)$ (we shall designate this Δ). The reason that we only consider invocations with deadline less that Δ is that the EDF CM will only allow those invocations to cause interference. Invocations with deadline after Δ, even if they are executing on other processors in the multiprocessor system, will not cause an interference since the EDF CM will abort them when a conflict occurs.

This can be computed by noting that the last invocation of T_j that satisfies this condition is the only one that satisfies both of the following inequalities:

$$(p-1)t(T_j) + t(T_j) \leq \Delta \text{ and } pt(T_j) + T_j > \Delta$$

which gives us

$$p \leq \frac{\Delta}{t(T_j)} \text{ and } p > \frac{\Delta}{t(T_j)} - 1$$

which gives $p = \left\lfloor \frac{\Delta}{t(T_j)} \right\rfloor$ since $\Delta = t(T_i) + t(T_j)$, this term becomes $p = \left\lfloor \frac{t(T_i)}{t(T_j)} \right\rfloor + 1$. We now bound the number of retries as (number of jobs of T_j that can potentially execute in the interval with earlier deadline than T_i)$\times$ (number of accesses for T_i to transactional objects shared with T_j), which gives

$$\left(\left\lfloor \frac{t(T_i)}{t(T_j)} \right\rfloor + 1 \right) \beta_{i,j}$$

Summing up for all threads that share STM objects with T_i completes the proof.

□

Claim 58. *The worst-case execution time for each invocation of a thread T_i when the EDF CM is used and the cost of a transactional retry is s is: $C_{full} = C_i + \sum_{T_j \in \gamma_i} \left(\left\lfloor \frac{t(T_i)}{t(T_j)} \right\rfloor + 1 \right) \cdot \beta_{i,j} \cdot s$, where C_i is the worst-case execution time of the thread without any retries.*

Proof. This follows directly from Claim 57. For each of the interferences derived in Claim 57, one retry cost s is incurred. □

Using Claim 58, it is possible to use the standard schedulability analysis techniques for global EDF to determine the schedulability of a system scheduled using the global EDF policy and STM with plain EDF contention management.

Claim 59. *When global EDF scheduling is used, the performance achieved using STM with EDF CM is better than or equal to the performance achieved using the lock-free approach described in [38] if $s \leq \frac{\sum_i \left(\sum_{T_j \in \gamma_i} \left(\lceil t(T_i)/t(T_j) \rceil + 1 \right) / t(T_i) \right)}{\sum_i \left(\sum_{T_j \in \gamma_i} \left(\lfloor t(T_i)/t(T_j) \rfloor + 1 \right) / t(T_i) \right)} \cdot r$ where s is the cost of one STM transactional retry, r is the cost of a retry loop of the lock-free approach described in [38] and we assume that $\beta_{i,j} = \beta$ for all i and j.*

Proof. Since we are considering global EDF, the same schedulability criteria apply for the two approaches. Therefore their relative merits can be determine by determining which approach places the most computing demand on the system. From Claim 58, we know that the worst-case execution time of a thread is upper bounded by $C_{full} = C_i + \sum_{T_j \in \gamma_i} \left(\left\lfloor \frac{t(T_i)}{t(T_j)} \right\rfloor + 1 \right) \cdot \beta_{i,j} \cdot s$ when STM with EDF CM is used. From Equation 11 in [38], we know that the worst-case execution time of a thread is upper bounded by $C_{full} = C_i + \sum_{T_j \in \gamma_i} \left(\left\lceil \frac{t(T_i)}{t(T_j)} \right\rceil + 1 \right) \cdot \beta_{i,j} \cdot r$ when the lock-free approach is used. Therefore, the STM with EDF CM solution places a utilization load on the system of $\sum_i \left(C_i + \sum_{T_j \in \gamma_i} \left(\left\lfloor \frac{t(T_i)}{t(T_j)} \right\rfloor + 1 \right) \cdot \beta_{i,j} \cdot s \right) / t(T_i)$ while the lock-free approach in [38] places a load on the system of $\sum_i \left(C_i + \sum_{T_j \in \gamma_i} \left(\left\lceil \frac{t(T_i)}{t(T_j)} \right\rceil + 1 \right) \cdot \beta_{i,j} \cdot r \right) / t(T_i)$.

The STM solution is at least as good as the lock-free solution if it places less or equal load on the system. Assuming $\beta_{i,j} = \beta_{l,m}$ for all i, j, l and m, this implies that

$$s \leq \frac{\sum_i \left(\sum_{T_j \in \gamma_i} \left(\lceil t(T_i)/t(T_j) \rceil + 1 \right) / t(T_i) \right)}{\sum_i \left(\sum_{T_j \in \gamma_i} \left(\lfloor t(T_i)/t(T_j) \rfloor + 1 \right) / t(T_i) \right)} \cdot r$$

□

Corollary 60. *For $s \leq c_1 \cdot r$, STM synchronization with EDF CM is always at least as good as the lock-free solution proposed in [38]. And $1 < c_1 < 2$, for threads with relatively prime periods, which implies that the STM with EDF CM solution can tolerate retry costs larger than those of the solution in [38] and still remain competitive.*

Proof. The term $\sum_{T_j \in \gamma_i} \left(\lceil t(T_i)/t(T_j) \rceil + 1 \right) / t(T_i))$ in Claim 59 can be expressed as $\sum_{T_j \in \gamma_i} ((x_i + 1) + 1) / t(T_i)$, and the term $\sum_{T_j \in \gamma_i} \left(\lfloor t(T_i)/t(T_j) \rfloor + 1 \right) / t(T_i)$ can be expressed as $\sum_{T_j \in \gamma_i} ((x_i) + 1) / t(T_i)$ if we assume that $t(T_i)$ and $t(T_j)$ are coprime for all i and j. Which simplifies to $\sum_{T_j \in \gamma_i} (x_i) / t(T_i) + 2n/t(T_i)$ and $\sum_{T_j \in \gamma_i} (x_i) / t(T_i) + n/t(T_i)$ assuming $|\gamma_i| = n$ (i.e. all threads have at least one shared object access in common).Thus, the term $\sum_i \left(\sum_{T_j \in \gamma_i} \left(\lceil t(T_i)/t(T_j) \rceil + 1 \right) / t(T_i) \right)$ becomes $\sum_i \left(\sum_{T_j \in \gamma_i} (x_i) / t(T_i) + 2n/t(T_i) \right)$ which simplifies to $\sum_i \left(\sum_{T_j \in \gamma_i} (x_i) / t(T_i) \right) + 2n^2 \sum_i 1/t(T_i)$. Likewise, $\sum_i \left(\sum_{T_j \in \gamma_i} \left(\lfloor t(T_i)/t(T_j) \rfloor + 1 \right) / t(T_i) \right)$ simplifies to $\sum_i \left(\sum_{T_j \in \gamma_i} (x_i) / t(T_i) \right) + n^2 \sum_i 1/t(T_i)$.

Thus the ratio $\frac{\sum_i \left(\sum_{T_j \in \gamma_i} \left(\lceil t(T_i)/t(T_j) \rceil + 1 \right) / t(T_i) \right)}{\sum_i \left(\sum_{T_j \in \gamma_i} \left(\lfloor t(T_i)/t(T_j) \rfloor + 1 \right) / t(T_i) \right)}$, becomes $\frac{\sum_i \left(\sum_{T_j \in \gamma_i} (x_i)/t(T_i) \right) + 2n^2 \sum_i 1/t(T_i)}{\sum_i \left(\sum_{T_j \in \gamma_i} (x_i)/t(T_i) \right) + n^2 \sum_i 1/t(T_i)}$. Assuming $\sum_i \left(\sum_{T_j \in \gamma_i} (x_i) / t(T_i) \right) = const_1$ and $n^2 \sum_i 1/t(T_i) = const_2$, the expression becomes $\frac{const_1 + 2const_2}{const_1 + const_2}$ which is upper bounded by 2 and lower bounded by 1.

□

Claim 61. *A transaction belonging to the real-time thread with the earliest deadline will always commit after a bounded period of time that is proportional to $O(n)$, where n is the number of conflicting transactions, when the plain EDF CM is used.*

Proof. The earliest deadline transaction will abort all n conflicting transactions in order to execute. It takes $O(n)$ to abort the n conflicting transactions. The Claim follows. □

Here we state some additional properties of the real-time CM managers.

