LWFG: A Cache-Aware Multi-core Real-Time Scheduling Algorithm

Aaron C. Lindsay

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

Master of Science
in
Computer Science and Applications

Binoy Ravindran, Co-Chair
Dennis G. Kafura, Co-Chair
Anil Kumar S. Vullikanti

June 8, 2012
Blacksburg, Virginia

Keywords: Real-Time, Scheduling, Multiprocessors, Cache-aware, Partitioning, Linux

LWFG: A Cache-Aware Multi-core Real-Time Scheduling Algorithm

Aaron C. Lindsay

(ABSTRACT)

As the number of processing cores contained in modern processors continues to increase, cache hierarchies are becoming more complex. This added complexity has the effect of increasing the potential cost of any cache misses on such architectures. When cache misses become more costly, minimizing them becomes even more important, particularly in terms of scalability concerns.

In this thesis, we consider the problem of cache-aware real-time scheduling on multiprocessor systems. One avenue for improving real-time performance on multi-core platforms is task partitioning. Partitioning schemes statically assign tasks to cores, eliminating task migrations and reducing system overheads. Unfortunately, no current partitioning schemes explicitly consider cache effects when partitioning tasks.

We develop the LWFG (Largest Working set size First, Grouping) cache-aware partitioning algorithm, which seeks to schedule tasks which share memory with one another in such a way as to minimize the total number of cache misses. LWFG minimizes cache misses by partitioning tasks that share memory onto the same core and by distributing the system's sum working set size as evenly as possible across the available cores.

We evaluate the LWFG partitioning algorithm against several other commonly-used partitioning heuristics on a modern 48-core platform running ChronOS Linux. Our evaluation shows that in some cases, the LWFG partitioning algorithm increases execution efficiency by as much as 15% (measured by instructions per cycle) and decreases mean maximum tardiness by up to 60%.

This work is supported by US NSWC under Grant N00178-09-D-3017-0011.

Dedication

I dedicate this thesis to my late father, Russell, who never let me settle for anything less than my full potential.

Acknowledgments

While I wrote this thesis, it is ultimately the result of inspiration from, and interaction with, a large number of people. I thank my advisor, Dr. Binoy Ravindran, for his support, guidance, and patience throughout my graduate career.

I also thank my co-advisor, Dr. Dennis Kafura, and Dr. Anil Vullikanti for serving on my committee and providing valuable feedback on my research and this thesis.

It was a pleasure working with everyone in the real-time systems lab, especially Matthew Dellinger, Ben Shelton, Antonio Barbalace, and Kevin Burns. Their friendship and camaraderie has been invaluable.

Finally, I would like to thank my family and friends for being supportive in all that I do; particularly my fiancée, Elizabeth, my mother, Paula, and my brother, Christopher.

Contents

List of Figures

List of Tables

List of Acronyms

BF	Baruah and Fisher's first-fit in order of non-decreasing relative deadline order partitioning algorithm
ccNUMA	Cache-Coherent Non-Uniform Memory Architecture
C-EDF	Clustered Earliest Deadline First scheduling algorithm
CPMD	Cache-related Preemption and Migration Delay
CPU	Central Processing Unit
DSR	Deadline Satisfaction Ratio
EDF	Earliest Deadline First
FFD	First-Fit Decreasing partitioning algorithm
FIFO	First In, First Out
G-EDF	Global Earliest Deadline First scheduling algorithm
LLC	Last Level Cache (before main memory)
LWFG	Largest WSS First, Grouping partitioning algorithm
MMT	Mean Maximum Tardiness
MTT	Multi-Threaded Task
PFair	Proportionate Fair scheduling algorithm
PMU	Performance Monitoring Unit
POSIX	Portable Operating System Interface [for Unix]
RMS	Rate Monotonic Scheduling algorithm
WCET	Worst-Case Execution Time

WFD Worst-Fit Decreasing partitioning algorithm

WSS Working Set Size

Chapter 1

Introduction

Processor manufacturers are producing chips with higher and higher core counts. Multi-core processors have become the norm in enterprise servers, desktop and laptop computers, and even some smartphones. The rise of multi-core processing has been motivated by the collision of ever-increasing transistor counts with heat and power constraints [55]. AMD's latest series of "Interlagos" Opteron^{TM} processors contain up to 16 cores and can be configured in a 4-processor arrangement for a total of 64 cores [56], Intel is working on an 80-core processor [2], and Tilera has announced plans for their TILE-Gx^{TM} family of processors, which will contain up to 100 cores [60]. It is clear that core counts will continue to increase, and that adapting existing real-time systems to these many-core platforms is an important research area.

Even as core counts continue to increase, a single shared memory model has been preserved in most common architectures – in part because it is extremely convenient for both application and system-level programmers, and also because most current software is designed around this model. Unfortunately, a single memory space shared among all processors can result in contention over the memory bus if accessed directly by all processors in a system. To make matters worse, the number of instructions executed per unit of time continues to increase due to both processor pipelining and modern multiprocessing. This instruction execution rate is increasing faster than the rate at which memory can service memory requests. The current solution to both of these problems is the modern caching hierarchy. Nearly all multi-core processors manufactured today use a hierarchical cache scheme. In such a scheme, the caches closer to main memory are larger and slower, and are shared between more processors than those closer to each core. (See Figure 4.1 for an illustration of a typical hierarchical cache architecture.)

With each layer added to the memory hierarchy, the penalty of fetching memory not recently accessed by the processor requesting it grows. Cache misses have the potential to be more costly as the memory hierarchy deepens, and should be avoided if at all possible. Furthermore, as the number of cores increases and applications adapt to take advantage of higher core

counts by introducing or increasing concurrency, the possibility of cross-processor memory interference likewise increases. This interference manifests itself when multiple cores/processors access and update memory from the same region within a short period of time, causing cache-invalidations and subsequent cache misses.

1.1 Limitations of Past Work

Multiprocessor scalability concerns are not new, and have been studied for several years. In the real-time systems research community, these concerns have largely been focused in the direction of the algorithmic complexity of the scheduling algorithms and the effects this has on the overall running time of applications. To this end, much work has been put into reducing the scheduling overheads of real-time systems.

One avenue of this research aims to reduce overheads in global schedulers by identifying the bottlenecks and minimizing their effects. These efforts include reducing the time spent in critical sections protected by global locks and finding scheduling algorithms with lower algorithmic complexity (i.e. reducing "Big O"). While these enhancements have enabled real-time systems to scale to platforms with slightly higher core counts, they often do not scale to the even higher core counts of next-generation processor architectures [29].

Scalability at these levels requires a more radical change. One of these proposed changes is to divide the tasks into groups, and assign each group to a certain number of cores. This type of scheduling is often called clustered scheduling. Each time the scheduler runs in a clustered setup it must only choose between the tasks in one group, and does not have to consider those from other groups. Another added benefit is that the scheduler may have multiple instances of itself running at the same time without interfering with each other - up to one instance per group. A special case of clustered scheduling is partitioned scheduling, in which each group is only allowed to run on one core.

Clustered/partitioned scheduling is more scalable than global scheduling because of the benefits mentioned above, but also because it limits task migrations. Whenever a task migrates between two cores, that task likely suffers a number of cache misses due to the violation of the locality of reference which is assumed by the underlying hardware. By minimizing or eliminating migrations, clustering and partitioning algorithms reduce the total number of cache misses suffered by applications utilizing those scheduling schemes.

Unfortunately, partitioned scheduling does not consider tasks which share memory with one another. It only reduces cache misses as a side effect from reducing migrations, and does not consider any further cache-related effects – such as cache invalidations due to memory-sharing tasks running on different cores, or capacity-related cache misses due to many tasks with very large working set sizes being partitioned onto the same core. Ignoring cache- and memory-related effects such as these can cause many more cache misses than necessary, and hurt execution efficiency.

1.2 Research Contributions

This thesis aims to address the perceived shortfalls of current real-time scheduling algorithms in terms of cache-awareness. In summary, our major contributions are three-fold:

1. We characterize how a partitioning strategy may affect the number of application cache-misses.
2. We design and discuss a cache-aware real-time partitioning scheme that seeks to minimize such cache misses, while still satisfying hard real-time requirements.
3. We implement our partitioning scheme in ChronOS Linux and evaluate it on a modern 48-core hardware platform using memory-intensive tasks. Our evaluation shows that our partitioning scheme increases execution efficiency of real-time workloads up to 15% and decreases mean maximum tardiness up to 60% in some cases over cache-unaware static partitioning algorithms.

1.3 Scope of Thesis

Although we have attempted to make the ideas, algorithms, and results contained in this thesis as general as possible to allow them to have the maximum impact, we realistically had to limit the scope of our research due to time and resource limitations. Below we outline the ways in which we have intentionally limited the scope of our research.

First among these limitations is our restriction of the task arrival model to include only periodic or constrained sporadic tasks. We chose to scope out the aperiodic task model for several reasons: doing so simplified not only the development of the application used for our experimental evaluation, but also lowered the complexity of the interactions between tasks, allowing for easier data analysis and clearer results. Moreover, the periodic task model is that which is most commonly dealt with in real-time systems, particularly in conjunction with partitioned real-time scheduling. However, we acknowledge that many real-time systems must consider aperiodic tasks, and leave the adaptation of our work to the aperiodic task arrival model as future work.

Second, our task model, as further described in Chapter 3, disregards dependencies between tasks. That is to say, we assume all tasks are capable of executing independently of one another, and must never suspend execution, or even busy-wait, for another task. This task model allows for a simplified and more direct cause-and-effect relationship between our algorithms and their results. However, we also recognize that many (if not most) real-world real-time applications contain at least some form of task dependencies. Although our approach allows for the use of dependencies, it does not make intelligent decisions based

on tasks' dependency relationships. Dependencies should be considered before adapting the ideas discussed in this thesis to real applications.

Our research is further limited by our choice of scheduling algorithms. We have only used the earliest deadline first (EDF) algorithm when comparing cache-aware to non-cache-aware approaches in this thesis. Although our approach is at least somewhat scheduler-independent, there are almost sure to be interactions between a chosen scheduling algorithm and the rest of the real-time system.

Finally, we note that our experimental evaluation was unable to test all possible combinations of taskset parameters (period, deadline, utilization) or memory access pattern (WSS, access stride, data structure). Nevertheless, we have attempted to generate the taskset distributions used in our experimental evaluation to be fairly representative of memory-intensive real-time applications.

There are, no doubt, other restrictions placed on the scope of this thesis. The above descriptions include most of the commonly associated research avenues which might have been considered as potential areas for related research for this work. Chapter 3: Models and Assumptions further clarifies the breadth of this thesis.

1.4 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 reviews the previous work done in the cache-aware and real-time scheduling space. Chapter 3 describes the models used and assumptions made regarding the types of real-time systems studied in this thesis. In Chapter 4, we discuss the concept of cache-awareness, and particularly how it applies to scheduling algorithms. Chapter 5 introduces the idea of cache-aware partitioning as well as our LWFG partitioning algorithm. An experimental evaluation of the LWFG algorithm is presented in Chapter 6. Finally, Chapter 7 contains our conclusions and Chapter 8 explores the areas we see as possible avenues for future work.

Chapter 2

Related Work

It has been obvious for several years that multi-core processors are on their way to prominence in most, if not all, computing environments. A corollary of this observation is that complex hierarchical caches and memory interconnects will become a reality as well. Because these changes are very real, many researchers have studied the effects of caches and memory organization on scheduling. However, because this field as a whole is still relatively new, there exist areas which have not been adequately researched.

We present a summary of the current state of the field of cache-aware real-time scheduling. First, we present a review of current practices in embedded systems and hardware-based cache partitioning in Section 2.1. A summary of non-real-time cache-aware scheduling algorithms and techniques follows in Section 2.2. Finally, Section 2.3 describes the state of the existing cache-aware real-time scheduling algorithms and Section 2.4 summarizes the related work.

Figure 2.1 shows the relationship between these areas of research. Although the embedded systems and cache partitioning work is relatively distinct from the focus of this thesis, we include it here because many of the underlying concepts are similar. The area of non-

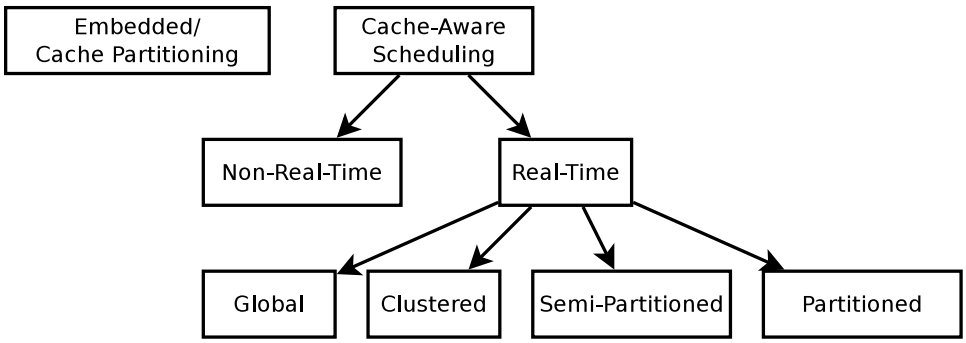


Figure 2.1: Relationship of Related Cache-Aware Research

real-time cache-aware scheduling is more closely related. This work takes many forms, but because the execution requirements are looser than they are for real-time systems, it is difficult to classify this work using the same taxonomy as we use for the related real-time research.

We classify the cache-aware real-time work based on allowed task migrations. Global scheduling algorithms allow any task to migrate to any core, while partitioned scheduling only allows a task to run on its one specified core. Clustered and semi-partitioned scheduling lie somewhere in between. Clustered (or hybrid, as it is sometimes called) scheduling typically allows tasks to migrate between some subset of the available processors, while semi-partitioned scheduling is essentially partitioned scheduling, but some tasks are allowed to migrate only under certain conditions.

2.1 Embedded Systems and Cache Partitioning

In 1997, Luculli and Natale published a paper about cache-aware real-time scheduling for embedded systems [47]. Their focus was on static real-time systems, where the task arrival times are known in advance. Within this realm, they showed that it was possible to reduce the pessimism of the worst-case execution time (WCET) estimations for static task sets by analyzing the interference patterns of the tasks based on preemptions and control flow within a task. Notably, their work only focused on instruction caches, which were deemed more important at the time, and omitted data caches entirely. Their work also does not apply to many current real-time systems used in the enterprise, which instead use sporadic task arrival models, and therefore cannot compute any of this information ahead of time.

There has also been some research completed which attempts to partition, or even reconfigure, caches and cache hierarchies with the help of hardware in embedded systems. The general idea behind research in this area is to divide the cache among the real-time tasks to eliminate cache misses caused by contention. The first of these techniques, SMART partitioning [43, 44, 45], partitions the cache and then divides these partitions among tasks such that the most time-critical tasks receive the most reserved partitions. Less critical tasks may be relegated to a shared pool of cache partitions if they are not important enough (as defined by the application). These cache partitions are governed by OS-level software, but would require non-standard custom hardware to implement the partitioning.

Another approach to cache partitioning is presented in [35]. The authors assume a mechanism for partitioning the cache is available (and claim this is reasonable due to other work on OS kernel-based cache partitioning) and have designed a simple fixed priority scheduling algorithm which schedules tasks in FIFO order as long as enough cache partitions are available.

More recent work in the embedded real-time community includes dynamic cache reconfiguration [65, 64]. The primary goal of this work is to reduce power consumption by reconfiguring

the cache "ways" in a scheduler-aware manner so that deadlines are still met while using as little cache as possible.

There have also been efforts to extend cache partitioning to more generic multi-purpose processors [24]. The authors of this work designed and simulated a system which attempts to improve the predictability of L2 cache accesses in a simple tile (i.e. on-chip mesh network) architecture. They do so by defining 'cache bins' and assigning these bins to processes so that each process effectively has its own private cache.

Unfortunately, these cache partitioning and reconfiguration approaches are not practical for many modern real-time applications. Real-time applications often rely upon a specific architecture or platform and are not easily portable to new reconfigurable hardware. Additionally, this hardware is likely not cost-effective.

2.2 Non-real-time Cache-aware Scheduling

There have been efforts in recent years to investigate how to improve the cache-related performance of non-real-time schedulers. In [69], the authors claim that the most challenging part of this process is identifying which resources processes are contending over, and why this contention is occurring. Their work introduces a task classification scheme based on the "pain" of co-scheduling any two tasks – the harm they would cause each other if they shared a cache. The pain of any combination is calculated using their stack distance profiles. They also developed a classification scheme based on miss rates, and claim that this scheme is nearly as good, and much simpler to implement in practice. Two scheduling policies were built around this cache miss rate classification scheme, and they were shown to significantly decrease task completion times relative to the standard Linux scheduler. There are several other papers with similar work, including [33, 59].

In [63], the authors study how best to measure tasks' cache contention and sensitivity. They show that using raw PMU (performance monitoring unit) cache miss rates to estimate the contention and sensitivity of a task (as defined by [69]) does not always give an accurate model of the system's contention. Furthermore, they show that by devising some simple formulas based upon PMU data, estimates on contention can be improved. The same research group has also developed a runtime for directly measuring cache contention based on IPC, and have shown improvement in task running times for a cache-conscious scheduling approach based on their runtime environment [50].

Some take a more theoretical approach, such as [37]. They examine the theory behind "co-scheduling", the practice of scheduling tasks which have the least cache interference with each other at the same time. They find that optimal co-scheduling is NP-complete, but do provide a polynomial-time algorithm for computing an optimal co-schedule when there are two cache-sharing cores, and another near-optimal heuristic for more complex cases. They also show that their algorithm is reasonably efficient in practice, and effectively co-schedules

tasks to minimize contention. They build on their work in [38], introducing and studying several new heuristic algorithms to co-schedule tasks with polynomial complexity.

NUMA systems add an additional level of complexity to cache-aware scheduling. There have been several studies which attempt to classify the performance of cache-coherent NUMA architectures, including [62]. The interaction of scheduling and memory allocation on (cc)NUMA architectures is one area of study. It has been shown that paying attention to where tasks are executed can improve performance, but attempting to place all tasks as close as possible to their local memory does not always result in the best performance [48, 49]. Additionally, there has been an extension to an earlier cache-aware scheduler [69], which shows significant performance improvements on memory-intensive SPEC benchmarks when they slightly modify their previous algorithm [16].

In response to similar concerns about NUMA-aware scheduling, several Linux kernel developers created a CPU topology-aware scheduling algorithm [17]. This work introduces the concept of a system of hierarchical scheduling domains (`struct sched_domain`) which approximately mirror the memory access topology. Within these domains, tasks such as workload balancing are most likely to happen among leaves – tasks are more likely to be traded between processors on the same NUMA node. Using this strategy, the average memory access latency is effectively decreased.