Claim 62. *When the EDF/Backoff contention manager is used, the transaction belonging to the earliest deadline thread spins for a randomized amount of time with a maximum of $2^{HIGH\_PRIO\_BACKOFF}$ ns to resolve conflicts, while conflicting transaction with later deadlines spin for a randomized amount of time with a maximum of $2^{MAX\_LOG\_BACKOFF}$ ns, where $HIGH\_PRIO\_BACKOFF < MAX\_LOG\_BACKOFF$. All threads, regardless of deadline, abort conflicting transactions after they have tried to back off for $MAX\_RETRIES$ attempts.*

Proof. This follows directly from lines 4-11 and line 1 in Algorithm 19. □

Corollary 63. *The EDF/Backoff contention manager favors transactions belonging to early deadline threads by making them wait, on average, for a shorter period of time than transactions belonging to threads with later deadlines.*

Claim 64. *The EDF contention manager provides wait-free progress semantics, for fault free systems.*

Proof. A wait-free system is one that is guaranteed to make progress in a bounded number of steps, and no thread may starve (i.e. no thread can be perpetually denied the ability to

commit its transactions). By Claim 61, the EDF contention manager guarantees that the earliest deadline thread in the system will complete in a bounded period of time. When this thread completes, the thread with the next earliest deadline is guaranteed to complete execution in a bounded period of time by Claim 61, etc. No threads will starve, since each thread is allowed to commit its transactions when it is the earliest deadline thread in the system (the last thread allowed to do so will be the thread with the furthest deadline).

The fault free caveat is necessary, since if a transaction belonging to some thread with an early deadline fails for some reason, the CM will keep resolving conflict in its favor thus violating the rule of no starvation for other threads. □

Claim 65. *When the plain TUF contention manager is used, the transaction belonging to the highest PUD thread (at the time conflict occurs) commits in a bounded period of time proportional to $O(n)$, where n is the number of conflicting transactions.*

Proof. Proof similar to that of Claim 61. □

Claim 66. *When the TUF/Backoff contention manager is used, the transaction belonging to the thread with the highest PUD (when the conflict occurs) spins for a randomized amount of time with a maximum of $2^{HIGH\_PRIO\_BACKOFF}$ ns to resolve conflicts, while conflicting transaction spin for a randomized amount of time with a maximum of $2^{MAX\_LOG\_BACKOFF}$ ns, where $HIGH\_PRIO\_BACKOFF < MAX\_LOG\_BACKOFF$. All threads, regardless of PUD, abort conflicting transactions after they have tried to back off for $MAX\_RETRIES$ attempts.*

Proof. Similar proof to Claim 62. □

Corollary 67. *The TUF/Backoff contention manager favors transactions belonging to high PUD threads by making them wait, on average, for a shorter period of time than transactions belonging to threads with lower PUDs.*

Claim 68. *The plain TUF contention manager provides lock-free progress semantics.*

Proof. A lock-free system is one in which overall system progress is guaranteed, but where the starvation of some thread is possible. When conflict occurs, the plain TUF contention manager resolves the conflict in favor of the transaction belonging to the highest PUD thread. This ensures that the highest PUD thread will always make progress towards completion. On the other hand, the arrival of high utility threads can result in starvation of a low PUD thread as its transactions are always aborted in favor of the arriving high PUD thread.

This is in contrast to Claim 64, where, since contention is resolved using deadlines, a thread cannot starve because there will eventually come a time when its absolute deadline is the earliest even if new threads with short relative deadlines keep arriving. □

Claim 69. *When the EDF/Eruption CM is used, contention is resolved in favor of the transaction belonging to the earliest deadline thread in at most $O(n)$ time if there are n conflicting transactions.*

Proof. The highest priority transaction (the transaction belonging to the earliest deadline thread) always aborts conflicting transactions immediately (lines 5-9 of Algorithm 20). If there are n conflicting transactions, each of them are aborted in constant time thus resolving the conflict in favor of the transaction belonging to the earliest deadline thread in at most $O(n)$ time. □

Claim 70. *Transactions belonging to threads other than the earliest deadline thread get conflict resolved in their favor in, at most,* $attemptsLimit \times SLEEPTIME + O(n)$ *when the EDF/Eruption CM is used and there are* n *conflicting transactions.*

Proof. The attempts made by a transaction to commit are kept track of (line 4 in Algorithm 20). When the transaction has made $attemptsLimit$ attempts, it gives up on waiting and aborts the conflicting transaction (lines 6-9). At which point, each of the conflicting transactions are aborted in constant time (i.e. in $O(n)$). If the $attemptsLimit$ has not been reached, the transaction waits for $SLEEPTIME$ before making another attempt (lines 14-15), giving us a total waiting time of $attemptsLimit \times SLEEPTIME$. Adding these two terms gives us $attemptsLimit \times SLEEPTIME + O(n)$. □

Corollary 71. *The EDF/Erutption CM favors transactions belonging to earlier deadline threads.*

Proof. This follows directly from Claims 69 and 70. □

Claim 72. *If a number of transactions are blocked on a high priority transaction, priority inheritance increases the priority of the head of the blocked list as the other transactions add their priority to it until its priority is higher than the priority of the transaction they are blocked on and it "erupts" past it.*

Proof. This follows directly from lines 10-13 in Algorithm 20. □

Proof. When the plain EDF CM is used, the transaction belonging to the earliest deadline thread will always abort conflicting transactions (lines 2-3 in Algorithm 17). If there are n conflicting transactions, they will all be aborted in $O(n)$ time. The winning transaction will then execute to completion in constant time C; therefore the transaction will commit in $O(n) + C = O(n)$ time. □

Corollary 73. *The response times of the earliest deadline thread will be the least affected by runtime jitter introduced by transactional retries when the plain EDF contention manager is used.*

Proof. By Claim 61, the transactions belonging to the earliest deadline thread will execute in bounded time by aborting transactions belonging to other threads. Therefore, these transactions will not be aborted by other threads and will therefore not need to be retried. □

12.5 Experiments

We perform a series of experiments to ascertain the performance of our contention managers. The platform we use is a 2.16GHz dual-core machine with 2GB of RAM installed. The machine runs a 2.6.29-rc7 Linux kernel with the PREEMPT patch applied. The experiments were conducted using Sun's RTSJ implementation, Java RTS. Java RTS has support for only one thread scheduling algorithm — preemptive fixed priority scheduling. Since EDF provides better CPU utilization than fixed priority scheduling, our first step was to augment the scheduling policy provided by Java RTS to include support for global EDF (G-EDF) scheduling. We implement a global EDF scheduler using the ideas presented in [88]. Threads start with priority HIGH, then call a reschedule method which assigns them a priority of either LOW or MEDIUM based on the global EDF scheduling rules.

In this work we use RealtimeThread, the "soft" real-time thread class of RTSJ, as the base for our derived thread classes since the DSTM2 library utilizes some heap memory and it would be a daunting task to re-write the entire library to use NoHeapRealtimeThread. We assign a priority lower than MEDIUM but higher than LOW to the GC to ensure that it does not interfere with critical real-time threads (which we define as threads with priority of MEDIUM or higher). We also reserve 10MB of memory for the critical real-time threads. We consider deadlines to equal periods.

12.5.1 STM Experimental Setup

In this section, we describe the experimental setup used to evaluate the performance of the contention managers described in Section 12.3. The first set of experiments conducted consist of a number of real-time threads (from 10 to 35) executing a number of STM operations. We consider two different microbenchmarks in our first set of experiments, a simple linked list and a Red/Black Tree. Three different operations are defined for these data structures; 1) insert, 2) remove and 3) contains.

Each of the threads in the system is periodic and executes for four periods. During each period, the thread attempts to perform 5 operations on the data structure (linked list/RB Tree), 60% of these operations are modify operations. The periods of the threads start at $120ms$ and increase in increments of $20ms$ for each additional thread. We ran the experiments 50 times for when there are 10, 15, 20, 25, 30 and 35 threads in the system. We record the average value for these runs, together with the 95% confidence interval, for two metrics. The first of these metrics is the Transactional Accrued Utility Rate (TAUR) of the contention managers. Assuming that each thread, T_i, has an assigned utility of $T_i.U$ and that the percentage of transactions issued by a thread that commit is designated $T_i.Comp$, we compute the TAUR as follows: $TAUR = \frac{\sum_{\forall i} T_i.U \times T_i.Comp}{\sum_{\forall i} T_i.U}$.

This metric is a weighted index of committed transactions with the weight equal to the

utility of the thread the transaction belongs to. This gives us an indication of the ability of the contention manager to favor high utility threads. Figure 12.5.1 depicts the result of this experiment.