2.3 Cache-aware Real-time Scheduling

Research on cache-aware scheduling is not restricted to non-real-time systems. Some work has investigated how to measure cache-related preemption and migration delays, and the effects these delays may have on real-time task schedulability. In [61], the authors attempt to more accurately classify the cache-related delay caused by task preemptions and develop a polynomial-time algorithm to more precisely estimate such delays for fixed priority periodic tasks. More recently, the authors of [15] present two methods to measure CPMD in real systems, and show that (in their system at least) preemptions and migrations have approximately the same cache-related delay for both preemptions and migrations. Finally, the authors of [58] seek to incorporate the interaction between processor pipelines and cache behavior (namely that pipeline stalls and prefetching can hide cache misses) into WCET estimates. They attempt to narrow the gap between pessimistic estimates and the true WCET of tasks. Such work is very important to understand the bounds necessary to guarantee deadline satisfaction on real hardware, but does not directly address improving performance or decreasing task execution times.

2.3.1 Global Scheduling Algorithms

Others have attempted to develop global cache-aware real-time scheduling algorithms. In this context, 'global' means that all active tasks in the system are considered at each scheduling event. One strategy is to use non-work-conserving scheduling (i.e. intentionally leaving some cores idle) to improve priority enforcement on priority-based scheduling algorithms [57]. In this work, they measure task performance with hardware performance counters. Whenever a high-priority task's performance is being negatively impacted by a lower-priority task, the lower-priority task can be temporarily disabled to ensure that the highest-priority task completes as soon as possible. As explained in the paper, the exact thresholds for disabling and re-enabling tasks can be configurable, and may be application-specific.

In [7], the authors approach the cache-aware scheduling problem by grouping together tasks which are likely to cause L2 cache thrashing into 'megatasks'. Cache thrashing is then discouraged by only allocating a certain number of cores to tasks in this 'megatask' such that the maximum combined WSS of all executing tasks is bounded at a value less than the capacity of the shared cache.

The authors of [20] explore five scheduling policies and their abilities to reduce cache misses for several synthetic task sets as well as in the kernel of a video encoding application in a simulator. Following this paper, they picked the best performing heuristic, and implemented it in a real system [21]. Their chosen implementation requires a cache profiler to estimate the WSS of each multi-threaded task (MTT) in the system. Their scheduler then schedules those MTTs with the lowest estimated WSS that will 'fit' in the shared cache without causing cache thrashing. They ensure some semblance of bounded tardiness by immediately scheduling any tasks which have become tardy, regardless of whether they would cause thrashing.

2.3.2 Partitioned, Semi-Partitioned, and Clustered Scheduling Algorithms

Because global scheduling is often deemed unfit for multi-core architectures due to scalability concerns, many variants of partitioned algorithms have arisen in recent years. By constraining certain tasks to run on a single core or set of cores, this class of algorithms limits or eliminates the number of migrations, migration distance (i.e. they can only migrate to cores which share an L3 cache), or both. In general, the less migrations a scheduling scheme produces, the less cache misses the system as a whole will suffer.

There has been a fair amount of work focused on solving the problem of how to best partition tasks; this is essentially a version of the bin-packing problem. Some have sought to solve this problem using polynomial-time approximation schemes (PTAS). There are many variations to these heuristics, some more simple than others; [12, 11] use a first-fit heuristic, considering the tasks in order of non-decreasing relative deadlines, others have taken the

dynamic programming approach [10], while still others have used a variable-accuracy lookup table to approximate the optimal solutions [22]. There has even been some work to study the comparable energy usage due to different partitioning schemes [8].

Unfortunately, traditional partitioning approaches suffer from the fact that tasksets with large tasks (in terms of utilization) often cannot be evenly partitioned onto m processors. For instance, three tasks each with a utilization of 0.6 clearly cannot be partitioned onto two processors without running one of the tasks on more than one processor (even though they have a total utilization of less than 2). The solution to this problem is somewhat obvious: allow a few tasks to run on more than one processor. This approach is termed semi-partitioned scheduling. There are both partitioned EDF variations of task-splitting algorithms (as they are also called) [41, 18, 42, 68], as well as RMS variations [46, 40]. In these papers, we see a few more partitioning heuristics: first-fit, considering tasks in order of non-decreasing period [40], or placing 'heavy' tasks (utilization $> 65\%$) onto dedicated processors first, then first-fit in order of non-decreasing period [42]. The authors in [14] conducted a survey of three existing semi-partitioned schedulers, finding them helpful in solving the problems with existing global schedulers as long as a few considerations are made.

Half-way between partitioned scheduling and global scheduling exists the concept of clustered scheduling, in which tasks are constrained to groups of processors (clusters) instead of to a specific processor. Partitioned and global scheduling can be thought of as special cases of clustered scheduling with cluster sizes of 1 and m, respectively. There are several variants of clustered scheduling, though the most common is clustered EDF [19]. There has also been more recent work with cache-conscious clustered scheduling [66], though this paper presents only a small simulated test.

The authors of [13] present an empirical comparison of partitioned, clustered, and global scheduling. They found that GEDF was never superior for the studied hard real-time tasksets, and that partitioned and clustered scheduling performed much better on platforms with high core counts.

2.4 Summary

In global scheduling (both real-time and non-real-time), the most oft-used techniques are encouraging or discouraging the co-scheduling of certain tasks based on their memory-usage characteristics, and attempting to minimize migrations and migration distance. These techniques have been shown to be practical and useful by some amount of research.

A 'stronger' approach to cache-awareness is to partition resources in order to minimize tasks' contention over them. One way this is done is to partition the cache/memory space itself. This technique typically involves hardware support, and at the very least presents an extremely restrictive programming environment that is not conducive to many existing real-

time applications and is more difficult of an environment in which to write new applications.

Another partitioning approach is to partition the available processors, thereby only allowing a task to run on a subset of the available cores (usually only one). Task partitioning is usually seen as a way to simplify the global scheduling problem in real-time systems. However, there is no work we are aware of which explicitly studies the algorithms used to partition tasks in a cache-aware manner.

Chapter 3

Models and Assumptions

In this chapter, we present the underlying assumptions and models we use to describe real-time systems in this thesis. We describe the properties of our real-time tasks, how they may interact with each other, and what happens when real-time constraints are violated. These models and assumptions aim to provide a common base for the rest of the discussion contained in this thesis.

3.1 Task Model

We assume there are one or more tasks present in a real-time system. Each of these tasks contains a real-time segment, which may execute one or more times over the lifetime of a task. We call each of these execution instances a 'job' (or sometimes a 'phase') of that task. For our experimentation, we assume a model where each task is implemented as one thread or process, in OS terms. In this model, a task typically executes the current job to completion, then waits for whatever condition triggers the execution of that task's next job. This condition may be the arrival of an interrupt indicating data is available for processing, or simply the expiration of a hardware timer.

3.1.1 Arrival Model

We assume a sporadic task arrival model where each task τ_i has a a minimum job-arrival separation p_i. This is called the constrained sporadic task model. No phase of a task may arrive before p_i time has elapsed after the arrival of the previous phase of that task. The job-arrival separation is commonly referred to as the period (hence the notation). We do not consider aperiodic tasks in this thesis.

3.1.2 Timeliness Model

We assume that each task τ_i has a worst-case execution time e_i, and a deadline d_i. If a phase of a task has not completed its execution d_i units of time after its arrival, it is said to be tardy. Using these parameters, we define a density for task τ_i, $\lambda_i = e_i/min(p_i, d_i)$. For the case where deadlines are less than or equal to periods (i.e. *constrained* sporadic task systems), the density for each task is equal to its utilization, $u_i = e_i/p_i$. This is equivalent to the task model presented by Baruah and Fisher in [12].

3.1.3 Abort Model

Under some circumstances, tasks may miss their deadlines. It is possible that tasks which miss their deadlines are no longer sufficiently valuable for the system to continue their execution. When this occurs, the system may decide to abort the task, effectively stopping its execution.

On our platform, we have implemented a shared memory abort mechanism. Each real-time task in the system is assigned a region of memory to watch. When this region of memory is set to a specific value, the task knows that it has been chosen to be aborted by the kernel, and should exit as soon as it is possible to do safely.

This shared memory space is created by a special character device created as a Linux kernel module. The memory is then is mapped into the memory space of the userspace task using the Linux `mmap()` system call. All of this functionality is provided via our userspace ChronOS library, `libchronos`.

3.1.4 Execution Time Calculation

We state above that we assume each task to have a worst-case execution time, e_i. This is only partially true. Though we call this term worst-case, calculating true WCETs is difficult for modern multi-core hardware [27], largely due to the presence of shared caches, as they prevent task execution behaviors from being analyzed in isolation [36]. While WCET research for multi-cores is actively progressing [34], for many current non safety-critical real-time systems, a tight, measurement-based execution time estimate is often sufficient [67, 32, 25]. Such an approach is cost-effective, particularly due to the current lack of WCET tools and techniques for determining WCETs for arbitrary combinations of processors and applications.

The above approach is the one taken in this thesis. We experimentally establish best-effort worst-case execution times when needed. The only exception to this rule is when we consider soft real-time environments or instances when execution estimates are incorrect. In these cases, we explicitly state as much.

3.1.5 Dependency Model

In real-time systems, tasks may have interdependencies. That is, it may be necessary for one task to execute before or after another task. These dependencies may be the result of a shared resource, or simply a happens-before relationship. In our experimental setup, any type of dependencies are allowed in userspace.

However, in addition to userspace-only dependency tracking, our experimental platform (ChronOS Linux) allows for kernel-level schedulers to be made aware of single-holder locks. Such locks must be created and managed through syscalls provided by ChronOS . This capability allows for kernel-level schedulers to be made aware of task dependencies and for the scheduler to create a more sophisticated schedule based on this information.

Because dependencies can affect application and real-time performance in potentially unexpected ways, we exclude them from the scope of this thesis.

3.1.6 Memory Usage Model

We assume that the working set size (WSS) of each task is known to the system. This assumption is premised upon the fact that analyzing an application's memory usage is at least as easy and accurate as calculating its WCET. In fact, theoretical WCET analysis presupposes accurate knowledge of the size of the memory used as well as its access pattern. We refer to the maximum WSS for a task τ_i as w_i. Our partitioning scheme relies upon the fact that a significant portion of a task's working set remains the same over a period of time larger than the frequency of scheduling events and context switches. Exactly how long the WSS must remain the same for our partitioning algorithm to have a positive effect is one avenue for future research.

In addition to knowledge of the memory usage footprint of each task in isolation, we rely on knowing which tasks share memory and how much memory is shared between them. For the sake of simplicity, we assume tasks which share memory share their entire WSS. However, we believe the principles discussed herein would apply similarly (though possibly to a lesser degree) to tasks which did not share all of their working set with each other.

3.2 Operating System Platform

We have chosen to base our work on a real-time Linux platform. There are a myriad of approaches which fall under this category [39], but a full discussion of their relative merits is outside the scope of this thesis. For simplicity and ease of development we have chosen to use ChronOS Linux [28], which is based on the `PREEMPT_RT` patch [52] to the Linux kernel.

3.2.1 PREEMPT_RT Patch

The PREEMPT_RT patch[1] for the Linux kernel [4] is the result of efforts by many top kernel developers to modify the kernel to better support real-time tasks. The main goal of the PREEMPT_RT patch is to decrease interrupt service latencies, which are a (if not *the*) primary concern for real-time systems. First of all, this patch converts all in-kernel spinlocks to mutexes, making the kernel more preemptible in general [53, 52]. Those spinlocks which must remain spinlocks (because their execution cannot be preempted) are converted to raw_spinlock_t locks. The patch also creates threaded IRQ handlers. The majority of the IRQ handling code is therefore entirely preemptible by any other higher-priority real-time task in the system, and that IRQ handling can be pinned to a specific CPU so it doesn't interfere with real-time tasks on other processors [1]. The PREEMPT_RT patch has also introduced priority inversion for in-kernel locks, and created a high-resolution timer infrastructure. Over time, many of the changes first introduced in the PREEMPT_RT patchset are making their way into the mainline kernel.

While not providing the absolute strictest hard real-time performance in terms of bounded interrupt latencies, the PREEMPT_RT patch provides 'good enough' performance for many applications [52]. The level of determinism provided has been studied by the Open Source Automation Development Lab (OSADL), and the PREEMPT_RT patch has been shown to provide sub-200μs latencies for a variety of modern processors from various architectures (x86, MIPS, ARM) [31]. There are countless Linux-based operating systems and enterprise installations based off of the PREEMPT_RT patch, including systems as critical as stock exchanges [6].

3.2.2 ChronOS Linux

ChronOS Linux implements an extensible real-time scheduling framework for the Linux kernel [28, 29]. Because ChronOS is based on the PREEMPT_RT patch for the Linux kernel, it enjoys the decreased latencies and increased determinism it provides. The ChronOS scheduling framework provides a base upon which global, clustered, and partitioned scheduling algorithms can be developed relatively easily[2].

ChronOS provides system calls, and a userspace library to simplify their use, which allow userspace programs to easily set the scheduler used for a certain 'scheduling domain' (set of processing cores), and enter and exit a real-time segment. The system calls which begin and end real-time segments also provide the necessary task parameters to the scheduler being used (period, deadline, WCET, etc.).

We chose ChronOS for the implementation and experimentation contained in this thesis

[1] So named for the kernel config option's historical name, CONFIG_PREEMPT_RT.

[2] Being in the kernel, coding can still be frustrating and hard to debug, but the complexities of creating a scheduler are greatly diminished compared to traditional Linux scheduler development.

because it is a stable platform providing sufficiently low kernel latencies, upon which it is easy to develop and test custom real-time scheduling algorithms.

Chapter 4

Cache Awareness

"Cache-awareness" is a term which frequently means different things in different contexts. The goal of this chapter is to provide a common ground for what we understand cache-awareness to be. In it we provide a summary of various types of cache/memory architectures, and clearly define the problem of cache-awareness and how it presents itself in real-time systems.

4.1 Current Architectures

4.1.1 Hierarchical Memory Architectures

Hierarchical caches are the most common type of memory architecture today. In a hierarchical cache architecture, there are several layers of cache between the processing cores and the shared physical memory. As you move from the cache nearest the processing core to that nearest main memory, the caches typically become larger, slower, and are shared between more cores. Figure 4.1 shows a typical example of a hierarchical cache organization. In this illustration, each core has a private L1 (data and instruction) and L2, and the pair of cores in each physical processor shares an L3 cache.

This type of cache hierarchy has been motivated by the growing gap between processor and memory speeds. By exploiting locality of reference, caches attempt to hide the high cost of memory accesses by keeping copies of part of main memory around which are likely to be used again (based on some architecture-defined heuristic). Caches most commonly exploit temporal locality, but with prefetching commonplace and cache-line sizes frequently larger than the amount of data being immediately requested, spatial locality is also used. I omit a more in-depth discussion of hierarchical caches, because this topic has been covered by countless others, including Ulrich Drepper's famous paper [30].

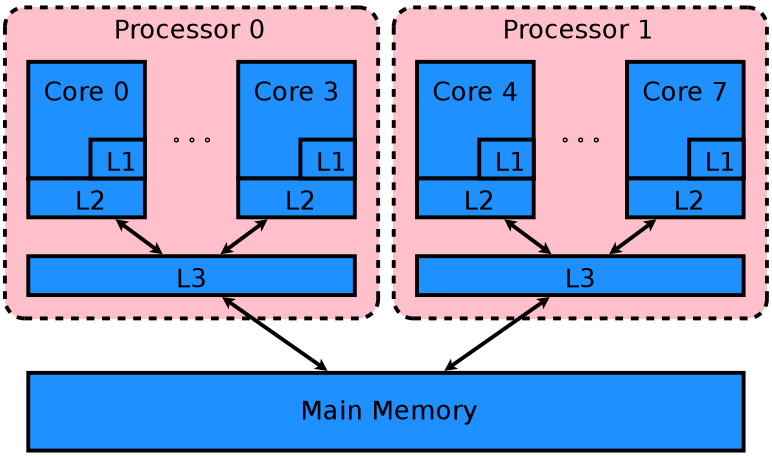


Figure 4.1: Traditional hierarchical cache/memory architecture featuring per-processor L1 and L2 caches, one L3 cache shared per packages, and a shared main memory.

Table 4.1: Memory Access Latencies on Intel® Core™ 2 Quad

Level	Size	Latency
L1	32k	1.26 ns
L2	4096k	5.86 ns
Main Memory	N/A	70.33 ns

Table 4.2: Memory Access Latencies on AMD Phenom™ II X6 1090T

Level	Size	Latency
L1	64k	0.94 ns
L2	512k	4.80 ns
L3	6144k	17.11 ns
Main Memory	N/A	52.73 ns

Table 4.3: Memory Access Latencies on Intel® Xeon® E5520

Level	Size	Latency
L1	32k	1.58 ns
L2	256k	4.00 ns
L3	8192k	15.44 ns
Main Memory	N/A	67.62 ns

Table 4.4: Memory Access Latencies on AMD Opteron™ 6164 HE

Level	Size	Latency
L1	64k	1.77 ns
L2	512k	could not measure
L3	5118k	could not measure
Main Memory	N/A	63.51 ns

In a hierarchical memory architecture, it is common for a memory operation which must go all the way main memory to take 50 or 100 times as long as a memory operation which was serviced from the L1 cache. Using the LMbench tool [54], we measured the cache and memory latencies of several modern processors with hierarchical caches. Those results are presented in tables 4.1 through 4.4. This discrepancy between L1 and main memory access latencies shows the importance of minimizing the number of trips to main memory a program must make.

Some architectures (for instance AMD's Opteron processors) support cache-coherent non-uniform memory access, or ccNUMA. ccNUMA adds additional complexity to the memory hierarchy. Main memory is divided up into 'nodes', where a node consists of a group of processing cores and a section of main memory owned by that group of cores. Any set of cores in a particular node then have fastest access to memory owned by them. Memory accesses from a core to memory not owned by that core's node therefore takes longer. How much longer is architecture-dependent and may differ depending on which remote node the memory belongs to. ccNUMA is therefore another important factor in the memory hierarchy which, if ignored, can cause significantly inefficient execution.

4.1.2 Mesh Network Architectures

There is another type of architecture currently being pioneered by Tilera, aptly named the Tile architecture. Rather than organizing caches strictly hierarchically, the Tile architecture (of which there currently exist several variants, TILE64™, TILE*Pro*64™, and TILE-Gx™) organizes the cores in a two-dimensional grid or "mesh network". In this arrangement, each "tile" (core) consists of the execution logic, register files, local caches, and a networking switch which it uses to communicate with other tiles and the rest of the hardware. This communication consists of everything necessary to communicate between cores and the rest of the machine: memory traffic, cache coherency, interrupts, etc.

Figures 4.2 and 4.3 show an overview of the mesh architecture and the detail of an individual tile, respectively. From these diagrams, we see there are only 12 tiles out of 64 which are directly connected to a memory controller. Therefore, any memory requests generated by tiles

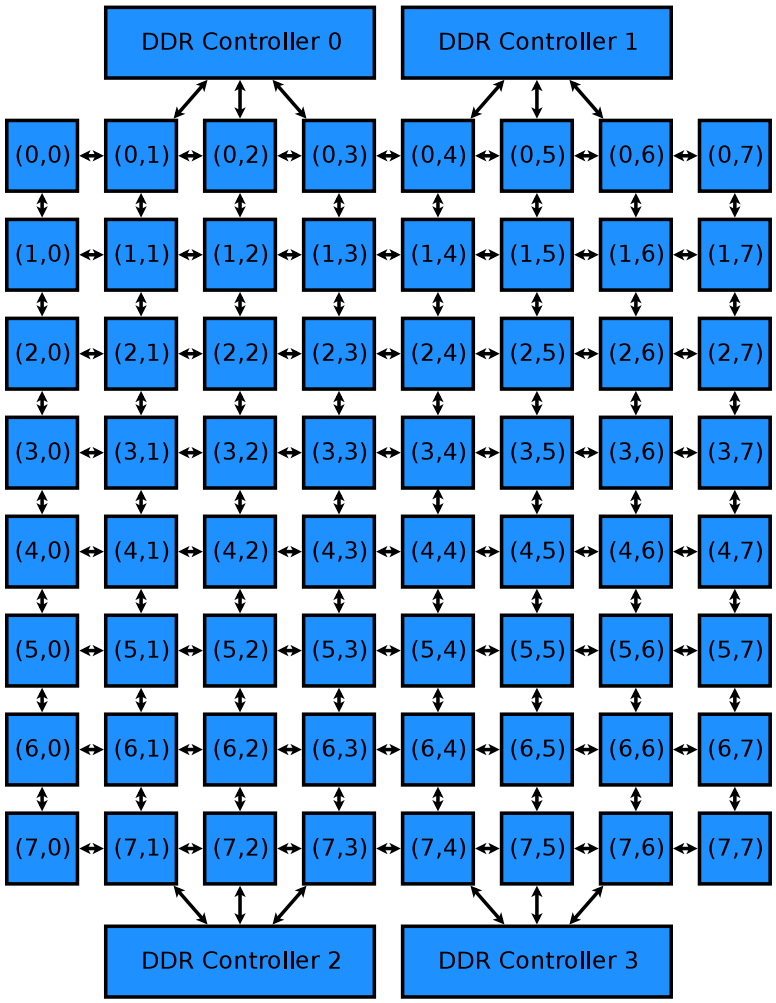


Figure 4.2: Example mesh network cache and memory structure from TILE*Pro*64™ architecture from Tilera [5].

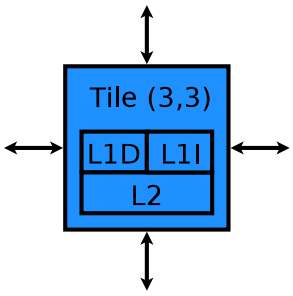


Figure 4.3: Detail of single tile from TILE*Pro*™ architecture [5].

Table 4.5: Published Memory Access Latencies on Tile*Pro*64™ architecture [5].

Operation	Latency (1 cycle → 1.33ns)
L1 hit	2 cycles
L1 miss, L2 hit	8 cycles
L1/L2 Miss, adjacent DDC™ hit	35 cycles
L1/L2 Miss, DDR2 page open, typical	69 cycles
L1/L2 Miss, DDR2 page miss, typical	88 cycles

not directly connected must be passed over the inter-tile network. Furthermore, although there does not appear to be any cache shared by more than one core, this does not mean that an L2 cache miss from a tile must always query main memory. The Tile*Pro*™ architecture contains a feature called Dynamic Distributed Cache (or DDC™). With DDC™, tiles are able to request memory in each others' caches via the mesh network to avoid being forced to fetch it from main memory. A summary of the related memory access latencies is presented in Table 4.5.

The tile architecture is unique, and is an attempt at mitigating the bottlenecks that occur with most existing shared-memory platforms. Time will tell whether this architecture is sufficient to overcome existing problems, but unfortunately it does not appear that it will gain significant traction in enterprise soon. In part because it is not a popular architecture, and in part because we were unable to run tests on a tile architecture because it is prohibitively expensive, we exclude study of this exciting new architecture from this thesis.

4.1.3 Cache Architecture Transparency

Regardless of the architecture, caches are generally implemented in hardware in a way which is transparent to the application programmer. This is dangerous. If programmers do not design their code with the memory hierarchy in mind, it can cause their programs' performance to suffer drastically. This is also true of system designers, who are responsible for creating algorithms which allocate resources such as memory and processor time to processes and who have the power to significantly hurt system performance if they are not careful. It is quite easy to implement an algorithm which violates the assumptions the hardware designers have made about the temporal or spatial locality of a memory region.

4.2 Applicability to Real-time Systems

Hard real-time systems provide worst-case execution times (WCETs) to the scheduling mechanism to ensure deadlines are met. In order for a task's WCET to be truly worst-case, the memory hierarchy must be taken into account when calculating it. Many traditional real-time schedulers ignore cache effects once they have been accounted for when calculating the WCET of a task, because ignoring them cannot, by definition, cause a hard real-time system to miss deadlines. Unfortunately, calculating true WCETs is often prohibitively expensive and difficult for modern multi-core platforms because of their complex cache hierarchies. Instead, estimated WCETs may be obtained via an experimental evaluation of the real-time application in question. In these situations, decreasing cache misses can help minimize the potential under- or over-estimation of the execution time with respect to the actual execution time, thereby improving task timeliness behavior (e.g., improved instructions per cycle, reduced tardiness).

Moreover, hard real-time systems may benefit from reduced cache misses if they are able to take advantage of the extra slack in the schedule to reduce the system's power consumption with dynamic voltage and frequency scaling, or DVFS [51].

On the other hand, soft real-time systems may encounter overload conditions (i.e. be presented more tasks than can be feasibly scheduled), and have the potential to benefit greatly from reduced cache misses, which can translate into smaller execution times and a higher number of satisfied deadlines. Because of their clear applicability to real-time systems running on real hardware, cache effects cannot be ignored when designing real-time scheduling algorithms.

4.3 Influencing Cache Behavior

The question remains: *What should we be aiming for when designing so-called 'cache-aware' algorithms?*

The obvious primary goal when designing cache-aware algorithms in general is to minimize cache misses and thereby increase execution efficiency. The particulars of this generic goal depend upon the level at which cache-awareness is being considered, but the strategy is almost always to increase locality of reference in one way or another. For example, when looking at cache-awareness in the scope of application design, changing the 2-D array access order can have a significant difference on the number of cache misses. Examples of cache-unaware and -aware 2-D array accesses are shown in figures 4.4 and 4.5, respectively.

Though this is a simplistic example, it is somewhat obvious that accessing array elements in the order $a[0][0]$, $a[0][1]$, $a[0][2]$, ..., $a[1][0]$, etc. is going to be more efficient than $a[0][0]$, $a[1][0]$, ..., $a[0][1]$, $a[1][1]$, etc. if the memory for the second dimension of the array is

```
int example_array[m][n];

for (j = 0; j < n; j++)
        for (i = 0; i < m; i++)
                example_array[i][j]++;
```

Figure 4.4: Example cache-unaware array access ordering.

```
int example_array[m][n];

for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
                example_array[i][j]++;
```

Figure 4.5: Example cache-aware array access ordering.

organized contiguously (the case on most x86 machines today). There are many domain- and application-specific methods for decreasing negative cache-effects for particular applications.

However, our focus on cache-awareness is more at the system level and does not deal directly with application-level algorithms. Our approach assumes that the memory-related characteristics of an application, such as the working set size or memory access pattern, are fixed and unchangeable by the system designer. Except for extremely application-specific or custom-designed operating systems, this is the model used: a system designer is tasked with creating a general-purpose operating system which fits the needs of many users, organizations, and applications. In this environment there are several mechanisms which can be used to influence cache performance. The two main classes of mechanisms are scheduling (where/when tasks will execute?) and controlling memory allocation (where will requested memory be allocated? On a specific ccNUMA node?).

This thesis focuses largely on cache-awareness from the angle of task scheduling. Many other avenues for improving cache miss rates exist, but we believe task scheduling to be the most promising of those areas in terms of being able to improve performance the most beyond existing approaches.

4.4 Optimal Cache Behavior

Cache Distance. Before continuing further, let us define the *cache distance* between two cores (A and B) as the best-case memory access latency from B to a word of memory that resides (and has not been invalidated) in A's L1 cache. In essence, the more (and faster) levels of the memory hierarchy that A and B share, the smaller the cache distance between

them (though in a mesh network architecture, this would be better approximated by the on-chip network latency between the two).

4.4.1 Goals

Using this definition, we can formulate a few main goals for designing a cache-aware scheduling algorithm. We note that these goals sometimes overlap, and that they can potentially compete with one another in some situations. Nevertheless, the goals are:

1. Minimize preemptions and migrations.
2. If migrations are necessary, minimize the sum cache distance between the cores tasks are migrated to and from.
3. Execute memory-sharing tasks on either the same core or cores with the lowest possible cache distance from each other.
4. Execute non-memory-sharing tasks on cores which share as little of the memory hierarchy as possible. In other words, spread out the memory pressure among the available cores so that cache invalidations are not increased due to unbalanced WSS distributions.

Let us now explore how these goals seek to improve the cache-awareness of a particular scheduling algorithm or strategy. Minimizing preemptions and migrations will increase the cache affinity of the tasks affected. If a task is preempted, it is possible (and extremely probable) that at least some of that task's working set size will be evicted from one or more caches as a result. Migrations must be minimized, because on current commodity hardware, a migration means that the task will be running on an execution core which does not share at least one cache level with its previous core. For these reasons, minimizing preemptions and migrations will reduce cache misses and increase execution efficiency.

A corollary to the minimization of migrations is that if such migrations must be made, they should be made between cores which share as much of the cache hierarchy as possible. This is echoed by the second goal above. If a task is migrated to a core which shares a cache or two with its previous core, it is likely that many of its cache lines are still present in these caches. This is clearly preferable to migrating the task to a core which shares no caches with its current core, thereby forcing all memory references to be fulfilled by main memory.

While the previous goals consider tasks in isolation, the remaining goals consider tasks' relationships to each other. The first of these goals is to execute tasks which share memory on cores which are as cache-close to each other as possible. The motivation for this is that because these tasks share memory, they will potentially be accessing the same memory addresses, and pulling the relevant cache lines into their caches. If these memory-sharing

tasks share caches, they can actually help each other out – if one task requests memory from a cache line before the other, the second task may be able to avoid a cache miss, even if this is the first time it has accessed that memory! The opposite, executing memory-sharing tasks on cores which don't share caches, is as harmful as co-scheduling is advantageous. If the tasks are frequently accessing the same memory in this case, they will frequently cause cache invalidations on each others' caches.

Conversely, executing tasks which do not share memory on cache-close cores can be detrimental to performance. This is particularly true if the combined working set size of the tasks is larger than the size of the shared cache (or perhaps even less, depending on the cache associativity). When this is true, the tasks will cause capacity/conflict cache misses in their shared caches due to competing memory needs.

While the above goals are not an exhaustive list of the ways by which locality of reference can be preserved, therefore increasing execution efficiency, they are a summary of our own thoughts and the research of others in the area. These goals provide the basis for our work in this thesis.

4.4.2 Difficulties

Unfortunately, these goals may conflict and there is no clear consensus regarding their priority relative to each other. This lack of consensus is due to many factors: differing application timing behavior, differing cache access patterns, different remaining execution times, etc.

Example

We present here an example of conflicting goal priorities. If there are a set of memory-sharing tasks with a sum utility high enough that they cannot be feasibly partitioned onto one core, it may be necessary to execute one or more of these tasks on another core. If at some point in the future it becomes possible to migrate this task onto the core containing the other tasks with which it shares memory (for example, because one of the tasks in the group has finished), should this migration be made? On one hand, executing on the same core with other memory-sharing tasks will improve cache performance in the long run because they share all of the same caches. On the other hand, migrating the task to that processor could incur a fair number of up-front cache misses due to the task being at least partially cache-hot on its current processor. This example is illustrated in Figure 4.6.

In the above scenario, it is likely that the solution to finding the most efficient execution depends on how long the tasks in question will execute after the potential migration as well as the properties of any other tasks currently executing in the system. If the task is migrated onto the core with tasks with which it shares memory and almost immediately afterwards finishes execution, it is likely that the costs of the migration outweigh its benefits.

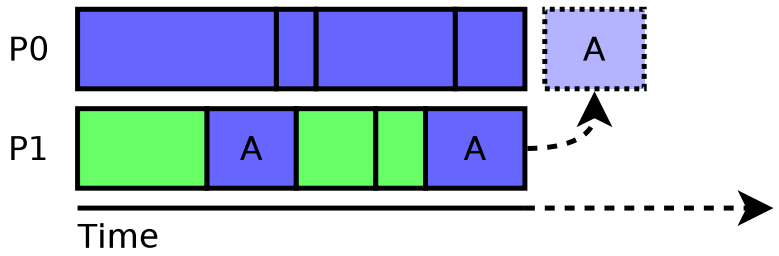


Figure 4.6: Demonstration of the "to migrate, or not to migrate?" dilemma. The blue tasks all share significant memory with each other. Task *A* was not previously able to execute on P0, but is now eligible to. Is it worth accepting a few up-front cache misses to migrate *A* from P1 to P0, or will these misses offset any gains from executing on the same cores with the other tasks with which it shares memory?

If, however, the task will execute for a long period of time after migration, it becomes more likely that the migration will be beneficial in the long run.

Further Uncertainty

In addition to conflicting goals, there are several other things which complicate the search for the *one true answer* for optimal real-time cache behavior. One of these complicating factors is ccNUMA. In ccNUMA, processing cores are divided (usually along physical processor boundaries) into nodes. Each node owns a certain amount of the system's memory and is able to access that memory much faster than any of the other nodes. Though accessing memory which belongs to another node will not cause cache misses, which we have been focusing on up until now, it will cause significantly higher memory access latencies and therefore decrease performance.

Another complicating factor is that of application memory characteristics. These characteristics can include working set size, how much memory application threads share with each other, memory access patterns, etc. Each of these factors has the potential to influence the cache-related behavior of the system as a whole. For example, the more memory the threads in Figure 4.6 share with each other, the more favorable making the migration becomes.

Finally, no two hardware architectures are exactly the same. These widely-varying architectural designs and behaviors make it difficult to design an algorithm generic enough to perform optimally (or possibly even well) on all potential platforms.

Due to the numerous aforementioned conflicts and variables, there does not (and perhaps cannot) exist a scheduling algorithm which provides optimal behavior with respect to memory latencies in all cases. This problem becomes even more complicated when dealing with real-time systems, due to the real-time guarantees which are necessary for correct application execution. Nevertheless, by attempting to take as many of these factors into account as

possible, we may create a more cache-aware scheduling algorithm.

4.4.3 Work Conserving Scheduling

One way to classify scheduling algorithms (real-time or not) is by whether they are work-conserving or non-work-conserving. A work-conserving scheduler will not, under any circumstances, allow a resource – in this case a processing core – to remain idle if it is aware of any other work which can be done. On the other hand, a non-work-conserving scheduler will allow resources to remain idle under certain circumstances, usually those circumstances which allow the system to optimize another metric at the expense of always executing as many tasks as possible.

A relevant example of a non-work-conserving scheduler would be one which refused to execute two tasks which shared memory at the same time. Such a scheduler would most likely reduce the overall number of cache misses, resulting in more efficient execution. If it were possible to put the non-used cores into an idle or sleep state, this execution could be more energy-efficient than a work-conserving one.

For this thesis, we scope out non-work-conserving schedulers. We base this decision on the observation that many real-world real-time systems are 'sporadic' in the sense that task arrival times are not known *a priori*. In cases where this is true, it makes sense to get as much work done as possible, as soon as possible, so that if other tasks do arrive at a later time, we have left them as much slack time in the schedule as possible. Therefore, we focus on work-conserving scheduling in this thesis.

With this said, it certainly does not mean that the techniques we discuss do not apply to non-work-conserving scheduling algorithms, but merely that we have not explicitly considered or studied the ramifications.

4.4.4 Summary

In this chapter, we have explored modern cache architectures and their effects on real-time systems. This exploration has provided us with the motivation of the existence of caches, as well as shown us under which conditions they perform well and under which they perform poorly. From these observations, we developed goals to guide our search for a cache-aware algorithm. Finally, we discussed the difficulties we may meet in attempting to meet these goals, as well as the trade-offs involved. This exploration of cache-awareness is a prerequisite to developing cache-aware algorithms from which real-time systems and applications can benefit.

Chapter 5

Cache-Aware Partitioning

Task partitioning algorithms are widely used in real-time scheduling because they eliminate migrations and are simple to use. However, eliminating migrations is not the full extent of 'cache-awareness', as we observed in Chapter 4. To our knowledge, no previous work has attempted to partition tasks with the goal of minimizing cache misses and improving overall execution efficiency.

To that end, the question we seek to answer in this chapter is: *How can we best partition multi-threaded real-time applications on platforms with complex and segregated memory/-cache hierarchies in order to decrease the execution time by minimizing cache misses and improving overall system efficiency?*

Partitioned, semi-partitioned, and clustered scheduling algorithms may be considered somewhat cache-aware, as they restrict task migrations. Restricting migrations makes it more likely for a task's working set to reside at least partially in the cache when it resumes execution, whereas a task migrating to a processor that shares no caches with the source processor would be forced to fetch its entire working set from main memory. By limiting the maximum cache distance a task is allowed to migrate, these approaches decrease the potential for costly cache misses. However, this does not mean there is no room for improvement among existing partitioned and clustered scheduling algorithms.

One problem with existing partitioning approaches is that it is possible for the tasks with the largest working set sizes to be distributed unevenly among the available processors. If the tasks partitioned on a particular core have a working set size (WSS) greater than the size of the largest cache (the last level cache, or LLC), it is impossible for their collective memory to be continuously housed in the LLC over the lifetime of the application. As the collective per-core WSS approaches and passes the size of the LLC, the number of cache misses per task execution is likely to increase.