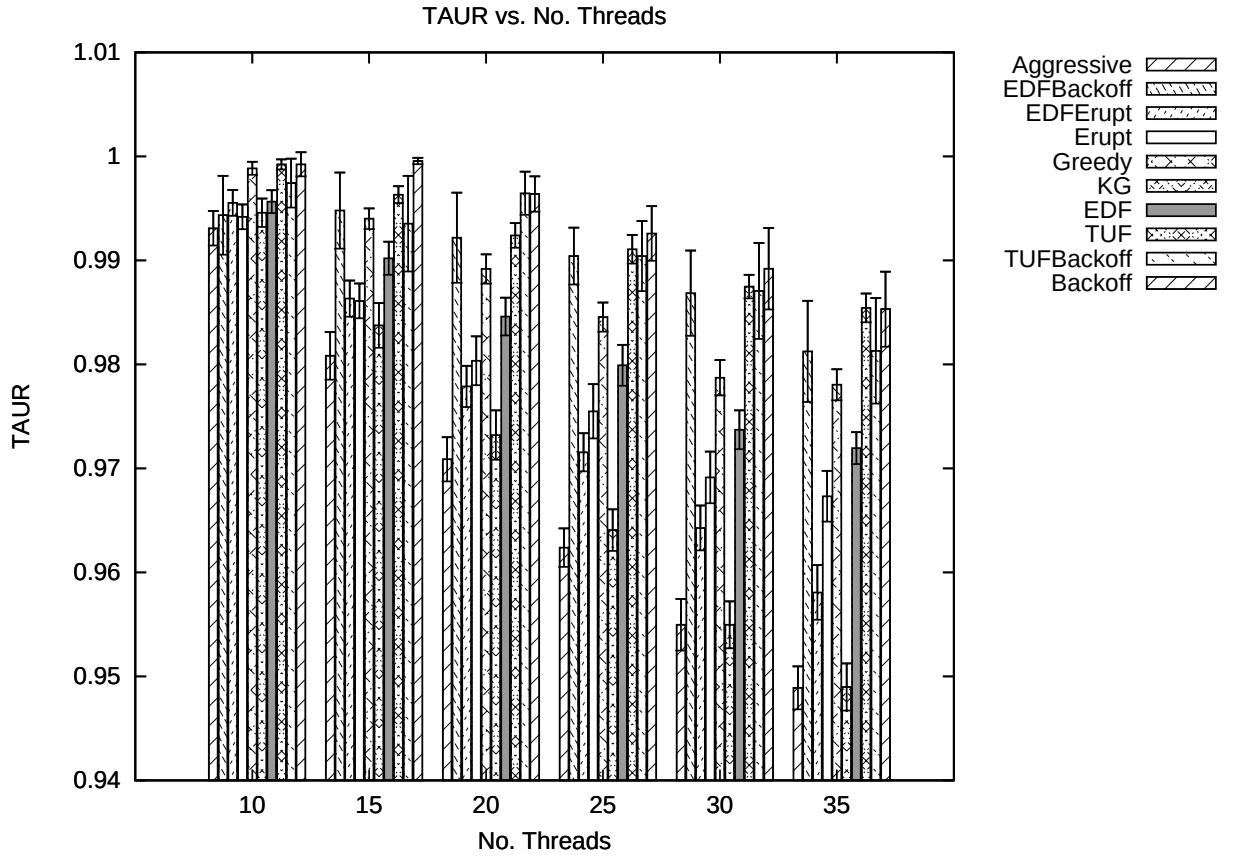

Figure 12.1: TAUR vs. No. Threads (list)

As can be seen in Figure 12.5.1, the real-time contention managers outperform other contention managers with the exception of the exponential backoff manager. The exponential backoff manager manages to hold its own against CMs specifically designed to take real-time criteria into account because it allows a lot of threads to execute (i.e. it increases throughput) unlike the real-time contention managers which favor earlier deadlines/higher PUDs. However, at very high contention levels (as will be seen in the remaining set of experiments) the backoff manager loses some of its advantage.

It should also be noted that the confidence interval for the real-time contention managers (especially for the EDF contention manager) are tighter than those for non-real-time contention managers. Thus indicating that the results they produce are more repeatable (since they are the result of a deliberate discrimination utilizing real-time criteria). The Kindergarten CM has the worst performance in this set of experiments since it provides equal "play time" to all the threads contending for shared memory, ignoring the fact that real-time systems

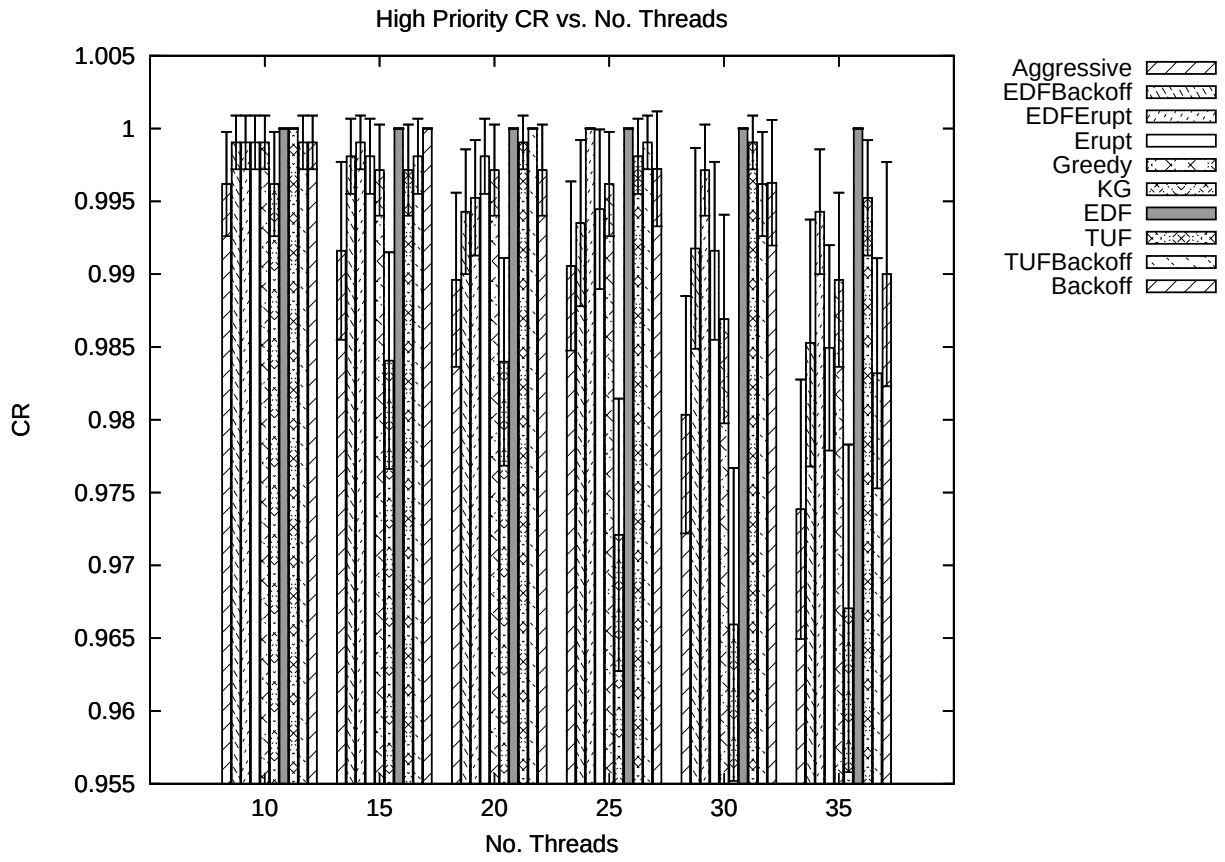

Figure 12.2: Percentage Committed (list)

require "unfair play" with the most urgent threads/highest PUD threads being "bullies" and receiving favorable treatment.

In the next set of experiments, we assigned thread utilities in inverse proportion to deadlines (thus the thread with the smallest relative deadline is assigned the highest utility) and we recorded the percentage of transactions started by the earliest deadline/highest PUD thread that actually committed. The results are depicted in Figure 12.5.1. As can be clearly seen, the real-time CMs perform the best according to this criteria. In particular, it should be noted that the backoff manager, especially at higher thread densities, performs worse than its TAUR results might suggest. Thus clarifying the fact that its high TAUR performance is due to accruing utility by minimizing the overall number of aborted transactions (thrashing) rather than favoring high PUD/early deadline threads while allowing thrashing in other threads.

We repeated the same set of experiments for Red/Black Trees and show the results in Figures 12.5.1 and 12.5.1. The real-time contention managers, especially the EDF CM, have a more obvious advantage in this experiment since the RB tree provides more opportunity for contention and thus showcases their ability to use real-time criteria to manage contention.