Furthermore, tasks may share memory with each other. It is advantageous to schedule memory-sharing tasks on either the same core or cores a small cache distance apart from one

another for two reasons: first, scheduling them on 'cache-close' cores increases the likelihood that the previously-running task may share memory with the current task, keeping the cache warmer and decreasing the number of cache misses, and second, scheduling memory-sharing tasks on different cores will result in a cache-invalidation whenever they modify memory which shares the same cache line as memory currently residing in the other's cache. Cache invalidations both cause more traffic on the shared bus and force the core whose cache line was invalidated to acquire a new copy of that memory whenever it is accessed again. Avoiding such contention is a priority.

5.1 Goals

With the cache-related deficiencies of current partitioning algorithms in mind, we turn our discussion to the design of a partitioning scheme that is more cache-aware. Building on our previous observations, we formulate two goals for our cache-aware partitioning scheme above those for traditional partitioning:

1. Evenly distribute the collective WSS of a taskset over all cores to decrease conflict and capacity-related cache misses
2. Partition tasks that share memory so the sum of the cache distance between all such pairs of tasks is minimized

Our hope is that if we can create a partitioning algorithm which meets both of these goals, that algorithm will produce partitionings which execute more efficiently than existing partitioning heuristics.

5.2 Existing Heuristics

Because bin-packing, and therefore partitioning, is NP-hard, several polynomial-time heuristics have been developed to approximate solutions to this problem. The simplest set of such heuristics includes algorithms such as best-fit, worst-fit, first-fit, and next-fit [26]. In the context of partitioning, these heuristics consider tasks one-by-one in some order, and place the task under consideration onto a particular core based on that heuristic's criteria. When partitioning, we assume m bins, each with a capacity of one, where the 'size' of a task is determined by its density (or utilization in constrained sporadic systems). The capacity of the bin represents its available processing time, measured in the total utilization/density available to tasks.

The best-fit heuristic places a task in the bin with the smallest remaining capacity that can still accommodate it, beginning with the first bin. The worst-fit heuristic places a task in

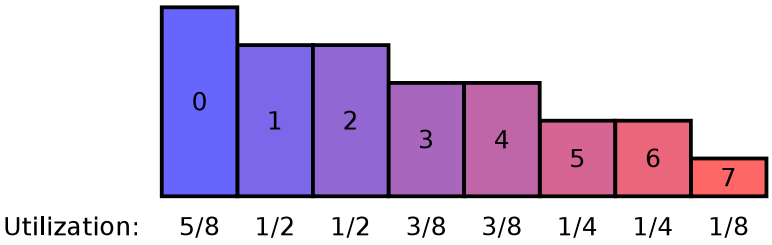


Figure 5.1: Example taskset, with tasks displayed in order of non-increasing utilization.

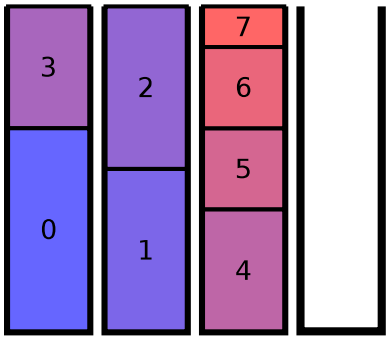


Figure 5.2: Tasks from Figure 5.1 partitioned onto four processors using the best-fit heuristic.

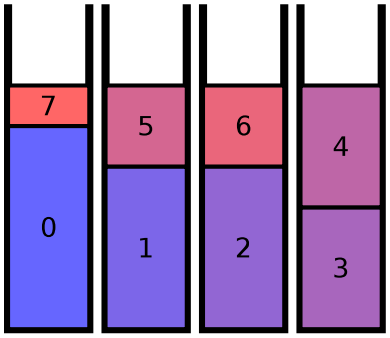


Figure 5.3: Tasks from Figure 5.1 partitioned onto four processors using the worst-fit heuristic.

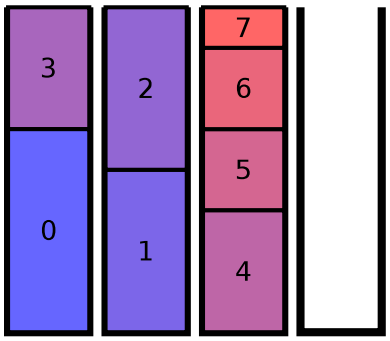


Figure 5.4: Tasks from Figure 5.1 partitioned onto four processors using the first-fit heuristic.

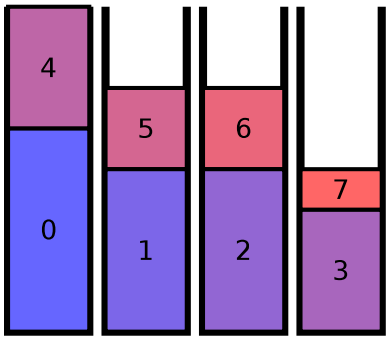


Figure 5.5: Tasks from Figure 5.1 partitioned onto four processors using the next-fit heuristic.

the bin with the largest remaining capacity, starting with the first bin. The first-fit heuristic places a task in the first bin it encounters with sufficient capacity, beginning with the first bin. Finally, the next-fit heuristic places a task in the first bin it encounters with sufficient capacity, starting with the bin immediately following the bin in which the last task was placed.

For each of these heuristics, there are a myriad of different orders in which they may consider the tasks. There have been many variations, but the most common are non-decreasing relative deadline order [12] (i.e. $\forall i \in [0, n) : d_i \leq d_{i+1}$) and non-increasing utilization order [13] ($\forall i \in [0, n) : u_i \geq u_{i+1}$). In addition to these two orderings, tasks may be ordered by period, utility, execution time, working set size, or virtually any other task attribute one can think of. Ordering tasks by utilization is common because this is the easy analog to 'size', which is used by the *-fit heuristics for bin-packing, and is also the metric used to determine whether the task "fits" in a particular bin (i.e. whether that processor has enough unclaimed processing time to accomodate the task under consideration).

Figures 5.2-5.5 demonstrate the partitionings which would result from using these four simple partitioning heuristics on the taskset displayed in Figure 5.1. For each heuristic, the tasks are considered in the order in which they are numbered, in non-increasing utilization order. As we can see from these diagrams, the best-fit and first-fit algorithms tend to clump the tasks onto as few of cores as possible, while the worst-fit and next-fit algorithms spread out the tasks more evenly.

5.3 Largest WSS First Algorithm

We now present the LWFG (Largest WSS First, Grouping) partitioning algorithm. This algorithm is a direct result of our previous observations on the deficiencies of other partitioning algorithms in terms of cache-awareness, and attempts to satisfy our goals from earlier in this chapter.

In an attempt to address our first goal from Section 5.1, evenly distributing WSS across cores, we propose partitioning the tasks with a next-fit heuristic in order of non-increasing working set size (i.e. $\forall i \in [0, n) : w_i \geq w_{i+1}$). The next-fit heuristic ensures that each core has a task with one of the m largest working set sizes in the taskset before adding a second task to any core.

To address our second goal, scheduling tasks which share memory on cache-close cores, we modify our partitioning algorithm to group tasks together if they share memory. Tasks are still considered in non-increasing WSS order, but whenever we consider a task, we also consider all tasks that share memory with our current task that have not previously been partitioned. The algorithm attempts to schedule this group of tasks as a whole – the sum of their task densities is considered when determining if a core has sufficient remaining capacity to schedule the group. If the next-fit heuristic fails to find a core with sufficient capacity to partition the group, the task that shares the least memory with the first task is removed from the group, and partitioning is retried with the smaller group. Pseudocode for this algorithm, which we term LWFG for 'Largest WSS First, Grouping', is presented in Listing 1.

Algorithm 1 Partition Largest WSS First, Grouped by Shared Memory

```
1: sort_decreasing_wss(taskset)
2: last ← 0
3: for all task ∈ taskset do          ▷ Find group members which share memory with task
4:     group ← get_memory_sharers(taskset, task)
5:     partitioned ← False
6:     while partitioned == False do
7:         for all cpu ∈ cpus start after last do   ▷ Loop through cpus starting after last assigned
8:             if capacity_left(cpu) ≥ density(group) then
9:                 partition(group, cpu)
10:                last ← cpu
11:                partitioned ← True
12:                break
13:            end if
14:        end for
15:        if num_tasks(group) > 1 then              ▷ Remove task from group if possible
16:            remove_least_shared(group)
17:        else
18:            return Partitioning Failed
19:        end if
20:    end while
21:    taskset.remove(group)
22: end for
23: return Partitioning Succeeded
```

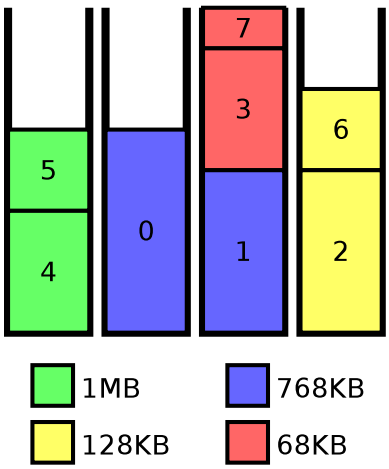


Figure 5.6: LWFG partitioning of a simple taskset onto four processors.

Figure 5.6 presents an example partitioning of the LWFG algorithm. To obtain this partitioning, the available tasks were first sorted by WSS. The task with the largest WSS was then chosen, and tasks which share memory with it were gathered together to form a group. This group is the green group in the example. The green group was then partitioned as one unit onto the first available processor by the next-fit heuristic.

The next un-partitioned task with the largest WSS was then chosen, and its memory-sharing tasks found, forming the blue group. Unfortunately, this group has a sum utilization (density for non-constrained sporadic task systems) greater than one. This means the entire group cannot be feasibly partitioned onto one core. In this case, the algorithm removes group members until the total utilization of the group is such that it can be feasibly partitioned [1].

This process is continued with the remaining tasks/groups until they are all partitioned. In this example, only one of four memory-sharing task groups was forced to be split across processor boundaries, and this was necessitated by the tasks' utilizations, not forced by the algorithm itself.

Figure 5.7 shows the same taskset, partitioned using the first-fit heuristic with tasks considered in non-increasing utilization order (i.e. FFD). In this example, we can see that many of the tasks which share memory are not mapped to the same cores. This partitioning would therefore most likely cause increased memory traffic, cache invalidations, and cache misses, resulting in a less efficient execution. Because other partitioning algorithms/heuristics do not consider cache effects when partitioning, they too will produce similarly-scattered partitionings.

[1] See lines 15-16 of Listing 1

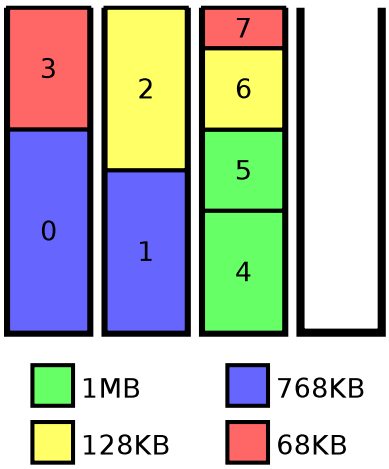


Figure 5.7: FFD partitioning of a simple taskset onto four processors.

5.4 Algorithmic Complexity

The worst-case complexity of LWFG is $O(n^2 \times m)$, where m is the number of processors and n is the number of tasks in the taskset being partitioned. Though this is more complex than the simple *-fit algorithms by a factor of n, we were able to partition approximately 50,000 tasksets using a simple Python implementation of the algorithm in less than 5 minutes on a 1.6Ghz processor (with no parallelism). We do not see running time as an impediment to this algorithm's adoption.

5.5 Feasibility

It is known that uniprocessor EDF can schedule any taskset T for which the sum of the task densities is no greater than one [23]:

$$\sum_{\tau_i \in T} \lambda_i \leq 1$$

Let π_j denote processor j and $P(\pi_j)$ denote all tasks partitioned to that processor. If all tasks in T are able to be partitioned using LWFG such that the total task density on any one core is less than one, that is

$$\forall j \in [0, m) : \sum_{t_i \in P(\pi_j)} \lambda_i \leq 1$$

then the taskset is schedulable on m instances of uniprocessor EDF using our partitioning strategy.

We do not attempt to establish a feasibility test for our partitioning scheme as a whole. Instead, we suggest that systems which require hard real-time schedulability 'fall back' to other partitioning algorithms (or perhaps even a global algorithm, if necessary) if LWFG fails to partition a taskset. This approach will improve performance when possible using our cache-aware partitioning, while still making schedulability guarantees.

Chapter 6

Experimental Evaluation

In order to test the LWFG partitioning strategy, we compared its performance to three of the the most widely used static partitioning schemes: worst-fit in order of non-increasing task utilization (WFD) [68, 13], first-fit in order of non-increasing task utilization (FFD) [8], and Baruah and Fisher's first-fit in order of non-decreasing relative deadline order (BF) [12]. We believe these partitioning schemes are representative of existing static partitioning schemes for purposes of comparison.

We leave comparison of our algorithm against semi-partitioned, clustered, and global real-time systems as future work. Such systems offer higher theoretical schedulability bounds at the cost of an increased number of migrations. Therefore, they are typically used by systems which have different requirements than those which use traditional partitioning schemes.

6.1 Methodology

We implemented and tested the LWFG partitioning scheme on ChronOS Linux[28], a real-time extension to the Linux kernel. The code for our kernel, userspace utilities, and testing infrastructure is available online at `http://chronoslinux.org`.

Similarly to the approach taken by Baker [9] and Calandrino and Anderson [21], we randomly generated synthetic tasksets based on certain allowable parameter ranges. All tasksets were first partitioned offline using Python scripts before being scheduled under ChronOS with one instance of uniprocessor EDF per core. Each data point in the plots below represents at least 25 different tasksets/runs unless otherwise indicated.

6.1.1 Test Application

We have created a complex yet flexible application to test the properties of scheduling algorithms and partitioning schemes. This test application, which we call `sched_test_app`, runs a set of tasks, specified by a taskset file, and measures that taskset's execution properties. The properties measured include deadline satisfaction ratio, accrued utility ratio, task tardiness, etc. These metrics collected from our test application enable us to objectively compare real-time scheduling methods.

The taskset files used by our application supply all the necessary real-time parameters. These parameters include the task period p_i, execution time e_i, and deadline d_i. In addition, the taskset file specifies the WSS of each individual task, which tasks belong to a group, and the WSS that all tasks in that group share with each other. This grouping mechanism allows for the representation of real-time systems which feature both memory-sharing tasks and non-memory-sharing tasks, while giving fine-grained control over their memory usage. The tasks in a group are created so that they are all in the same thread group - i.e. all share the same memory space. We find this to be a suitable way to create and manage the test application threads because it is the traditional way real-time applications are created and managed - one process per application, and threads within that process.

By supporting any combination of the aforementioned parameters, and allowing textual file-based and command line configuration, we believe we have created a flexible general-purpose real-time scheduler testing tool.

6.1.2 Workload

In our evaluation, each task accesses its memory in a sequential pattern, implemented by incrementing elements in an array in strides equal to the cache line size (64 bytes on our platform). We believe this memory access pattern is generic enough to be representative of other memory-intensive real-time applications. For example, the fast Fourier transform algorithm largely accesses an array in-order, doing computation on each element in varying strides, depending on which pass of the algorithm it is on. Many other digital signal processing applications also have largely in-order memory access patterns, including multimedia encoding and decoding.

6.1.3 Taskset Distributions

Calandrino and Anderson define their tasksets in terms of multi-threaded tasks, or MTTs, which each consist of several individual tasks that share memory with each other [21]. All our randomly-generated tasksets were generated using this MTT model. Tasks within an MTT share the entirety of their WSS with one another, and no tasks not in the same MTT share

Table 6.1: Taskset Distribution Parameter Bounds

Taskset	Tasks per MTT	Utilization	u_i is per-	Period (ms)	p_i is per-	WSS (kB)
MLU	$[1, 4]$	$[0.01, 0.1]$	MTT	$[24, 240]$	MTT	$e_i/3 * 128$
MMU	$[1, 4]$	$[0.1, 0.4]$	MTT	$[24, 240]$	MTT	$e_i/3 * 128$
MWL	$[1, 8]$	$[0.1, 0.4]$	MTT	$[10, 250]$	MTT	$[64, 512]$
MWH	$[1, 8]$	$[0.1, 0.4]$	MTT	$[10, 250]$	MTT	$[4096, 8192]$
MWLP	$[1, 8]$	$[0.1, 0.4]$	task	$[10, 250]$	MTT	$[64, 512]$
MWHP	$[1, 8]$	$[0.1, 0.4]$	task	$[10, 250]$	MTT	$[4096, 8192]$
MWLU	$[1, 8]$	$[0.1, 0.4]$	MTT	$[10, 250]$	task	$[64, 512]$
MWHU	$[1, 8]$	$[0.1, 0.4]$	MTT	$[10, 250]$	task	$[4096, 8192]$

memory (with the exception of the application and shared library text, which is minimal).

In addition to testing our algorithm with two of their taskset distributions, we generate our own distributions in which we relax some of the restrictions on certain parameters in order to test our algorithm's applicability to a wider range of real-world applications. For instance, we randomly vary the WSS independently of the execution time of each task inside of a particular range. In other variants of these tasksets, we additionally allow either the period or utilization to vary for tasks within an MTT. It is worth noting that although we claim our algorithm is generalizable to the sporadic task model, for simplicity we only test with periodic tasks where $p_i = d_i$.

When generating the tasksets we create MTTs one at a time, keeping track of the sum utilization of all tasks added to a particular taskset thus far. If adding the MTT currently under consideration would place the taskset over the taskset utilization bound, the MTT is discarded and the taskset is completed. Therefore, for our experimental results which follow, the taskset utilization is not a precise value, but instead a close upper bound.

To generate each MTT, the number of threads the MTT will contain is first chosen randomly within the specified bounds. Next, the period and utilization for the MTT are randomly generated, from which the execution time is calculated. Finally, the WSS is either randomly generated or calculated based off of the execution time for the MTT, depending on the taskset distribution. Note that for those tasksets that allow periods or utilizations to be different between tasks in the same MTT, the previous steps are per-task rather than per-MTT.

Table 6.1 describes the exact distributions tested and matches them with their names as we will refer to them for the remainder of this paper. The first two taskset distributions are slightly-modified distributions borrowed from [21], while the remaining distributions are of our own design. For those of our own design, we aim to test conditions not covered by previous taskset distributions – namely working set sizes which vary independently of execution times, and multi-threaded tasks which share memory but do not necessarily share

periods or utilizations. Dropping these previous assumptions and testing the effects of these parameter changes will help us to better understand the cache-related performance of real-time applications using the studied partitioning schemes.

Note that we exclude from this study taskset distributions which contain tasks with extremely large utilizations. As the average task utilization in a taskset approaches and passes $\frac{1}{2}$, the benefits seen from our approach will diminish. This is due to the fact that if all task utilizations are greater than $\frac{1}{2}$, no two tasks can feasibly be placed on the same core, which results in a much smaller degree of cache-sharing than is possible if more than one memory-sharing task reside on the same core. We do not present results with tasksets that contain tasks with utilizations greater than $\frac{1}{2}$ because our partitioning algorithm will deteriorate to other existing known algorithms under these conditions.

6.1.4 Task Arrival Pattern

In our description of the LWFG algorithm, we note that it is capable of handling constrained sporadic tasks in addition to the traditional period task model. However, to avoid possibly unpredictable interactions between sporadic tasks, and to reduce overall complexity, we have elected to test only periodic tasksets. We believe our results are still generalizable to the constrained sporadic task model, but do recommend that a similar experimental study be undertaken specifically for this model to verify our claim.

6.1.5 Partitioning

Because we consider tasksets with sum utilizations close to that of the theoretical maximum of the machine, it is inevitable that some partitionings will not succeed. In order to cope with this inevitability while maintaining fairness between algorithms, we partition tasks using the given partitioning algorithm as long as that partitioning algorithm can feasibly partition the task being considered (using the uniprocessor feasibility bound). If the algorithm encounters a task it cannot feasibly partition, that task is placed onto the core which has the least sum utilization.

We note that this strategy of partitioning tasks in overload conditions is not particularly beneficial to our algorithm, as it will tend to distribute tasks randomly around the cores. Therefore, we are not biasing our results in favor of the LWFG partitioning algorithm.

6.1.6 `perf`: The Linux Profiling Tool

We use the Linux `perf` [3] tool to measure information using the performance monitoring units (PMU's) present on our test architecture. `perf` is a hybrid kernel/userspace mechanism

Table 6.2: 12-core AMD Opteron™ 6164 HE Memory Hierarchy

Level	Shared Between	Size	Associativity
L1-I	1 core	64 Kb	2-way
L1-D	1 core	64 Kb	2-way
L2	1 core	512 Kb	16-way
L3	6 cores	5118 Kb	48-way
NUMA Node	12 cores	4 Gb	n/a
Main Memory	48 cores	16 Gb	n/a

in Linux that is able to track hardware performance counters on a per-system, per-core, or per-process basis. Using `perf`, we track the number of lowest-level cache (LLC) loads and misses, the number of instructions executed, and the number of cycles.

6.1.7 Hardware Platform

We tested our partitioning algorithm using the aforementioned tasksets on a modern many-core hardware platform: a machine with four 12-core AMD *Magny-Cours* Opteron processors running at 1.7 Ghz. See table 6.2 for the memory hierarchy information for this machine. Using this platform, we tested tasksets with total utilization caps up to the total number of cores on the platform: 48.

6.2 Results

6.2.1 Instructions Per Cycle Results

The instructions per cycle of each taskset distribution at each load tested on our 48-core platform are presented in Figures 6.1-6.8 (Tables 6.3-6.10 contain the numerical results). Instructions per cycle, or IPC, is widely used as a measure of execution efficiency. An improvement in IPC shows that a partitioning scheme not only decreases the total number of cache misses, but that this decrease causes more efficient execution overall.

Observation 1. *The LWFG partitioning scheme more than doubles the IPC of WFD in some cases.* WFD is widely used [13, 68] and is praised for its ability to evenly distribute the total taskset utilization among all available cores [14, 8]. The LWFG scheme outperforms WFD for all distributions and all loads, and rarely demonstrates an improvement in IPC of less than 20%. Our results show that cache-aware partitioning is an extremely important