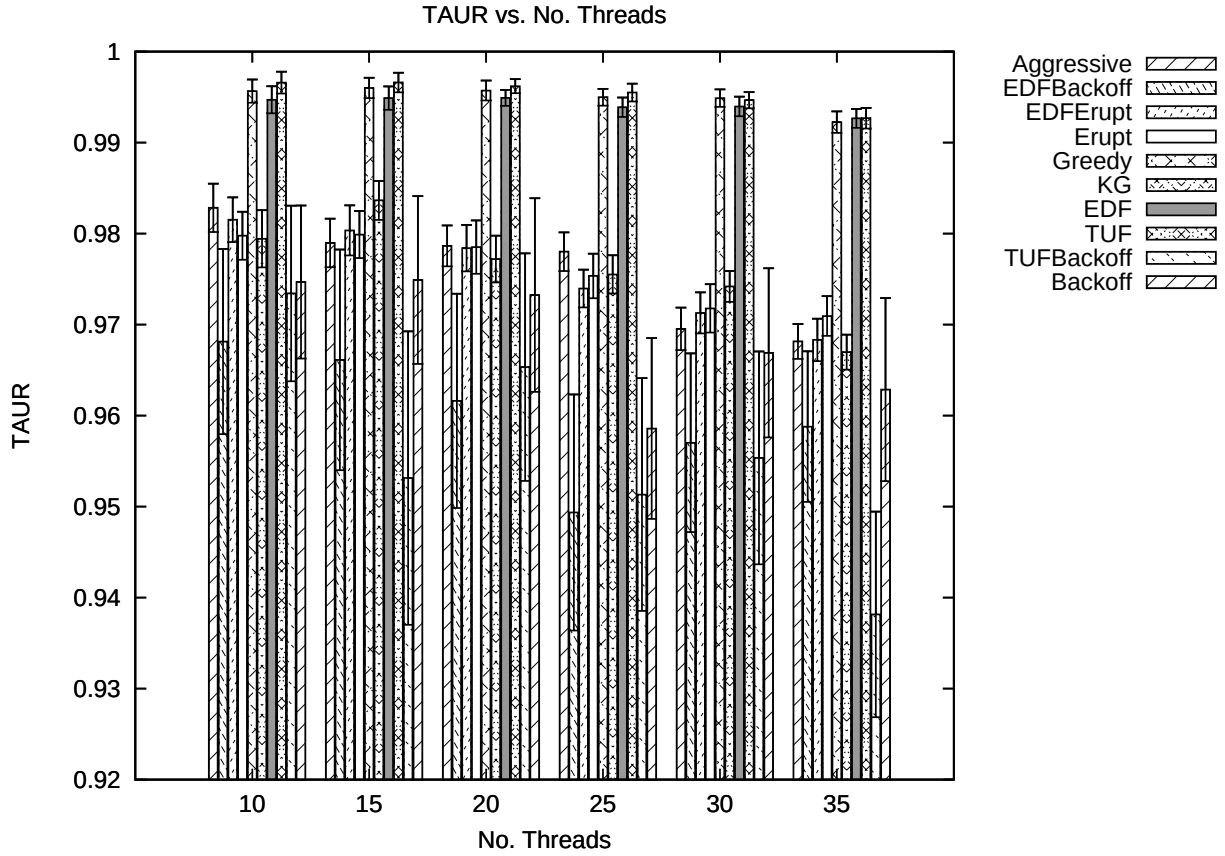

Figure 12.3: TAUR vs. No. Threads (RBTree)

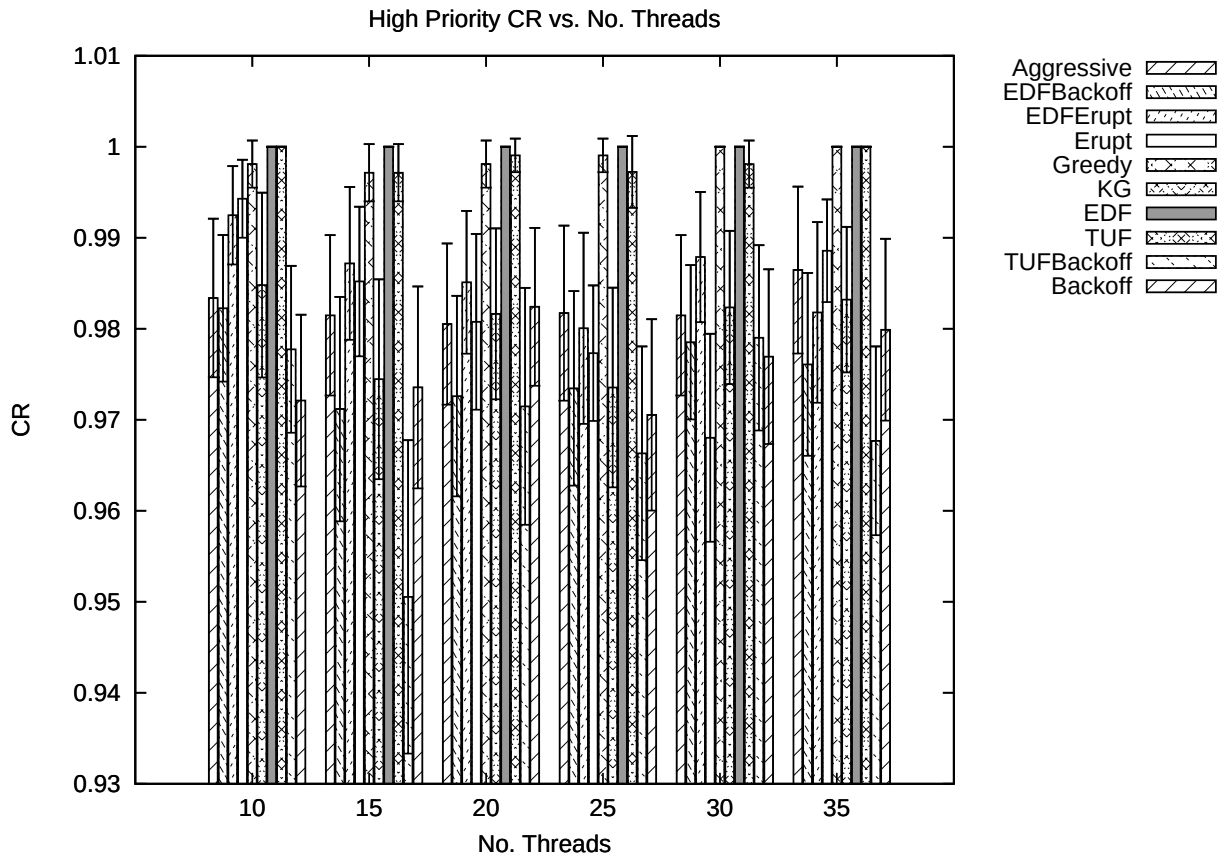

Figure 12.4: Percentage Committed (RBTree)

In order to further elucidate the behavior of the CMs, we conducted an experiment consisting of only two threads (Thread 8 and Thread 9, as reported by getId()). Each of these threads run for four periods performing 20 update transactions on a Skip List during each period (for a 100% update rate). The period of thread 8 is 320ms and the period of thread 9 is 340ms. We ran the experiment 50 times and plotted the commit rate (the percentage of transactions started by each thread that commits) as a time series. Figure 12.5 depicts the results of our experiment.

As can be seen, the real-time contention managers clearly favor Thread 8; its commit rate is higher and its suffers less jitter than thread 9. However, in the non-real-time CMs, the commit rates for the two threads are roughly equivalent and both suffer a similar amounts of jitter.

Worth noting is the fact that the TUF CM favors thread 8, but not to the extent of the EDF CM (this is also apparent from the results of previous experiments). This happens because the TUF manager will sometimes resolve contention of a thread with further away deadline if its PUD is higher than that of a thread with an earlier deadline. We conducted similar experiments for simple linked lists and Red/Black trees with similar, but less pronounced (since they offer less opportunity for contention), results. We do not include these results in this chapter due to space limitations.

12.6 Conclusions

In this chapter we presented a number of contention managers that use real-time criteria while arbitrating among conflicting transactions. We implemented the contention managers in Sun's DSTM2 STM implementation and tested their performance. The results of the experiments clearly indicate that contention managers specifically designed to take real-time criteria into account have better performance.