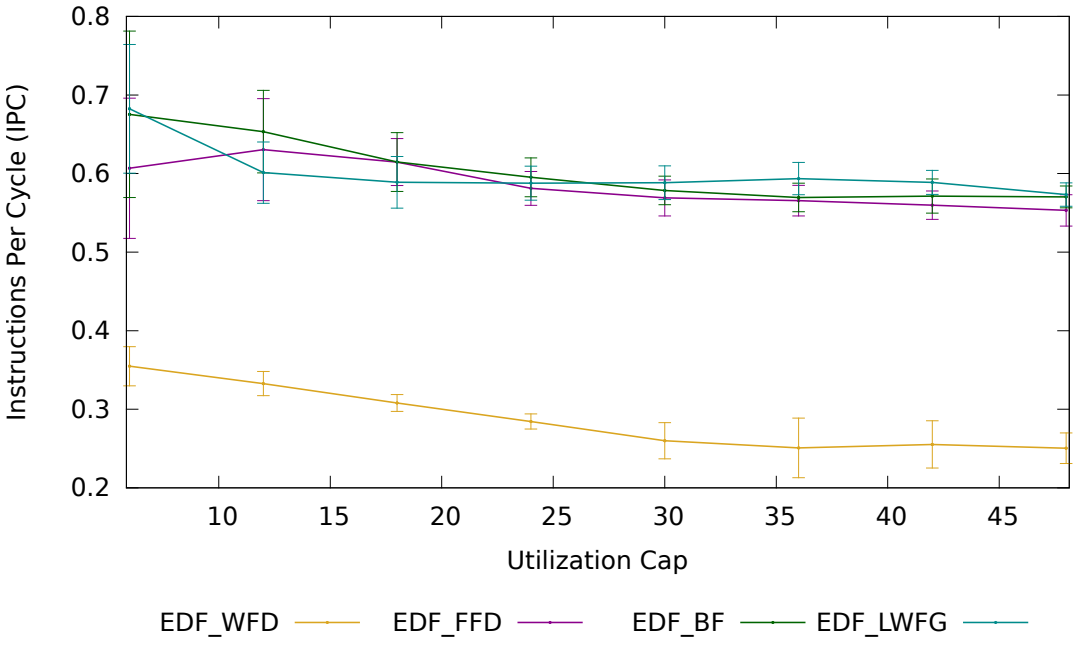


Figure 6.1: IPC for MLU Distribution

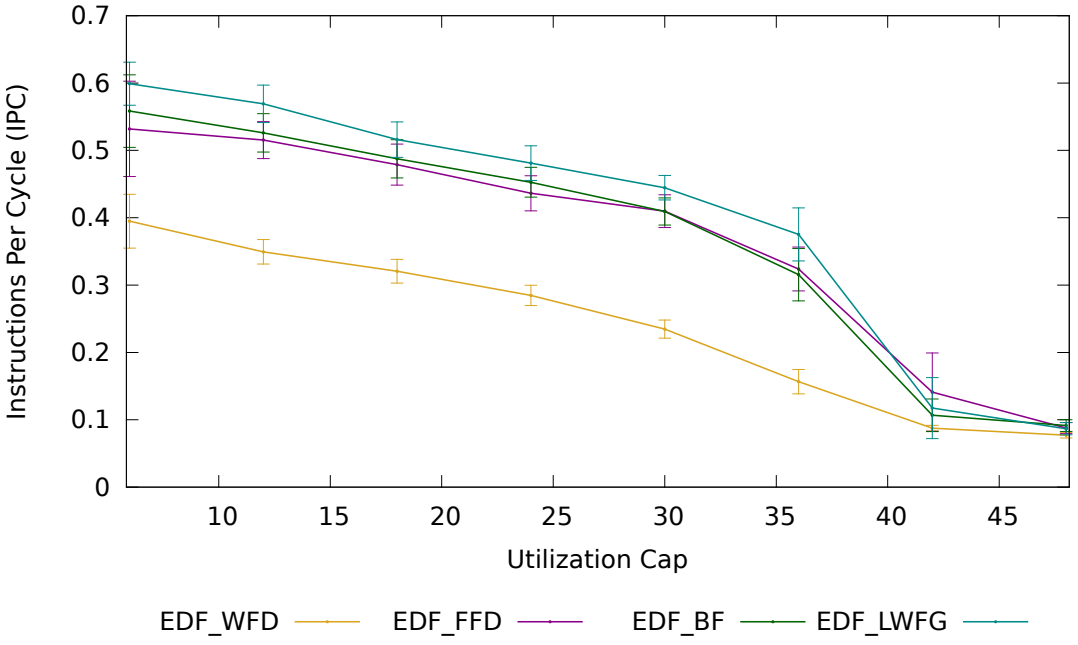


Figure 6.2: IPC for MMU Distribution

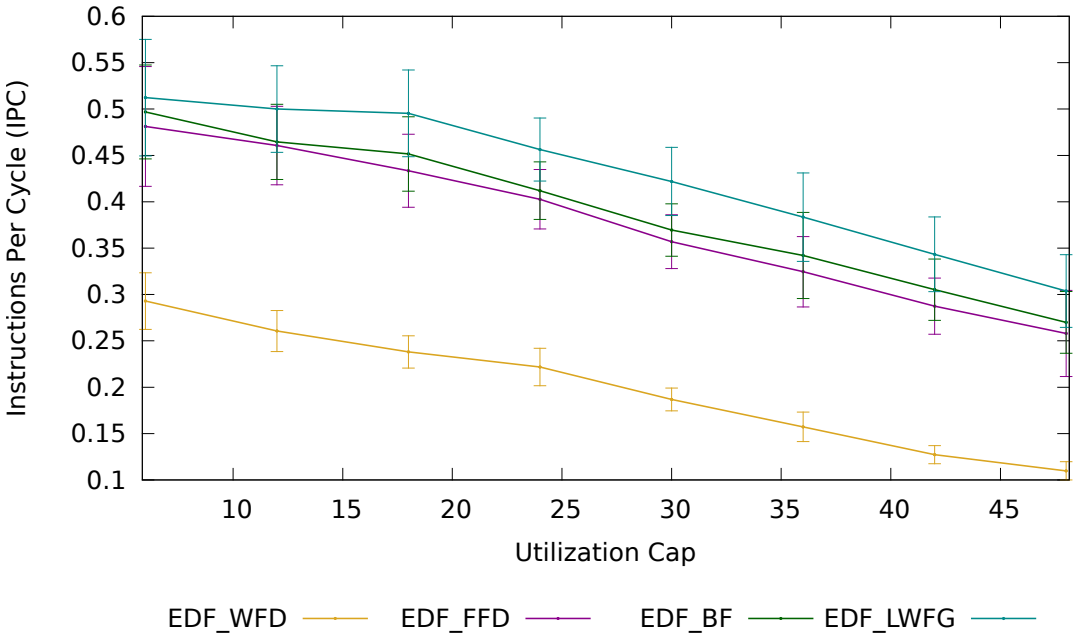


Figure 6.3: IPC for MWL Distribution

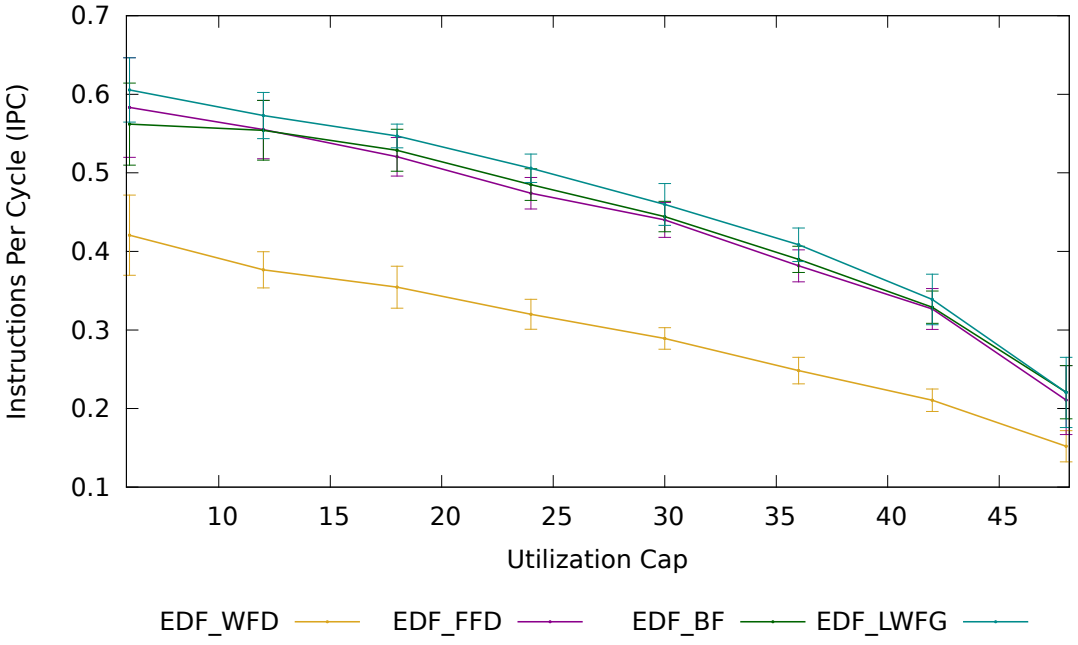


Figure 6.4: IPC for MWH Distribution

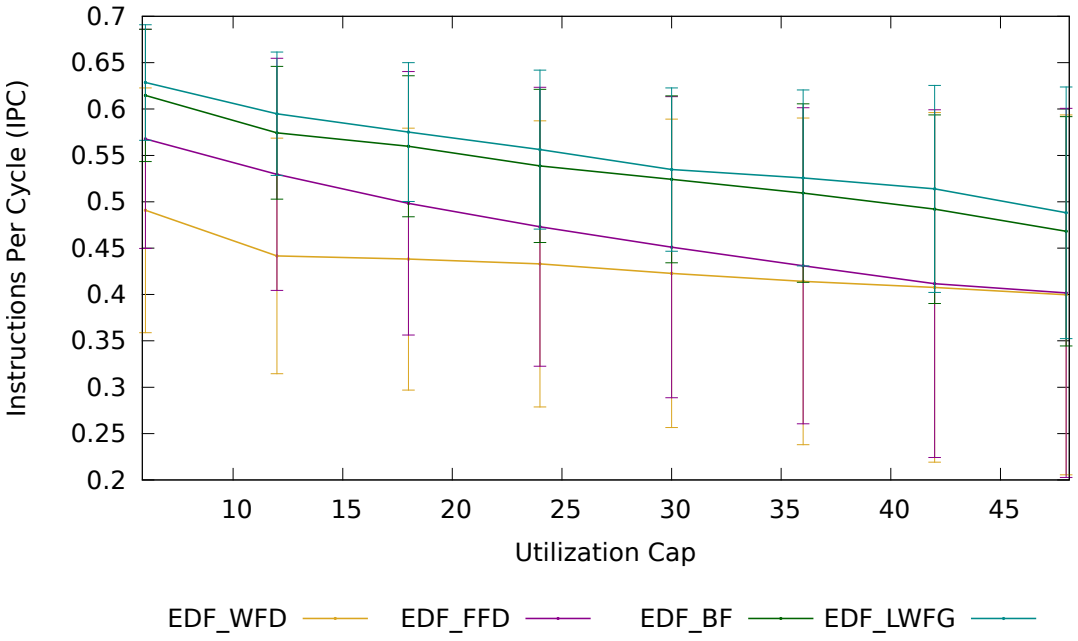


Figure 6.5: IPC for MWLP Distribution

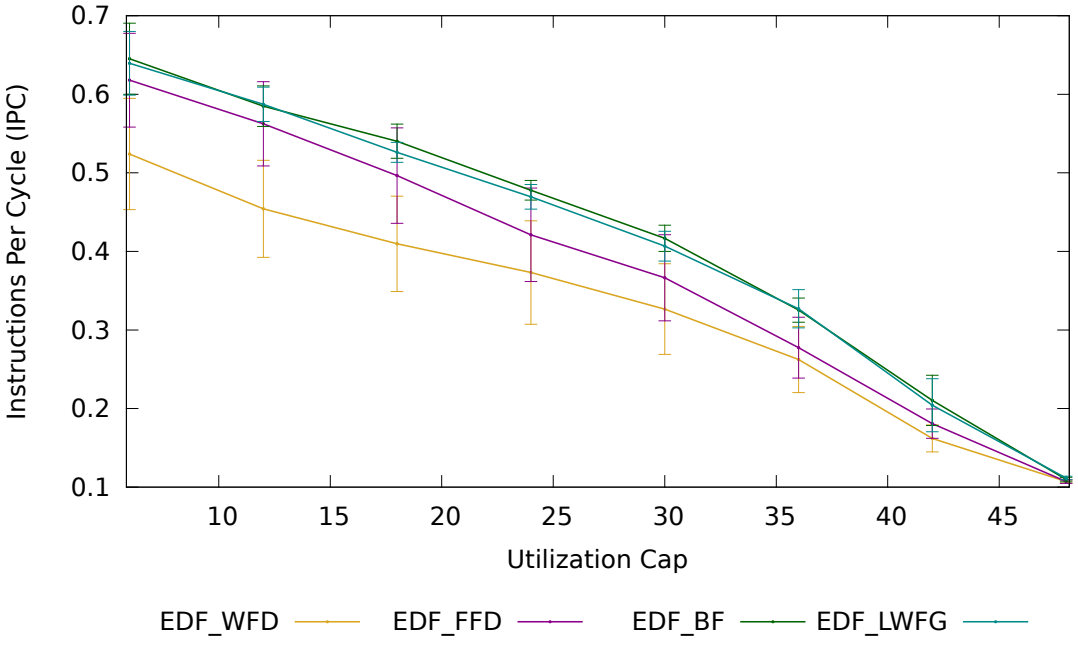


Figure 6.6: IPC for MWHP Distribution

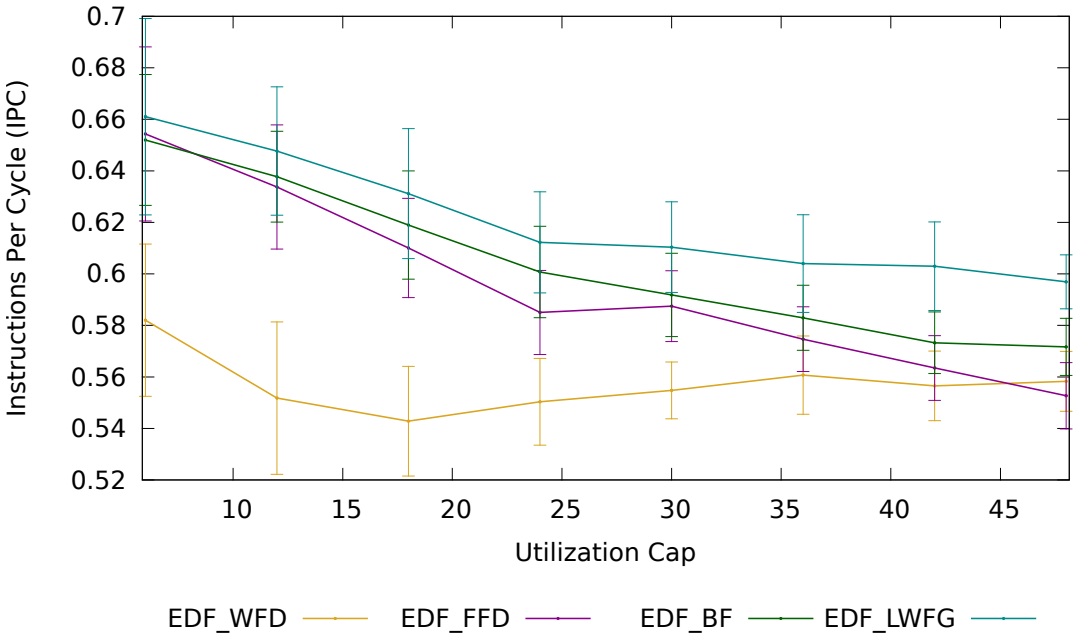


Figure 6.7: IPC for MWLU Distribution

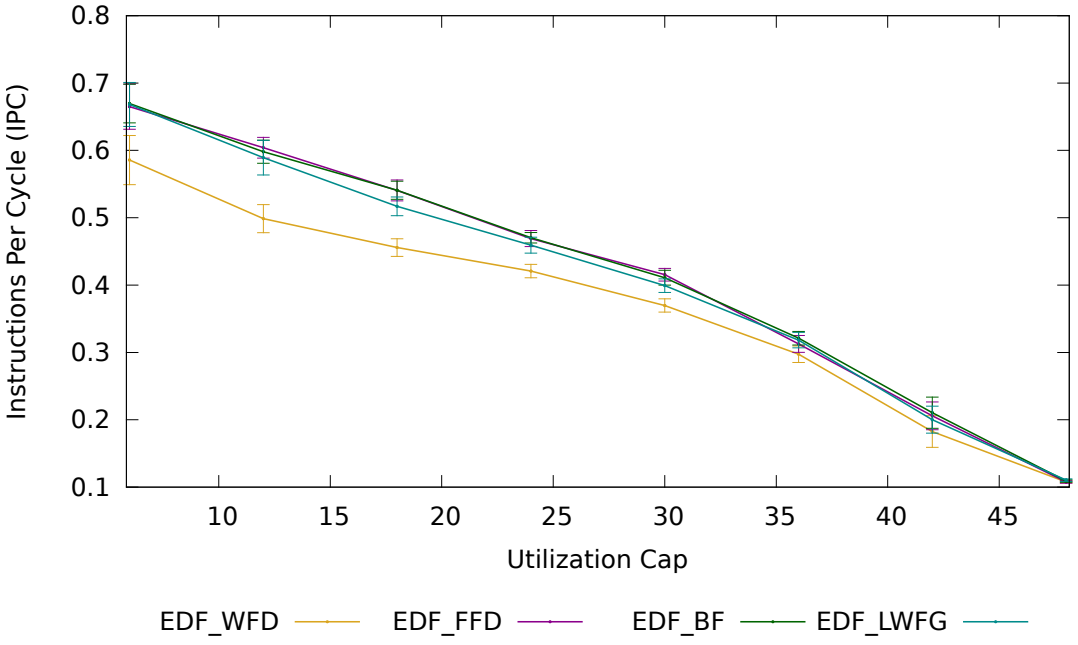


Figure 6.8: IPC for MWHU Distribution

Table 6.3: IPC for MLU Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	IPC	Stdev	IPC	Stdev	IPC	Stdev	IPC	Stdev
6	0.35477	0.02494	0.60674	0.08923	0.67538	0.10590	0.68237	0.08196
12	0.33266	0.01541	0.63042	0.06499	0.65332	0.05252	0.60122	0.03900
18	0.30797	0.01070	0.61465	0.02981	0.61469	0.03749	0.58887	0.03282
24	0.28445	0.00962	0.58117	0.02154	0.59529	0.02472	0.58775	0.02165
30	0.25991	0.02304	0.56902	0.02290	0.57848	0.01822	0.58832	0.02150
36	0.25083	0.03788	0.56556	0.01942	0.56957	0.01820	0.59360	0.02058
42	0.25529	0.03006	0.55976	0.01803	0.57135	0.02181	0.58879	0.01525
48	0.25044	0.01951	0.55312	0.01994	0.57027	0.01393	0.57322	0.01491

Table 6.4: IPC for MMU Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	IPC	Stdev	IPC	Stdev	IPC	Stdev	IPC	Stdev
6	0.39483	0.03985	0.53188	0.07079	0.55845	0.05386	0.59903	0.03189
12	0.34940	0.01811	0.51532	0.02740	0.52608	0.02848	0.56914	0.02782
18	0.32062	0.01757	0.47888	0.03046	0.48756	0.02839	0.51606	0.02628
24	0.28478	0.01507	0.43633	0.02600	0.45268	0.02207	0.48113	0.02576
30	0.23473	0.01341	0.40980	0.02437	0.40931	0.02010	0.44461	0.01819
36	0.15658	0.01815	0.32392	0.03253	0.31551	0.03891	0.37529	0.03941
42	0.08754	0.00422	0.14104	0.05817	0.10668	0.02415	0.11737	0.04535
48	0.07721	0.00417	0.08774	0.00824	0.09140	0.00868	0.08670	0.00916

Table 6.5: IPC for MWL Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	IPC	Stdev	IPC	Stdev	IPC	Stdev	IPC	Stdev
6	0.29290	0.03054	0.48117	0.06462	0.49684	0.05063	0.51228	0.06285
12	0.26063	0.02210	0.46069	0.04231	0.46462	0.04056	0.49999	0.04665
18	0.23809	0.01742	0.43339	0.03938	0.45155	0.04013	0.49533	0.04673
24	0.22180	0.02017	0.40267	0.03206	0.41197	0.03108	0.45632	0.03405
30	0.18681	0.01224	0.35692	0.02902	0.36959	0.02826	0.42185	0.03676
36	0.15726	0.01588	0.32457	0.03795	0.34211	0.04642	0.38340	0.04772
42	0.12726	0.00970	0.28745	0.03026	0.30520	0.03303	0.34330	0.04029
48	0.10984	0.00983	0.25797	0.04637	0.26989	0.03329	0.30374	0.03923

Table 6.6: IPC for MWH Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	IPC	Stdev	IPC	Stdev	IPC	Stdev	IPC	Stdev
6	0.42064	0.05094	0.58316	0.06346	0.56200	0.05216	0.60547	0.04097
12	0.37653	0.02307	0.55512	0.03711	0.55409	0.03802	0.57296	0.02941
18	0.35447	0.02671	0.52048	0.02450	0.52875	0.02668	0.54693	0.01513
24	0.31993	0.01907	0.47411	0.02006	0.48506	0.02010	0.50587	0.01806
30	0.28922	0.01369	0.44014	0.02224	0.44433	0.01939	0.45969	0.02661
36	0.24826	0.01687	0.38165	0.02029	0.38987	0.01676	0.40848	0.02135
42	0.21060	0.01421	0.32666	0.02603	0.32901	0.02050	0.33887	0.03213
48	0.15202	0.01984	0.21077	0.04380	0.22080	0.03384	0.22055	0.04466

Table 6.7: IPC for MWLP Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	IPC	Stdev	IPC	Stdev	IPC	Stdev	IPC	Stdev
6	0.49082	0.13201	0.56782	0.11822	0.61466	0.07130	0.62859	0.06233
12	0.44159	0.12701	0.52960	0.12522	0.57438	0.07163	0.59490	0.06652
18	0.43814	0.14130	0.49834	0.14209	0.55983	0.07601	0.57514	0.07484
24	0.43302	0.15426	0.47309	0.15040	0.53861	0.08263	0.55625	0.08565
30	0.42282	0.16631	0.45101	0.16238	0.52421	0.09001	0.53474	0.08803
36	0.41416	0.17617	0.43103	0.17044	0.50939	0.09616	0.52576	0.09489
42	0.40761	0.18851	0.41169	0.18748	0.49203	0.10170	0.51390	0.11162
48	0.39962	0.19405	0.40176	0.19906	0.46816	0.12368	0.48814	0.13565

Table 6.8: IPC for MWHP Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	IPC	Stdev	IPC	Stdev	IPC	Stdev	IPC	Stdev
6	0.52391	0.07078	0.61795	0.05967	0.64519	0.04527	0.63947	0.04055
12	0.45421	0.06169	0.56246	0.05359	0.58485	0.02575	0.58716	0.02185
18	0.40963	0.06067	0.49647	0.06069	0.54027	0.02172	0.52608	0.01270
24	0.37313	0.06578	0.42114	0.05955	0.47776	0.01251	0.46949	0.01563
30	0.32663	0.05764	0.36643	0.05487	0.41663	0.01665	0.40670	0.01895
36	0.26241	0.04215	0.27755	0.03876	0.32527	0.01538	0.32703	0.02427
42	0.16202	0.01715	0.18084	0.01861	0.21039	0.03203	0.20418	0.03372
48	0.10710	0.00193	0.10712	0.00231	0.10966	0.00293	0.11160	0.00223

Table 6.9: IPC for MWLU Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	IPC	Stdev	IPC	Stdev	IPC	Stdev	IPC	Stdev
6	0.58200	0.02955	0.65435	0.03380	0.65198	0.02540	0.66105	0.03813
12	0.55177	0.02960	0.63374	0.02408	0.63776	0.01763	0.64770	0.02491
18	0.54282	0.02130	0.61005	0.01926	0.61899	0.02102	0.63116	0.02524
24	0.55034	0.01685	0.58503	0.01631	0.60074	0.01773	0.61225	0.01964
30	0.55475	0.01102	0.58748	0.01373	0.59183	0.01616	0.61037	0.01759
36	0.56070	0.01520	0.57465	0.01258	0.58293	0.01263	0.60399	0.01895
42	0.55651	0.01352	0.56348	0.01257	0.57326	0.01194	0.60297	0.01720
48	0.55826	0.01163	0.55269	0.01286	0.57165	0.01110	0.59692	0.01049

Table 6.10: IPC for MWHU Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	IPC	Stdev	IPC	Stdev	IPC	Stdev	IPC	Stdev
6	0.58569	0.03646	0.66485	0.03358	0.66962	0.02873	0.66826	0.03266
12	0.49859	0.02082	0.60402	0.01539	0.59810	0.01714	0.58924	0.02574
18	0.45569	0.01310	0.54058	0.01552	0.54089	0.01346	0.51688	0.01395
24	0.42087	0.00995	0.46909	0.01189	0.47032	0.00775	0.45925	0.01179
30	0.36973	0.00985	0.41542	0.00940	0.41094	0.01108	0.39925	0.01008
36	0.29740	0.01230	0.31267	0.01245	0.32115	0.01012	0.31831	0.01153
42	0.18250	0.02349	0.20580	0.02066	0.21048	0.02317	0.20014	0.02005
48	0.10760	0.00142	0.10756	0.00184	0.10901	0.00185	0.11040	0.00196

area of research, because the cache-related effects of a commonly-used partitioning algorithm are not understood and have been shown to be quite detrimental to execution efficiency.

Observation 2. *The LWFG partitioning scheme performs more consistently in terms of IPC than any of the tested partitioning algorithms.* Although the LWFG partitioning scheme does not obtain significantly higher IPC than all of the competitors in all cases, it never performs significantly worse than the best algorithm for any distribution at any load. This result demonstrates that while competing partitioning schemes perform on par with LWFG for some taskset distributions in terms of IPC, they do not perform as consistently as LWFG across the board.

For example, both the FFD (first-fit, non-increasing utilization) and BF (first-fit, non-decreasing relative deadlines) partitioning algorithms perform nearly as well as LWFG for the MWL and MWH distributions. In these distributions, both periods and utilizations are the same for all tasks within an MTT. Because FFD and BF consider tasks in the order of task utilizations or periods when partitioning, they 'accidentally' partition most tasks in an MTT to the same core, similarly to LWFG. The performance gains by LWFG over the other algorithms are due to the fact that LWFG more evenly distributes the working set over available cores and is intentional about grouping the tasks with the largest working set sizes together first. In contrast to the MWL and MWH results, we see that the FFD algorithm does not perform nearly as well on the MWHP and MWLP taskset distributions. This is because tasks within an MTT in one of these distributions do not share the same utilizations. Therefore, partitioning tasks in order of decreasing utilization is much less likely to 'accidentally' result in a cache-aware partitioning.

Observation 3. *The LWFG partitioning scheme does not out-perform the FFD and BF schemes in all cases.* We note that for those taskset distributions where the periods of tasks in a MTT are not all the same, but are randomly-generated on an per-task basis (MWLU, MWHU), the performance of our algorithm drops somewhat relative to the competitors, and the IPC of LWFG is even less than that of FFD and BF on some loads. This decrease in IPC improvement is at least partially due to the timing of the task periods and deadlines. Two tasks which share the same period and are partitioned on the same core will tend to be scheduled immediately after one another in uniprocessor EDF (assuming no task with an earlier deadline arrives in the meantime). The same two tasks, if partitioned to different cores, will tend to execute at the same time if the other tasks partitioned to their processors have similar periods and deadlines relative to their own. This means that tasksets where tasks in MTTs share periods demonstrate worst-case behavior for partitioning algorithms which do not group memory-sharing tasks because the timing of these tasks' executions will tend to maximize cache invalidations.

Relaxing the restriction that all tasks in an MTT must have the same period means that tasks in an MTT will not tend to execute at the same time as frequently. The cache effects

due to cache invalidations in this case will not be as severe as they would be if all periods remained the same in an MTT. The difference between the collective partitioning algorithms' performance for the MWHP (periods equal in MTT) and MWHU (periods randomized) can be seen in figures 6.6 and 6.8, respectively. We note that WFD performs around 15% better at low loads when periods are randomized, and that the IPC of all algorithms is clumped much closer together in this case.

The lack of improvement due to the LWFG scheme therefore appears to come from the naturally-better performance of the other algorithms rather than a fault its own. Regardless, we feel it is important to note that our partitioning algorithm does not improve execution efficiency in all cases.

Observation 4. *IPC performance in general is unpredictable on MWLP and MWHP distributions.* The error bars on the presented plots represent the standard deviation of all the tasksets at that point for that partitioning algorithm. It is obvious that the MWLP and MWHP distributions have much higher variation in IPC than do the other taskset distributions, particularly for those partitioning algorithms that partition tasks in order of decreasing utilization (FFD and BF). We attribute this to the fact that the utilizations of tasks in each MTT are not the same for these two distributions. Therefore, the efficacy of these partitioning algorithms is based on the random chance of two tasks in the same MTT receiving the same (or different) utilizations, rather than performing universally good or bad because all tasks in an MTT shared the same utilization.

Observation 5. *At high loads, the execution efficiency drops for all partitioning algorithms across all taskset distributions.* We mentioned in Section 6.1.5 that when a task is considered during partitioning using a given algorithm, if no core can be found with a current sum utilization small enough so that this task will fit (in terms of EDF utilization bounds, i.e. the cores's new sum utilization would be less than one), we place it on the core with the lowest sum utilization. We employ this strategy in an attempt to minimize task tardiness and deadline misses. Unfortunately, employing this strategy means that tasksets with total utilizations near to the total number of cores may abandon their previous partitioning scheme in favor of this greedy behavior. This is one possible reason for the decrease in efficiency.

Another reason is that a higher number of tasks in the system increases the total number of context switches, and increases the likelihood that a running task will be preempted by a higher-priority which just arrived. So, as the total number of tasks increases, the number of cache misses is likely to increase more than linearly.

Finally, for the LWFG algorithm, groups of tasks are more likely to be split up when loads are higher. Splitting memory-sharing tasks across more than one core increases general memory traffic, particularly in the form of cache invalidations. This, along with the other detriments listed above, contributes to the general drop observed in IPC at high loads.

Observation 6. *Execution efficiency is poor for tasksets that contain high numbers of tasks with large WSSes.* This trend is evident from the much steeper downward slopes of the tasksets with larger working set sizes (i.e. tasksets that begin with 'MWH'). This is not surprising considering that tasks with larger working set sizes tend to replace more cache lines with their own when they execute. However, it demonstrates the need for applications to minimize their effective working set sizes. Methods of accomplishing this include simply minimizing the amount of memory accessed, or changing memory references to maximize access locality (i.e. ensuring looping over 2-D matrices is done in the proper order – see Section 4.3).

6.2.2 LLC Miss Rate Results

Figures 6.9-6.16 and Tables 6.11-6.18 show the last level cache (LLC) miss rate $\left(\frac{\text{misses}}{\text{total loads}}\right)$ results for the studied taskset distributions. We present these results because while the IPC plots show that the LWFG partitioning algorithm improves overall execution efficiency, they do not demonstrate why it does so. Demonstrating that a decrease in cache misses occurs simultaneously with the increase in IPC suggests that the two are correlated.

Observation 1. *The LWFG partitioning scheme consistently causes a decrease in the LLC cache miss rate for all studied taskset distributions.* In some cases, the LWFG scheme decreases the cache miss rate by more than 30% against the next best partitioning scheme. There is only one instance where LWFG's cache miss rate is not lower than all the compared algorithms. This is for the MWHU distribution. For this distribution, the cache miss rates for all algorithms are much closer than they are for the other distributions, and the LWFG algorithm displays a higher cache miss rate for only one data point (and even then, this is well within the standard deviation of the measurements).

Observation 2. *Similarly to the IPC results, the LWFG LLC miss rate results do not display a marked improvement for the MWLU and MWHU distributions.* In fact, for these two distributions, the miss rates are extremely close for all the tested partitioning schemes. Moreover, they sustain much lower miss rates than do any of the other distributions tested. We hypothesize that this relatively small improvement of the LWFG algorithm is due to the fact that the qualities of the two distributions leave less room for improvement because they inherently incur fewer cache misses than the other distributions.

This could be because these two distributions are the only two for which tasks in the same MTT are allowed to have different periods. Having distinct periods for tasks which share memory means it is less likely for two tasks in the same MTT which are partitioned on different cores to execute at the same time. Therefore, the potentially harmful cache invalidations this would cause are much less likely than they are in the other taskset distributions.