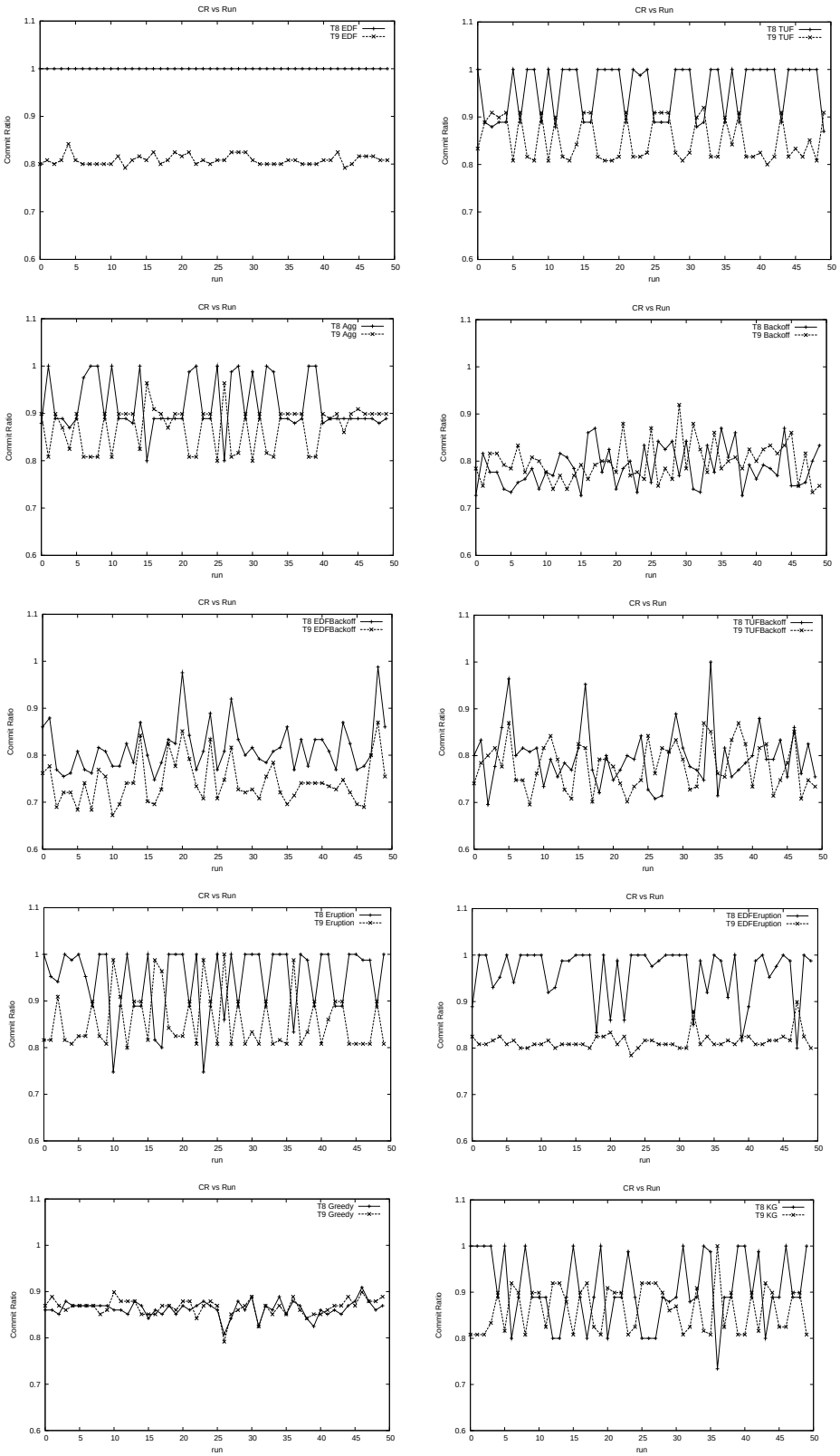

Figure 12.5: 2 threads behavior (SkipList).

Chapter 13

Conclusions and Future Work

In this dissertation, we addressed the problem of scheduling distributable threads in distributed real-time systems. Specifically, we investigated the possibility of providing timeliness assurances to distributed real-time systems that are partially synchronous and that are subject to failure.

Our research clarified a number of issues. First, it indicates that collaborative scheduling algorithms can provide better timeliness assurances than independent node scheduling algorithms for certain thread sets. This improved timeliness is achieved because all nodes in the system have full information of system state and therefore can make decisions that optimize system-wide timeliness. In contrast, the lack of global information at each node in independent node scheduling forces these nodes to make scheduling decisions using local information only, thus, possibly, jeopardizing global timeliness.

Our research indicates that this is particularly obvious for certain thread sets where the execution time ratios of the sections belonging to each thread is not one to one – i.e., the execution times of the sections of a thread are significantly different. These differences may lead each node to consider the thread of varying degrees of importance – remember that we consider the potential utility density of a thread during scheduling. For sections with small execution times, the thread will be considered important, while for sections with large execution times, the thread will be considered less important. This can lead to conflicting decisions occurring on each node. As mentioned in the previous paragraph, this leads to local minima in the decisions being taken on each node – i.e., while the decisions being made on each node may be locally optimal, they are not optimal in a system-wide sense.

Another advantage of collaborative scheduling is that it can abort a thread early if a later section of the thread is not feasible. Since collaborative scheduling, at least the variants thereof that we develop in this dissertation, considers future sections as well as current sections in its computations, it has the ability to detect earlier on if a thread is eventually going to miss its end-to-end deadline because a later section misses its deadline. This allows

the system to prune that particular thread out of the system, thus, saving the resources it would have consumed needlessly, since it was going to miss its deadline anyway, for executing other threads which actually have a chance to meet their deadlines. Independent scheduling, on the other hand, has no knowledge of what is going to happen in the future. Therefore, it keeps a thread if its current section is schedulable. Even if the future sections of this thread will not meet their deadlines, the thread continues execution oblivious of the fact that its eventual fate is to miss its end-to-end deadline. Naturally, this wastes resources as they are used to make progress on a thread that is never going to meet its deadline anyway.

However, the improved timeliness of collaborative scheduling comes at the price of higher scheduling overhead and thus can benefit only those systems that can tolerate their higher overhead. We also believe that our collaborative scheduling algorithms, since they incorporate failure detection with scheduling, allows us to more seamlessly provide performance assurances during failures.

The high overhead of collaborative scheduling becomes especially obvious when we consider distributed dependencies. We showed in this dissertation that the overhead introduced by distributed lock management and distributed deadlock detection and resolution algorithms is quite significant.

Thus, we proposed an alternative concurrency control mechanism. Specifically, we addressed the use of software transactional memory for concurrency control in distributed real-time systems. We proposed two different schedulability analysis algorithms for distributed systems programmed using software transactional memory for concurrency control. The schedulability analysis we proposed allows programmers of distributed real-time systems to add software transactional memory to their repertoire of tools.

However, the analysis we provide is a first step towards that goal. In particular, we currently do not consider cross-node transactions. Also, the analysis for the multiprocessor case hinges on having transactions of small size. While this is reasonable given the small size of critical sections in most typical embedded real-time applications, it would be interesting to see how far this assumption can be relaxed and what the resulting analysis would look like.

We also made a brief foray into the field of contention management. Specifically, we showed how we can use basic real-time information, such as deadlines and TUFs, to improve the timeliness of applications that use STM and to tighten the retry bound of transactions on systems scheduled using the GEDF discipline.

Before STM can become an integral part of distributed real-time programmers' repertoires, it is necessary that a number of research issues are addressed. Therefore, after a brief reiteration of the contributions of this dissertation in Section 13.1, we propose, in Section 13.2, a number of research directions that can further elaborate on the issue of STM in distributed systems.

13.1 Contributions

In this dissertation, we studied a number of issues. In particular, our research on collaborative scheduling resulted in the design of three different collaborative scheduling algorithms. The first of these, ACUA, is based on the distributed consensus problem. ACUA is designed to run on a partially synchronous distributed system where both message loss and communication delay are stochastically described.

Thus, our first step in designing ACUA was to determine the feasibility of implementing one of the Chandra-Toueg failure detectors on such a platform [22]. In Chapter 3, we showed how a stochastic S-class failure detector can be designed in such an environment and how this FD can be used to implement a solution to the distributed consensus problem.

We then proposed a collaborative scheduling algorithm that uses this solution to the distributed consensus problem to come to agreement on scheduling decisions. We empirically and analytically evaluated the properties of ACUA and showed that it could provide better timeliness assurances than independent node scheduling for systems that could tolerate its higher overhead.

In an attempt to lower the overhead associated with the consensus algorithm (particularly in the presence of failure), we designed a quorum-based collaborative scheduling solution, QBUA. QBUA operates in the same environment as ACUA but does not depend on a solution to the distributed consensus problem. In Chapter 4, we described this algorithm and empirically and analytically evaluated its properties. The same was performed for an extension of QBUA, DQBUA, that allowed the algorithm to handle distributed dependencies (described in Chapter 5).

At this point in our research, we identified distributed concurrency control as one of the major sources of overhead in these algorithms. Thus we turned our attention to alternatives to lock-based concurrency control solutions. In particular, we identified software transactional memory as a promising solution to this problem.

Toward that end, we designed two schedulability analysis algorithms for providing upper bounds on the response times of tasks in distributed systems programmed using STM (Chapters 10 and 11). The first of these algorithms provides such assurances for distributed systems where nodes are uniprocessors while the second is an algorithm that targets distributed systems where each node is a multiprocessor. To further our goal of making STM a part of distributed real-time systems, we identified some of the issues that need to be resolved in order to do so. Some of these issues and their proposed solutions are discussed in Chapters 1 and 9. In Section 13.2, we summarize some of these issues which we suggest as future work. We also designed real-time contention managers and showed how these impact on the performance of STM in Chapter 12.

Another contribution of this dissertation is the design and implementation of ChronOS, a real-time kernel based on Linux that includes all the scheduling algorithms considered in this

dissertation.

13.2 Future work

As previously mentioned, we only briefly touched on the topic of STM in distributed real-time systems. There are many research problems that need to be addressed before a fully fledged implementation of STM on distributed real-time systems can be implemented. Below, we enumerate some of these.

13.2.1 Designing Distributed STM Protocols

The algorithms and protocols that need to be designed depend on the programming abstraction chosen to implement STM on the target system. Some of the necessary abstractions have been touched upon in Section 9.3.1, here we elaborate on these points.

For the model where code migrates, creating cross-node transactions, and data is immobile, the main abstraction that needs to be designed is a real-time distributed commit protocol. Since cross-node transactions are permitted, with each node involved hosting part of the transaction, a distributed commit protocol is necessary to ensure atomicity. A number of distributed commit protocols have been studied in the literature, with the two phase commit protocol being the most commercially successful protocol. Unfortunately, the blocking semantics of the two phase commit protocol may not be very suitable for real-time systems. Therefore alternatives like the three phase commit protocol (despite its larger overhead) may be more appropriate due to its non-blocking semantics. Other alternatives that involve the relaxation of certain properties of distributed commit protocols in order to improve efficiency are discussed in [60]. We propose the design of distributed commit protocols whose timeliness behavior can be quantified theoretically and/or empirically, in order to allow the system to provide guarantees on end-to-end timeliness as future work. The retry costs and the feasibility of rolling back in real-time systems should also be addressed in future work.