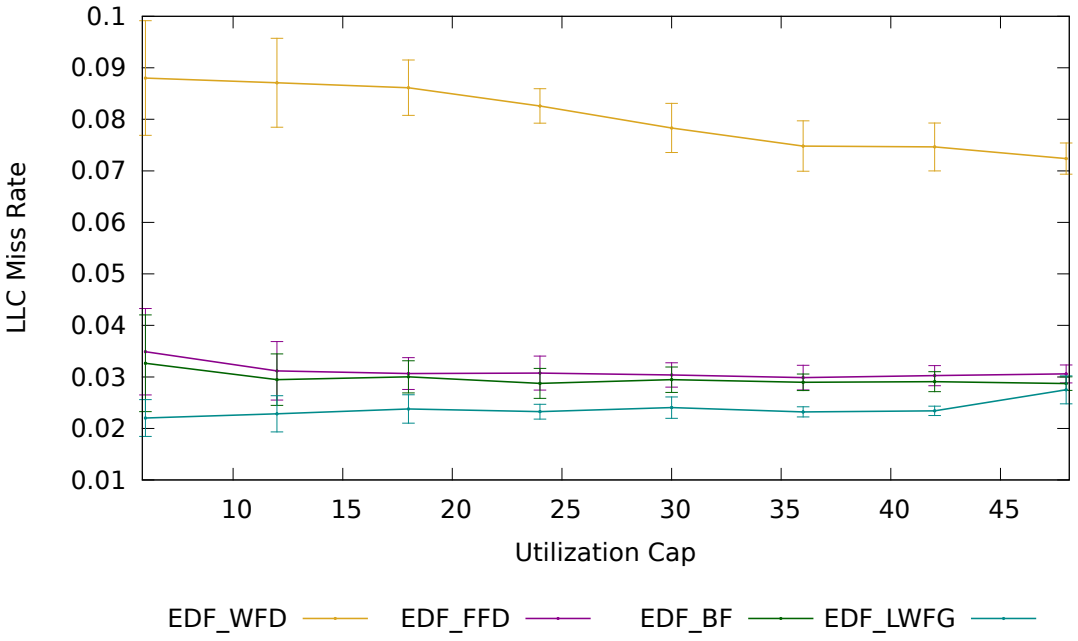


Figure 6.9: LLC Miss Rate for MLU Distribution

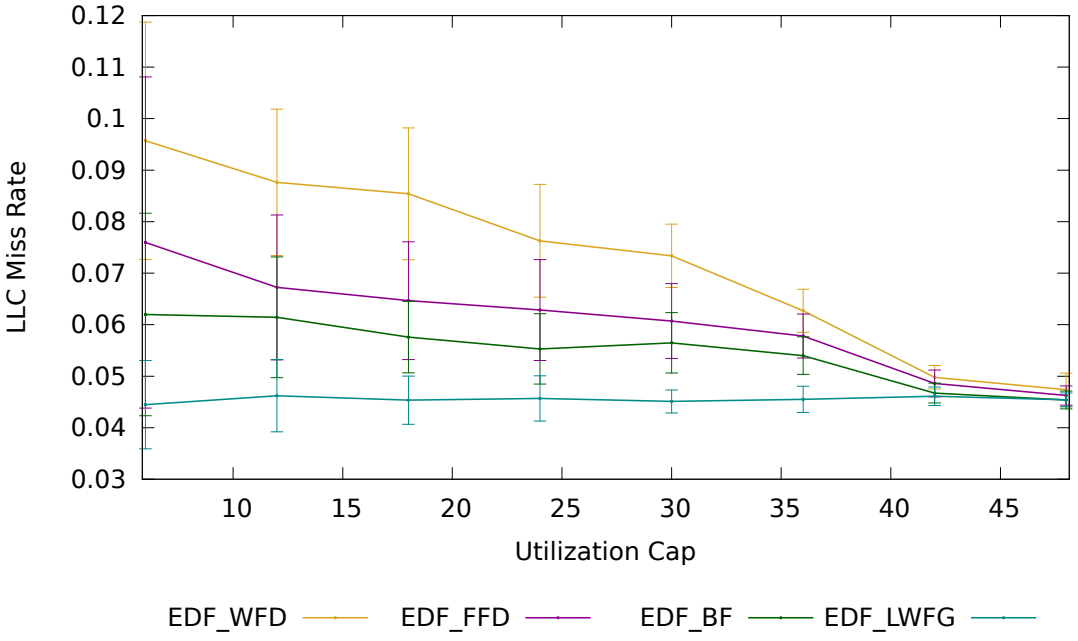


Figure 6.10: LLC Miss Rate for MMU Distribution

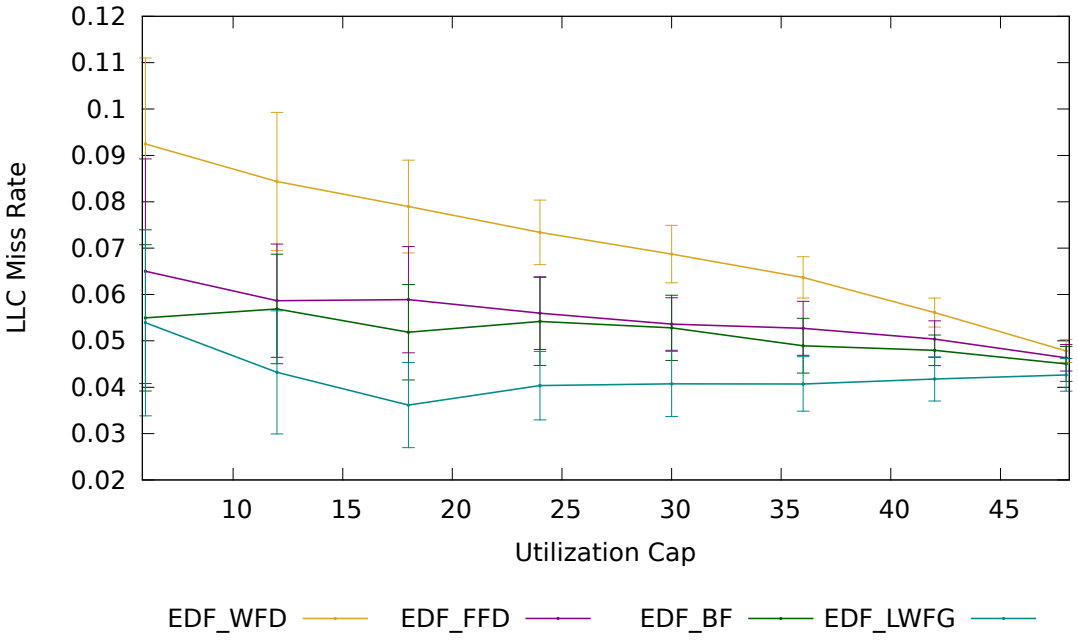


Figure 6.11: LLC Miss Rate for MWL Distribution

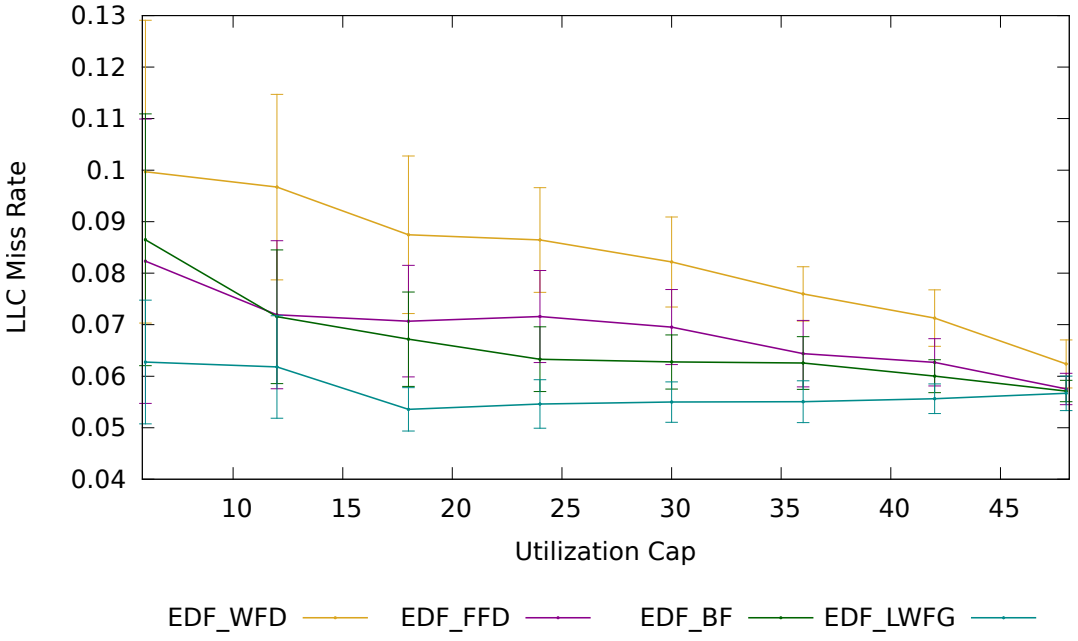


Figure 6.12: LLC Miss Rate for MWH Distribution

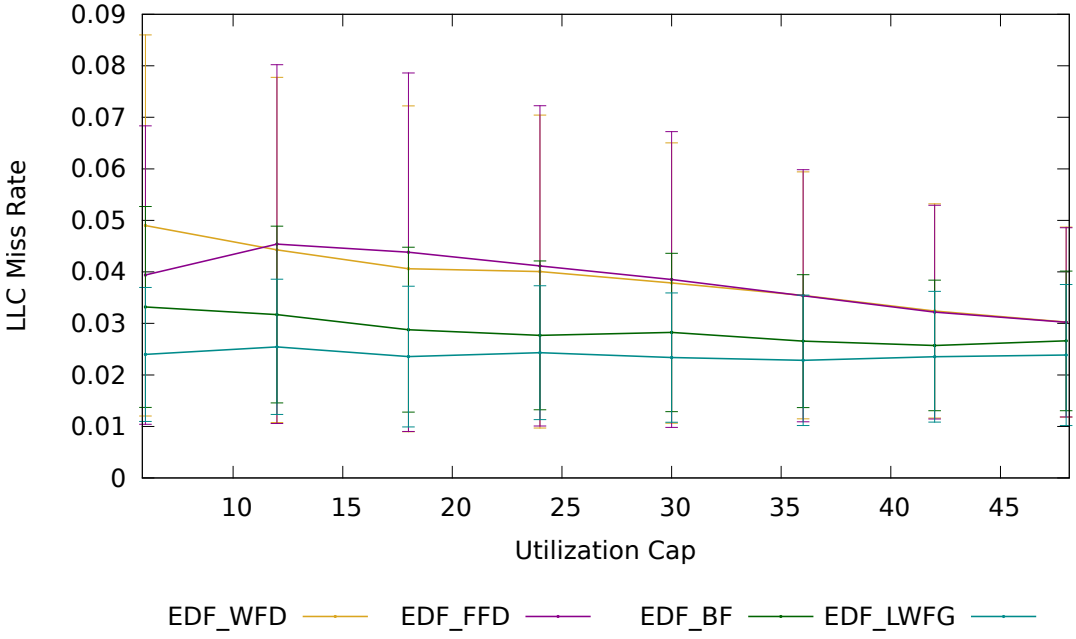


Figure 6.13: LLC Miss Rate for MWLP Distribution

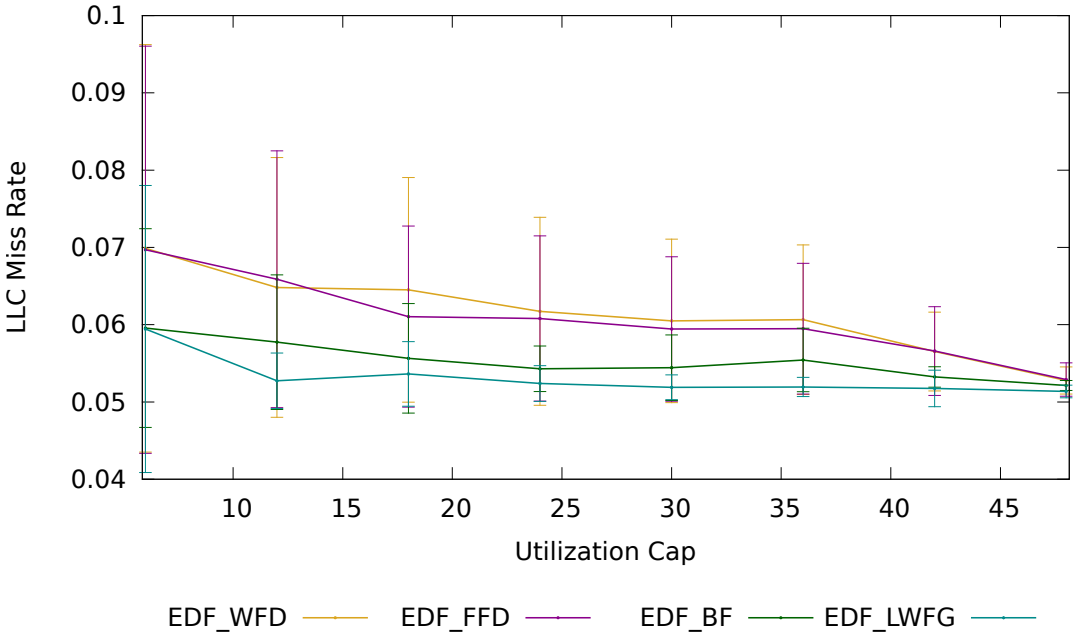


Figure 6.14: LLC Miss Rate for MWHP Distribution

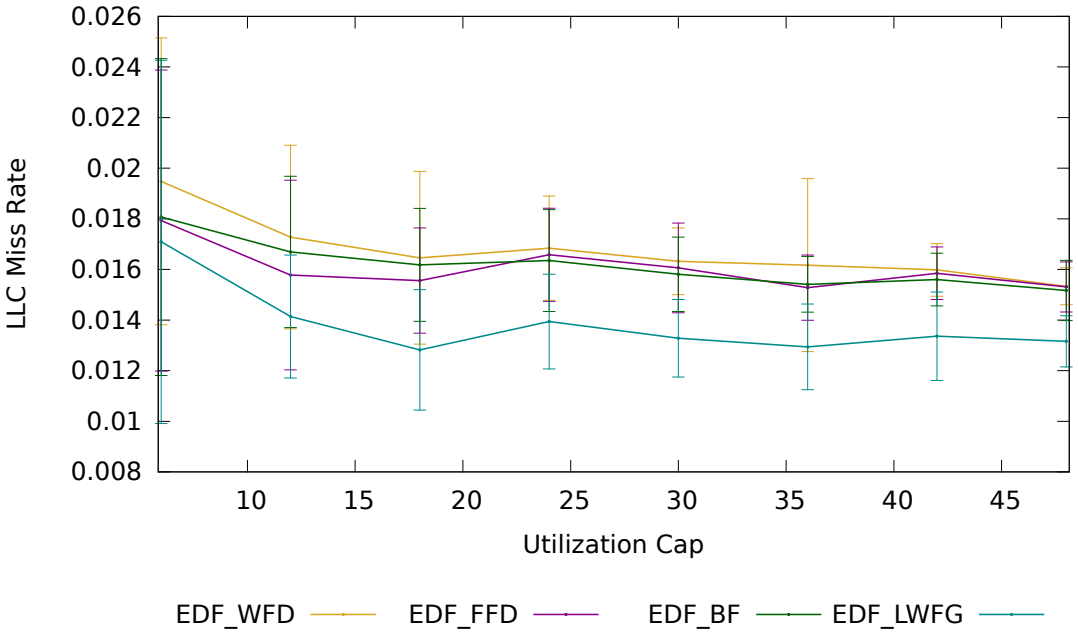


Figure 6.15: LLC Miss Rate for MWLU Distribution

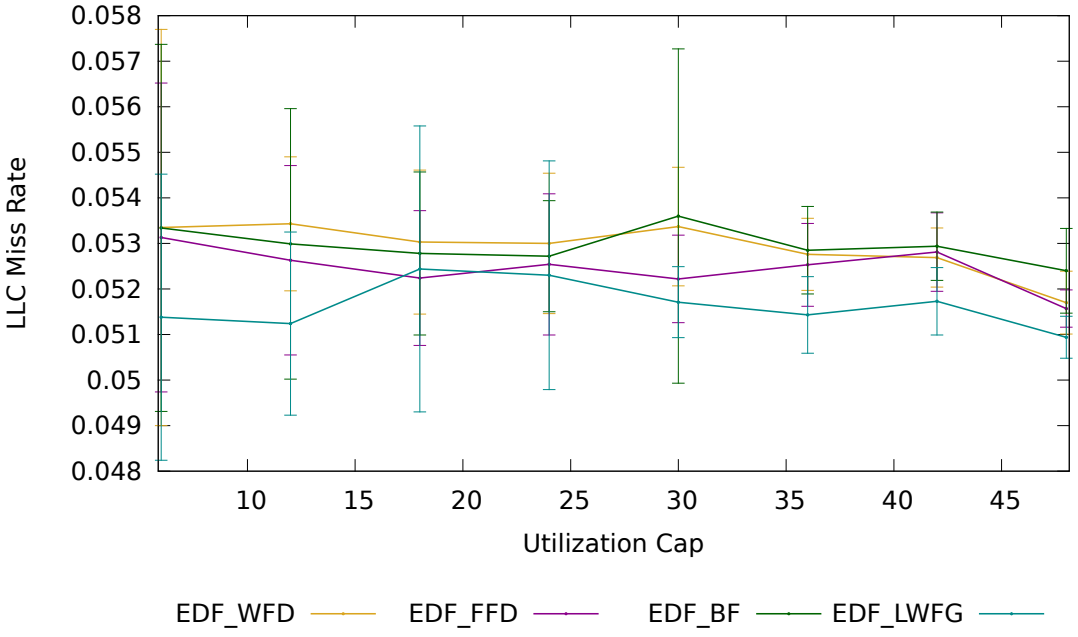


Figure 6.16: LLC Miss Rate for MWHU Distribution

Table 6.11: LLC Miss Rate for MLU Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev
6	0.08801	0.01113	0.03489	0.00840	0.03264	0.00939	0.02203	0.00357
12	0.08709	0.00863	0.03118	0.00568	0.02948	0.00499	0.02285	0.00351
18	0.08614	0.00537	0.03064	0.00309	0.03003	0.00312	0.02377	0.00276
24	0.08260	0.00335	0.03075	0.00330	0.02875	0.00290	0.02325	0.00144
30	0.07832	0.00477	0.03038	0.00235	0.02946	0.00246	0.02404	0.00208
36	0.07481	0.00489	0.02987	0.00239	0.02896	0.00161	0.02321	0.00098
42	0.07464	0.00465	0.03025	0.00195	0.02909	0.00194	0.02341	0.00091
48	0.07237	0.00303	0.03058	0.00173	0.02871	0.00132	0.02749	0.00270