For the approach where code is immobile and data migrates, the most important protocol that needs to be designed is a distributed real-time cache coherence protocol. This protocol needs to be location aware in order to reduce communication latency and should be designed to reduce network congestion. The cache coherence problem for multiprocessors has been extensively studied in the literature [128]. There are also some solutions for the distributed cache coherence problem (see [3, 24, 79, 132] for a, not necessarily representative, sample of research on this issue). Distributed cache coherence bears some similarity to distributed hash table (or DHT) protocols which have been an active topic of research recently due to the popularity of peer-to-peer applications. Examples of DHT algorithms that are of interest are [70, 113, 117] – none of these algorithms take into account timeliness constraints.

We propose that future work consider a cache coherence algorithm based on hierarchical clustering to reduce network traffic and path reversal to synchronize concurrent requests, an approach used in [67]. Other approaches for implementing distributed cache coherence are also candidates for future work. An important part of research in this area will be the design of cache coherence protocols that can provide timeliness guarantees that can be verified theoretically and empirically.

For the hybrid abstraction, where both code and data can move, several issues need to be determined. Among the issues that need to be resolved are the different methods of distributing transactional meta-data in order to ensure efficient execution of the STM system, providing a mechanism to support atomic commitment when code is allowed to migrate thus resulting in multi-node transactions, aggregating communication in order to reduce the effect of the extra communication necessary to manage the STM system (possibly by piggybacking this information over normal network traffic) and optimizing network communication to reduce latency. It is also necessary to design appropriate mechanisms for choosing whether data or code migration is going to occur. Currently, the choice of which part of the program to migrate is performed under programmer control [19]. We propose that future work consider the design of automated methods for deciding which part of the program moves through either compile-time analysis or at run-time.

13.2.2 Programming Language Support for Distributed STM

The programming abstraction chosen and the protocols and algorithms necessary to support them need to be incorporated into a suitable programming language. Issues that need to be addressed are extending the programming language syntax to include support for higher level abstractions built upon STM. Syntactic modifications to support the new constructs need to be developed for the target programming language. The most basic syntactic extension required is a method for demarcating atomic blocks (i.e. blocks of code that will be executed within the context of STM), additions such as programmer controlled retry and providing alternative transactional execution can also be considered.

In addition to these syntactic extensions, modifications to the run-time environment are also required. Naturally, the actual modifications made to the programming language will depend on the programming abstraction chosen. Some of the design issues involved are choosing appropriate meta-data to represent STM objects, providing appropriate mechanisms to atomically commit transactions (for example by using atomic hardware instructions such as compare-and-swap, or CAS, on suitably indirected meta-data), providing implementations for the different design choices of STM (e.g., visible reads versus invisible reads and weak versus strong atomicity). Some other extensions to the programming language/compiler used include:-

- **Compiler instrumentation for STM:** As mentioned in Chapter 1, one of the major

hurdles to mainstream acceptance of STM is the overhead associated with the unnecessary instrumentation of loads and store.

If only loads and stores within transactions are instrumented, this may lead to weak atomic semantics where non-transactional access to shared memory can lead to violation of atomicity properties. In addition, a careful analysis of the code in an application can reduce the number of loads and stores that need to be instrumented and hence reduce the overhead of STM.

Thus, what is required is some sort of static analysis at compile time that would allow us to minimize the number of loads and stores to instrument while at the same time, possibly, preventing weak atomicity by ensuring that all loads and stores that need instrumenting are in fact instrumented. This problem is akin to the problems of alias analysis and escape analysis in standard compiler theory.

From lessons learned from research on alias analysis and escape analysis, we know that this problem is quite challenging. However, we believe that most real-time code, in contrast to general purpose code, is more structured and would allow a reasonably efficient implementation of such an analysis.

We suggest the development of algorithms to automatically instrument loads and stores where necessary as future work.

- **Hybrid data/code migration**: We touched on the research issues involved in developing a hybrid data/code migration scenario for reducing the overhead of STM in Chapter 9. Here we reiterate some of the most salient points for this direction of research. This approach is touched upon in [19]. This is a hybrid approach where either data objects or code can migrate while still retaining the semantics of STM. By allowing either code or data to migrate, we can choose a migration scenario that results in the least amount of communication overhead. For example, suppose we have a simple transactional program that increments the value of a shared variable X and stores the new value in the transactional store. Assume further that X is remote, using a data flow abstraction would necessitate two communication delays; one to fetch X from its remote location and the other to send it back once it has been incremented. Using a control flow abstraction in this case may be more efficient since it will only involve a single communication delay.

  On the other hand, assume that several processes need access to a small data structure and that these processes are in roughly the same location and are far away from the data they need. Since communication delay depends on distances, it may make sense to migrate the data to the processes in this case rather than incur several long communication delays by moving the code to the data. In short, the choice of whether to migrate code or data can have a significant effect on performance. In [19], this is accomplished under programmer control by allowing an *on* construct which a programmer can use to demarcate code that should be migrated.

We propose the design of solutions that would use static analysis at compile-time (or dynamically at run-time) to make decisions about which part of the application to move using a number of heuristics such as, for example, size of code/data and locality considerations as future work.

13.2.3 Schedulability Analysis

Our current schedulability analysis (see Chapters 10 and 11) for systems where concurrency control is managed using STM has concentrated on traditional hard real-time systems. While this analysis provides a good idea about the schedulability of a system, it does not say anything about the type of assurances that can be offered during overloads.

As mentioned in Chapter 1, there are a number of emerging distributed real-time applications that operate in environments where transient or sustained overloads are possible. Thus, it becomes necessary to extend our analysis to include these cases.

Time Utility Functions, as previously stated, are ideally suited for describing the timeliness requirements in such systems since they provide us with the ability to describe an activity's urgency and importance separately. Utility Accrual schedulers allow us to take advantage of the descriptive power of TUFs and provide timeliness assurances that gracefully degrade during overloads.

Designing schedulability analysis for the various different approaches for incorporating STM into distributed systems and looking at the possibility of providing some assurances during overloads using the TUF abstraction are possible directions for future research.

Appendices

Appendix A

Selected Data Structures added to the 2.6.24 Linux Kernel

```
/*protocol datagram description*/
typedef struct list_of_dts{
  unsigned long long gtid;
  struct list_head list;
}list_of_dts;

/*description per dt*/
typedef struct dt_description{
    int node_number;  //number of nodes hosting service
    struct timespec release_time; //release time of section
    struct timespec deadline; //deadline of section
    unsigned long WCET; //execution time estimate of section
    int max_util; //Maximum utility of thread
    unsigned long sec_id;  //logical section id
    pid_t section_pid; //physical pid of section
    struct list_head list;
} dt_description;


struct mutex_data
{
        atomic_t val;
        int owner;
};

struct abort_info
```

```
{
        int aborted;
        struct timespec deadline;
        unsigned long exec_time;
        int max_util;
};


struct rt_info
{
        struct task_struct *task;
        struct timespec deadline;
        struct timespec period;
        unsigned long exec_time;
        struct timespec left;
        int rt_sched;
        int max_util;
        long invValDen;
        struct rt_info *next;
        struct rt_info *prev;
        struct rt_info *p1;
        struct rt_info *p2;
        struct rt_info *p3;
        struct rt_info *p4;
        struct rt_info *p5;
        struct rt_info *p6;
        struct mutex_data *requested_resource;
        struct rt_info *dep;
        struct abort_info abortinfo;
        int about_to_abort;
        u64 gtid;
        unsigned char flags;
};
```

Appendix B

Selected System Calls added to the 2.6.24 Linux Kernel

```
//This function converts a normal thread into a real-time thread
//with the parameters passed in.

.long sys_setrtinfo(int tid, int rt_sched, int max_util,
struct timespec __user *deadline, struct timespec *period,
unsigned long exec_time)

//Convert a real-time thread identified by the PID tid,
//to a normal thread.

.long sys_end_rt_seg(int tid)

//This function issues a request for a mutex
//It should be used to acquire locks for resources from
//within our real-time tasks.

.long sys_request_rt_resource(struct mutex_data
                                __user *mutexreq)

//This function releases a lock from within a real-time task.

.long sys_release_rt_resource(struct mutex_data
                                   __user *mutexreq)

//This function retrieves the real-time scheduling parameters
```

```
//of the task with PID tid, and places them in ``output".

.long sys_getrtinfo(int tid, struct rt_info *output)

//Add information about the DT with ID gtid, to the kernel
//level data structures. The information is contained in the
//structure, hello, passed to the function.

.long sys_dt_add_all_info(dt_gtid_t __user *gtid,
                          struct dt_description *hello)

//Remove information about DT with ID gtid from
//the kernel level data structures.

.long sys_dt_remove_all_info(dt_gtid_t __user *gtid)

//Perform the consensus round using the suggested list
//present in parameter ``hello".

.long sys_dt_consensus_results(struct list_of_dts *hello)
```

Bibliography

[1] The homepage of Interbench the Linux interactivity benchmark. `http://users.on.net/~ckolivas/interbench/`.

[2] QNX Realtime Operating System. `http://www.qnx.com/`.

[3] J. Aguilar and E. L. Leiss. A general adaptive cache coherency-replacement scheme for distributed systems. In *IICS '01: Proceedings of the International Workshop on Innovative Internet Computing Systems*, pages 116–125, London, UK, 2001. Springer-Verlag.

[4] M. K. Aguilera, G. L. Lann, and S. Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. In *DISC '02*, pages 354–370. Springer-Verlag, 2002.

[5] Y. Amir and A. Wool. Optimal availability quorum systems: Theory and practice. Technical report, Jerusalem, Israel, Israel, 1996.

[6] J. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *IEEE RTSS*, pages 92–105, 1996.

[7] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *IEEE RTSS*, pages 28–37, 1995.

[8] J. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. Lock-free transactions for real-time systems. In *Real-Time Databases: Issues and Applications*. Amsterdam: Kluwer Academic Publishers., 1997.

[9] J. H. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *J. Comput. Syst. Sci.*, 68(1):157–204, 2004.

[10] J. S. Anderson and E. D. Jensen. Distributed real-time specification for java: a status report (digest). In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 3–9, New York, NY, USA, 2006. ACM.

[11] S. ANSI/ISO/IEC-8652:1995. *Ada 95 Reference Manual*. Intermetrics, Inc., January 1995.

[12] T. ARP. PC Power Management Guide Rev. 2.0. `http://www.techarp.com/showarticle.aspx?artno=420`.

[13] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *PODC '06*, pages 308–315, New York, NY, USA, 2006. ACM.

[14] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3(1):67–99, 1991.

[15] D. Beal. Linux as a real-time operating system. `http://www.freescale.com/files/soft_dev_tools/doc/white_paper/CWLNXRTOSWP.pdf`.

[16] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *IEEE RTSS*, pages 149–160, 2007.

[17] R. Bettati and J. W. s. Liu. End-to-end scheduling to meet deadlines in distributed systems. pages 452–459, 1992.

[18] J. Bobba, R. Rajwar, and M. Hill. Transactional memory biblography. `http://www.cs.wisc.edu/trans-memory/biblio/swtm.html`.

[19] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08*, pages 247–258, 2008.

[20] G. Bracha and S. Toueg. A distributed algorithm for generalized deadlock detection. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 285–301, New York, NY, USA, 1984. ACM.

[21] J. R. Cares. *Distributed Networked Operations: The Foundations of Network Centric Warfare*. iUniverse, Inc., 2006.

[22] T. D. Chandra. Unreliable failure detectors for asynchronous distributed systems. Technical Report TR93-1377, 1993.

[23] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[24] Y. Chang and L. N. Bhuyan. An efficient tree cache coherence protocol for distributed shared memory multiprocessors. *IEEE Transactions on Computers*, 48(3):352–360, 1999.

[25] W. Chen, S. Lin, Q. Lian, and Z. Zhang. Sigma: A fault-tolerant mutual exclusion algorithm in dynamic distributed systems subject to process crashes and memory losses. In *PRDC '05*, pages 7–14, Washington, DC, USA, 2005. IEEE Computer Society.

[26] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, 2002.

[27] H. Cho, B. Ravindran, and E. D. Jensen. Space-optimal, wait-free real-time synchronization. *IEEE Transactions on Computers*, 56(3):373–384, 2007.

[28] A. N. Choudhary, W. H. Kohler, J. A. Stankovic, and D. Towsley. A modified priority based probe algorithm for distributed deadlock detection and resolution. *IEEE Trans. Softw. Eng.*, 15(1):10–17, 1989.

[29] R. Clark, E. Jensen, and F. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *1993 Winter USENIX Conf.*, pages 127–146, 1993.

[30] R. Clark, E. D. Jensen, et al. An adaptive, distributed airborne tracking system. In *IEEE WPDRTS*, volume 1586 of *LNCS*, pages 353–362. Springer-Verlag, April 1999.

[31] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, CMU, 1990. CMU-CS-90-155.

[32] R. K. Clark, E. D. Jensen, F. D. Reynolds, E. Douglas, J. Franklin, and D. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *In Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 127–146, 1993.

[33] J. Corbet. Approaches to realtime Linux. `http://lwn.net/Articles/106010/`.

[34] E. Curley, J. S. Anderson, B. Ravindran, and E. D. Jensen. Recovering from distributable thread failures with assured timeliness in real-time distributed systems. In *IEEE SRDS*, pages 267–276, 2006.

[35] P. H. Dana. Global positioning system (gps) time dissemination for real-time applications. *Real-Time Syst.*, 12(1):9–40, 1997.

[36] J. R. G. de Mendivil, A. Demaille, J. B. Auban, and J. R. Garitagoitia. Correctness of a distributed deadlock resolution algorithm for the single request model. In *PDP '95: Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing*, page 254, Washington, DC, USA, 1995. IEEE Computer Society.

[37] J. R. G. de Mendívil, n. Federico Fari J. R. Garitagoitia, C. F. Alastruey, and J. M. Bernabeu-Auban. A distributed deadlock resolution algorithm for the and model. *IEEE Trans. Parallel Distrib. Syst.*, 10(5):433–447, 1999.

[38] U. C. Devi, H. Leontyev, and J. H. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *ECRTS '06*, pages 75–84, Washington, DC, USA, 2006. IEEE Computer Society.

[39] U. Drepper. Futexes are tricky. Dec 2005.

[40] U. Drepper and I. Molnar. The Native POSIX Thread Library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`.

[41] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HOTOS '01*, pages 75–80, 2001.

[42] A. K. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Rec.*, 15(3):37–45, 1986.

[43] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[44] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Fast scheduling of distributable real-time threads with assured end-to-end timeliness. Technical report, Virginia Tech, ECE Dept., November 2007. Available at: `http://www.real-time.ece.vt.edu/RST_TR.pdf`.

[45] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Fast scheduling of distributable real-time threads with assured end-to-end timeliness. In *International Conference on Reliable Software Technologies - Ada-Europe 2008*, June 2008. To appear, available at: `http://www.real-time.ece.vt.edu/rst08.pdf`.

[46] S. F. Fahmy, B. Ravindran, and E. D. Jensen. On scalable synchronization for distributed embedded real-time systems. In *SEUS*, 2008. `http://www.real-time.ece.vt.edu/seus08.pdf`.

[47] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Scheduling dependent distributable real-time threads in dynamic networked embedded systems. In *DIPES*, September 2008. To appear, available at: `http://www.real-time.ece.vt.edu/dipes08.pdf`.

[48] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Scheduling distributable real-time threads in the presence of crash failures and message losses. In *ACM SAC*, pages 294–301, March 2008.

[49] 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 '09*, 2009.

[50] S. F. Fahmy, B. Ravindran, and E. D. Jensen. Response time analysis of software transactional memory-based distributed real-time systems. In *SAC '09*, pages 334–338, New York, NY, USA, 2009. ACM.

[51] M. J. Fischer and M. Merritt. Appraising two decades of distributed computing theory research. *Distrib. Comput.*, 16(2-3):239–247, 2003.

[52] B. Ford and J. Lepreau. Evolving mach 3.0 to a migrating thread model. In *USENIX Technical Conference*, pages 97–114, 1994.

[53] FSF. GNU General Public License. `http://www.gnu.org/licenses/gpl.html`.

[54] J. Goldberg, L. Gong, I. Greenberg, R. Clark, E. Jensen, K. Kim, and D. Wells. Adaptive fault-resistant systems, 1994.

[55] J. Goldberg, I. Greenberg, et al. Adaptive fault-resistant systems (chapter 5: Adpative distributed thread integrity). Technical Report csl-95-02, SRI International, January 1995.

[56] J. B. Goodenough and L. Sha. The Priority Ceiling Protocol: A method for minimizing the blocking of high priority ada tasks. In *IRTAW '88: Proceedings of the second international workshop on Real-time Ada issues*, pages 20–31, New York, NY, USA, 1988. ACM.

[57] D. Grossman. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 695–706, New York, NY, USA, 2007. ACM.

[58] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust contention management in software transactional memory. In *Proceedings of SCOOL*, October 2005.

[59] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC '05*, pages 258–264, 2005.

[60] R. Gupta, J. Haritsa, K. Ramamritham, and S. Seshadri. Commit processing in distributed real-time database systems. *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 220–229, 4-6 Dec 1996.

[61] W. A. Halang and M. Wannemacher. High accuracy concurrent event processing in hard real-time systems. *Real-Time Syst.*, 12(1):77–94, 1997.

[62] M. G. Harbour and J. C. Palencia. Response time analysis for tasks scheduled under edf within fixed priorities. In *IEEE RTSS*, page 200, 2003.

[63] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402. 2003.

[64] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. *icdcs*, 00:522, 2003.

[65] M. Herlihy, M. Moir, and V. Luchangco. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, oct 2006.

[66] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[67] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.

[68] J.-F. Hermant and G. L. Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, 51(8):931 – 944, August 2002.

[69] J.-F. Hermant and J. Widder. Implementing reliable distributed real-time systems with the Θ-model. In *OPODIS*, pages 334–350, 2005.

[70] K. Hildrum, R. Krauthgamer, and J. Kubiatowicz. Object location in realistic networks. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 25–35, New York, NY, USA, 2004. ACM.

[71] C. Hoare. Towards a theory of parallel programming. In C. Hoare and R. Perrott, editors, *Operating System Techniques*, pages 61–71. Academic Press, 1972.

[72] P. Holman and J. H. Anderson. Supporting lock-free synchronization in pfair-scheduled real-time systems. *J. Parallel Distrib. Comput.*, 66(1):47–67, 2006.

[73] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quaterly*, 21:177–185, 1974.

[74] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) System Interfaces, Issue 6.* 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[75] E. Jensen, C. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems, 1985. IEEE RTSS, pages 112–122, 1985.

[76] B. Kao and H. Garcia-Molina. Subtask deadline assignment for complex distributed soft real-time tasks. Technical report, Stanford, CA, USA, 1993.

[77] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1268–1274, 1997.

[78] B. Kao, H. Garcia-molina, and B. Adelberg. On building distributed soft real-time systems. In *In The Third Workshop on Parallel and Distributed Real-Time Systems*, pages 13–19, 1995.

[79] C. A. Kent. Cache coherence in distributed systems. *WRL Technical Report 87/4*, 1987.

[80] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A practitioner's handbook for real-time analysis.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[81] T. F. Knight. An architecture for mostly functional languages. In *ACM LFP*, pages 500–519, Aug 1986.

[82] B. Korenfeld and M. Medina. Transactional memory. Technical Report MIT/LCS/TM-475, University of Tel-Aviv Computer Engineering Dept., Jun 2006.

[83] N. Krivokapić, A. Kemper, and E. Gudes. Deadlock detection in distributed database systems: a new algorithm and a comparative performance analysis. *The VLDB Journal*, 8(2):79–100, 1999.

[84] A. D. Kshemkalyani and M. Singhal. A one-phase algorithm to detect distributed deadlocks in replicated databases. *IEEE Trans. on Knowl. and Data Eng.*, 11(6):880–895, 1999.

[85] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[86] P. Li. Time/utility function decomposition techniques for utility accrual scheduling algorithms in real-time distributed systems. *IEEE Trans. Comput.*, 54(9):1138–1153, 2005. Student Member-Haisang Wu and Senior Member-Binoy Ravindran and Member-E. Douglas Jensen.

[87] P. Li, B. Ravindran, H. Cho, and E. D. Jensen. Scheduling distributable real-time threads in tempus middleware. *Parallel and Distributed Systems, International Conference on*, 0:187, 2004.

[88] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE Transactions on Software Engineering*, 30(9):613–629, 2004.

[89] P. Li, H. Wu, B. Ravindran, and E. D. Jensen. A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints. *IEEE Trans. Comput.*, 55(4):454–469, 2006.

[90] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[91] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[92] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, New Jersey, 2000.

[93] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, CMU, 1986. CMU-CS-86-134.

[94] C. D. Locke. *Best-effort decision-making for real-time scheduling*. PhD thesis, Pittsburgh, PA, USA, 1986.

[95] R. Love. *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, Inc., 2007.

[96] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06*, pages 198–208. 2006.

[97] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time java. *RTSS*, 0:62–71, 2005.

[98] V. J. Marathe and M. L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, University of Rochester Computer Science Dept., Jun 2004.

[99] W. Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[100] D. P. Maynard, S. E. Shipman, et al. An example real-time command, control, and battle management application for alpha. Technical Report Archons Project 88121, CMU CS Dept., December 1988.

[101] D. Mills. Network Time Protocol (Version 3) Specification, Implementation, 1992.

[102] D. P. Mitchell and M. J. Merritt. A distributed algorithm for deadlock detection and resolution. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 282–284, New York, NY, USA, 1984. ACM.

[103] I. Molnar. CONFIG_PREEMPT_REALTIME, 'Fully Preemptible Kernel', VP-2.6.9-rc4-mm1-T4. `http://lwn.net/Articles/105948/`.

[104] A. Mostéfaoui and M. Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach. In *DISC '99*, pages 49–63, London, UK, 1999. Springer-Verlag.

[105] OMG. Real-time corba 2.0: Dynamic scheduling specification. Technical report, Object Management Group, September 2001.

[106] M. T. Ozsu. *Principles of Distributed Database Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[107] P. Pagano, P. Batra, and G. Lipari. A framework for modeling operating system mechanisms in the simulation of network protocols for real-time distributed systems. *IPDPS*, 0:160, 2007.

[108] J. Palencia and M. G. Harbour. Offset-based response time analysis of distributed systems scheduled under edf. *ECRTS*, 00:3, 2003.

[109] J. C. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. of the 19th IEEE RTSS*, pages 26–37, 1998.

[110] G. Pardo-Castellote. Omg data-distribution service: Architectural overview. *ICDCSW*, 00:200, 2003.

[111] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[112] R. Pellizzoni and G. Lipari. Improved schedulability analysis of real-time transactions with earliest deadline scheduling. *RTAS*, pages 66–75, 2005.

[113] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320, New York, NY, USA, 1997. ACM.

[114] B. Ravindran, J. S. Anderson, and E. D. Jensen. On distributed real-time scheduling in networked embedded systems in the presence of crash failures. In *IFIP SEUS Workshop*, pages 67–81, 2007.

[115] B. Ravindran, E. Curley, J. S. Anderson, and E. D. Jensen. On best-effort real-time assurances for recovering from distributable thread failures in distributed real-time systems. In *ISORC '07*, pages 344–353. IEEE Computer Society, 2007.

[116] M. Roesler and W. A. Burkhard. Resolution of deadlocks in object-oriented distributed systems. *IEEE Trans. Comput.*, 38(8):1212–1224, 1989.

[117] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[118] M. Saksena and S. Hong. An engineering approach to decomposing end-to-end delays on a distributed real-time system. *Parallel and Distributed Real-Time Systems, Workshop*, 0:244, 1996.

[119] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004. In conjunction with PODC'04.

[120] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004.

[121] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of ACM PODC*, Las Vegas, NV, Jul 2005.

[122] W. N. Scherer III and M. L. Scott. Randomization in stm contention management (poster). In *Proceedings of ACM PODC*, Las Vegas, NV, Jul 2005.

[123] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[124] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[125] C. Shih and J. A. Stankovic. Survey of deadlock detection in distributed concurrent programming environments and its application to real-time systems. Technical report, Amherst, MA, USA, 1990.

[126] M. Spuri. Analysis of deadline scheduled real-time systems. Technical report, In Rapport de Recherche RR-2772, INRIA, 1996.

[127] W. Stallings. *Operating Systems*. Macmillan, 1992.

[128] P. Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[129] B. Sterzbach. GPS-based clock synchronization in a mobile, distributed real-time system. *Real-Time Syst.*, 12(1):63–75, 1997.

[130] J. Sun. *Fixed-Priority End-To-End Scheduling in Distributed Real-Time Systems*. PhD thesis, UIUC, 1997.

[131] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[132] Y. Tamir and G. Janakiraman. Hierarchical coherency management for shared virtual memory multicomputers. *Journal of Parallel and Distributed Computing*, 15(4):408–419, 1992.

[133] The Open Group. <dlfcn.h>. `http://www.opengroup.org/onlinepubs/009695399/basedefs/dlfcn.h.html`.

[134] The Open Group. *MK7.3a Release Notes*. The Open Group Research Institute, Cambridge, Massachusetts, October 1998.

[135] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing & Microprogramming*, 50(2-3), 1994.

[136] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. 50:117–134, 1994.

[137] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 1, Berkeley, CA, USA, 1994. USENIX Association.

[138] A. Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley & Sons, 2004.

[139] Wikipedia. Software transactional memory — wikipedia, the free encyclopedia, 2008. `http://en.wikipedia.org/w/index.php?title=Software_transactional_memory&oldid=213906392`, [Online; accessed 24-May-2008].

[140] Wikipedia. Thundering herd problem — wikipedia, the free encyclopedia, 2009. [Online; accessed 20-April-2010].

[141] I. William N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05*, pages 240–248, New York, NY, USA, 2005. ACM.

[142] Y. Zhang, B. Thrall, S. Torri, C. Gill, and C. Lu. A real-time performance comparison of distributable threads and event channels. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 497–506, Washington, DC, USA, 2005. IEEE Computer Society.

[143] Y. Zhao, E. A. Lee, and J. Liu. Programming temporally integrated distributed embedded systems. Technical Report UCB/EECS-2006-82, EECS Department, University of California, Berkeley, May 2006.

[144] Y. Zhao, J. Liu, and E. A. Lee. A programming model for time-synchronized distributed real-time systems. In *RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.