Table 6.12: LLC Miss Rate for MMU Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev
6	0.09570	0.02304	0.07596	0.03214	0.06199	0.01965	0.04446	0.00857
12	0.08764	0.01420	0.06724	0.01407	0.06143	0.01172	0.04621	0.00701
18	0.08541	0.01279	0.06466	0.01144	0.05759	0.00693	0.04534	0.00469
24	0.07628	0.01094	0.06285	0.00979	0.05530	0.00683	0.04569	0.00439
30	0.07336	0.00615	0.06070	0.00726	0.05648	0.00586	0.04510	0.00223
36	0.06270	0.00418	0.05779	0.00427	0.05399	0.00365	0.04551	0.00255
42	0.04978	0.00229	0.04859	0.00261	0.04673	0.00192	0.04611	0.00178
48	0.04737	0.00322	0.04625	0.00187	0.04538	0.00173	0.04540	0.00135

Table 6.13: LLC Miss Rate for MWL Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev
6	0.09246	0.01854	0.06501	0.02423	0.05496	0.01577	0.05391	0.02007
12	0.08437	0.01490	0.05865	0.01221	0.05688	0.01182	0.04321	0.01328
18	0.07897	0.01002	0.05889	0.01146	0.05186	0.01030	0.03614	0.00918
24	0.07340	0.00698	0.05596	0.00784	0.05420	0.00951	0.04034	0.00738
30	0.06873	0.00619	0.05362	0.00567	0.05281	0.00702	0.04071	0.00700
36	0.06369	0.00446	0.05270	0.00583	0.04895	0.00591	0.04070	0.00587
42	0.05610	0.00314	0.05038	0.00394	0.04797	0.00330	0.04178	0.00477
48	0.04782	0.00250	0.04635	0.00286	0.04503	0.00376	0.04266	0.00352

Table 6.14: LLC Miss Rate for MWH Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev
6	0.09970	0.02938	0.08232	0.02760	0.08649	0.02444	0.06275	0.01201
12	0.09671	0.01801	0.07192	0.01437	0.07154	0.01297	0.06180	0.00995
18	0.08746	0.01529	0.07069	0.01083	0.06718	0.00917	0.05357	0.00421
24	0.08644	0.01016	0.07159	0.00894	0.06330	0.00629	0.05460	0.00471
30	0.08218	0.00874	0.06953	0.00728	0.06276	0.00525	0.05498	0.00393
36	0.07599	0.00527	0.06438	0.00645	0.06256	0.00512	0.05504	0.00406
42	0.07127	0.00548	0.06269	0.00459	0.06001	0.00320	0.05562	0.00288
48	0.06238	0.00465	0.05752	0.00304	0.05712	0.00207	0.05668	0.00335

Table 6.15: LLC Miss Rate for MWLP Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev
6	0.04900	0.03699	0.03939	0.02896	0.03318	0.01950	0.02397	0.01302
12	0.04426	0.03351	0.04539	0.03483	0.03172	0.01715	0.02545	0.01312
18	0.04060	0.03163	0.04381	0.03480	0.02878	0.01601	0.02356	0.01367
24	0.04006	0.03038	0.04116	0.03108	0.02767	0.01445	0.02433	0.01299
30	0.03785	0.02721	0.03853	0.02871	0.02824	0.01537	0.02337	0.01255
36	0.03544	0.02397	0.03537	0.02448	0.02657	0.01290	0.02284	0.01265
42	0.03241	0.02079	0.03220	0.02074	0.02572	0.01267	0.02353	0.01268
48	0.03026	0.01844	0.03021	0.01836	0.02661	0.01355	0.02386	0.01370

Table 6.16: LLC Miss Rate for MWHP Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev
6	0.06988	0.02637	0.06970	0.02634	0.05956	0.01286	0.05944	0.01857
12	0.06482	0.01680	0.06588	0.01662	0.05774	0.00871	0.05274	0.00360
18	0.06451	0.01453	0.06104	0.01172	0.05565	0.00709	0.05362	0.00418
24	0.06173	0.01217	0.06080	0.01069	0.05429	0.00295	0.05239	0.00231
30	0.06050	0.01058	0.05944	0.00934	0.05443	0.00424	0.05190	0.00161
36	0.06065	0.00968	0.05949	0.00846	0.05543	0.00411	0.05194	0.00125
42	0.05653	0.00510	0.05659	0.00574	0.05324	0.00132	0.05175	0.00237
48	0.05278	0.00176	0.05291	0.00216	0.05213	0.00064	0.05136	0.00082

Table 6.17: LLC Miss Rate for MWLU Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev
6	0.01948	0.00567	0.01793	0.00595	0.01807	0.00626	0.01709	0.00718
12	0.01728	0.00363	0.01578	0.00375	0.01669	0.00299	0.01414	0.00243
18	0.01646	0.00341	0.01556	0.00208	0.01618	0.00223	0.01282	0.00238
24	0.01684	0.00206	0.01658	0.00184	0.01635	0.00201	0.01394	0.00187
30	0.01632	0.00132	0.01606	0.00177	0.01581	0.00147	0.01328	0.00153
36	0.01617	0.00342	0.01528	0.00129	0.01541	0.00110	0.01294	0.00169
42	0.01598	0.00104	0.01585	0.00104	0.01560	0.00104	0.01336	0.00175
48	0.01533	0.00073	0.01531	0.00099	0.01517	0.00119	0.01316	0.00101

Table 6.18: LLC Miss Rate for MWHU Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev	Miss rate	Stdev
6	0.05335	0.00435	0.05313	0.00339	0.05334	0.00403	0.05138	0.00314
12	0.05343	0.00147	0.05263	0.00208	0.05299	0.00297	0.05124	0.00201
18	0.05303	0.00158	0.05224	0.00148	0.05278	0.00179	0.05244	0.00314
24	0.05300	0.00154	0.05254	0.00155	0.05272	0.00122	0.05230	0.00251
30	0.05337	0.00130	0.05222	0.00096	0.05360	0.00367	0.05171	0.00078
36	0.05276	0.00079	0.05253	0.00091	0.05285	0.00096	0.05143	0.00084
42	0.05269	0.00065	0.05281	0.00086	0.05294	0.00075	0.05173	0.00074
48	0.05170	0.00069	0.05157	0.00041	0.05240	0.00093	0.05094	0.00046

6.2.3 Deadline Satisfaction Ratio Results

We include deadline satisfaction ratio (DSR) plots in Figures 6.17-6.24 (Full numerical results can be found in Tables **??**-6.26) . These plots are included to demonstrate that we do not sacrifice meeting deadlines for increased efficiency and IPC. DSR is calculated as the total number of deadlines met over the total number of real-time task instances. The maximum (and optimal) value for DSR is therefore 1.0.

Observation 1. *The DSR for the LWFG scheme is generally comparable to the other partitioning schemes.* While the DSR of the LWFG scheme dips below 1.0 at high loads, it is never considerably worse than the other three partitioning heuristics we compared it against. We attribute these deadline misses primarily to infeasibly-partitioned tasksets. If a 'fall-back' partitioning approach were taken, where another scheme were used if LWFG failed to feasible partition a taskset, we assert that these deadline misses would be minimized.

Observation 2. *The DSR of the BF scheme outperforms all other partitioning algorithms for tasksets with large amounts of memory.* LWFG's DSR is consistently higher than that of WFD, and is generally comparable to that of FFD. However, the BF partitioning scheme appears to be the best out of those studied at meeting deadlines. While LWFG's improvement in execution efficiency helps it avoid some missed deadlines it is not enough to overcome its higher number of infeasible tasksets. Again, this leads us to suggest using LWFG as a primary partitioning scheme, and using a 'fallback' scheme in the cases where LWFG fails to feasibly partition a taskset.

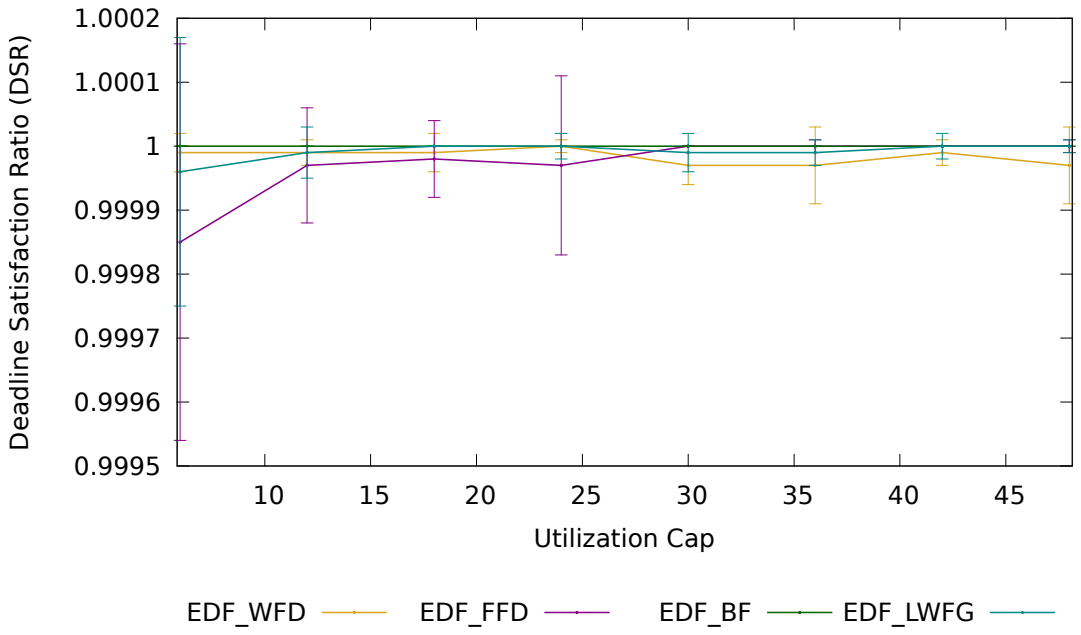


Figure 6.17: DSR for MLU Distribution

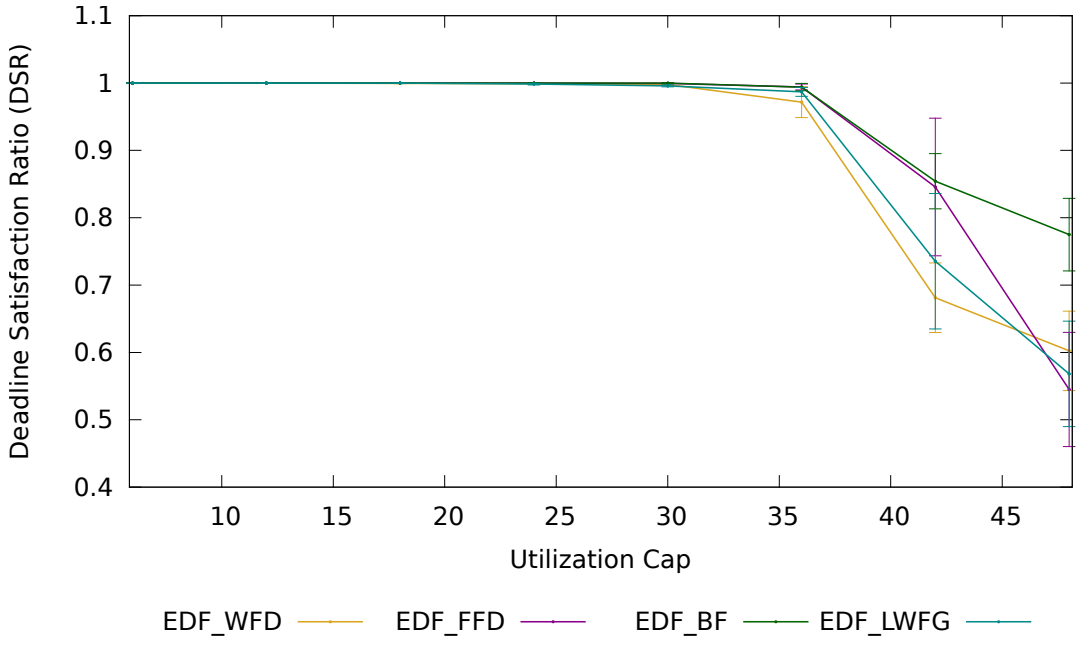


Figure 6.18: DSR for MMU Distribution

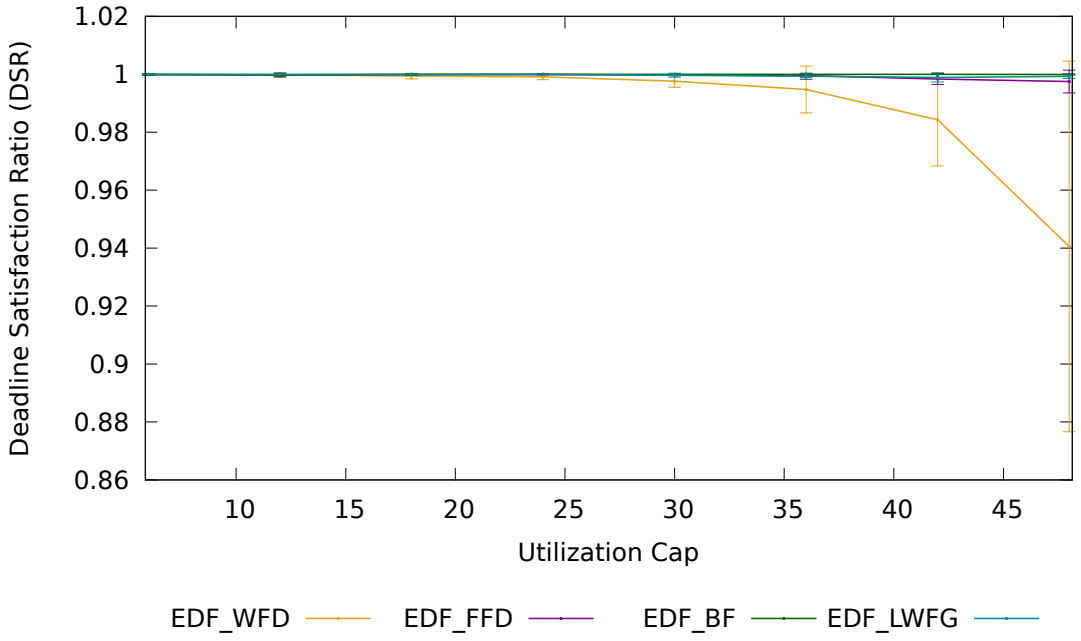


Figure 6.19: DSR for MWL Distribution

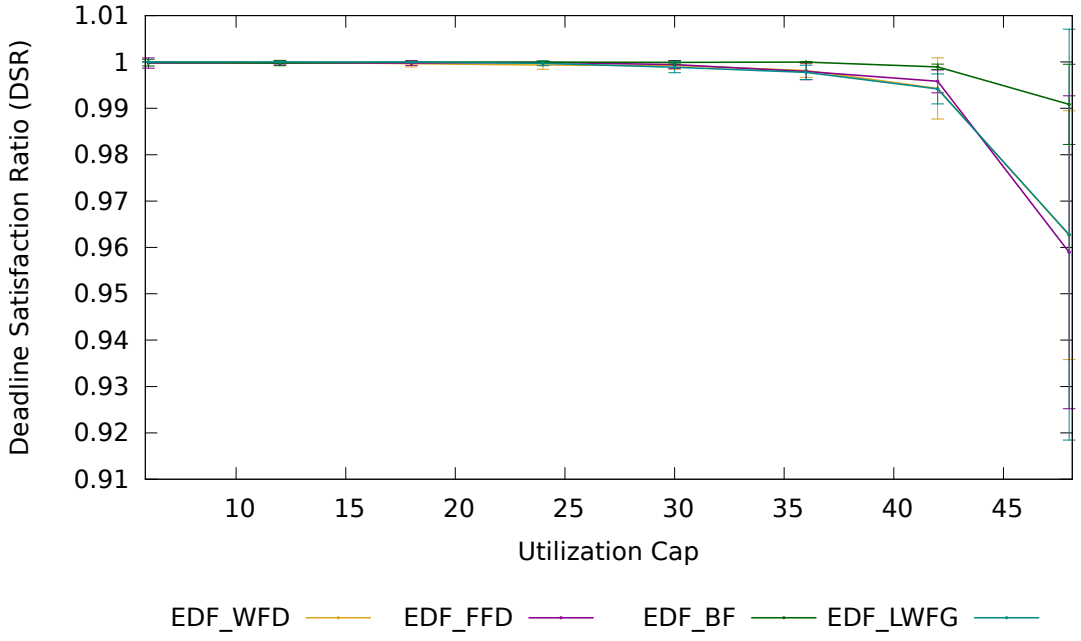


Figure 6.20: DSR for MWH Distribution

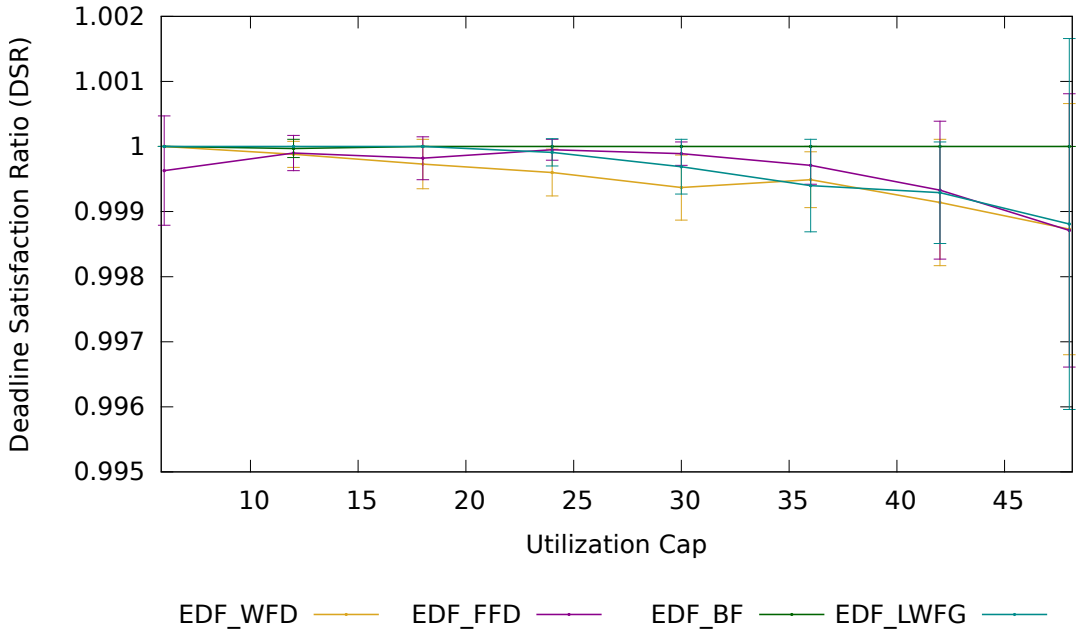


Figure 6.21: DSR for MWLP Distribution

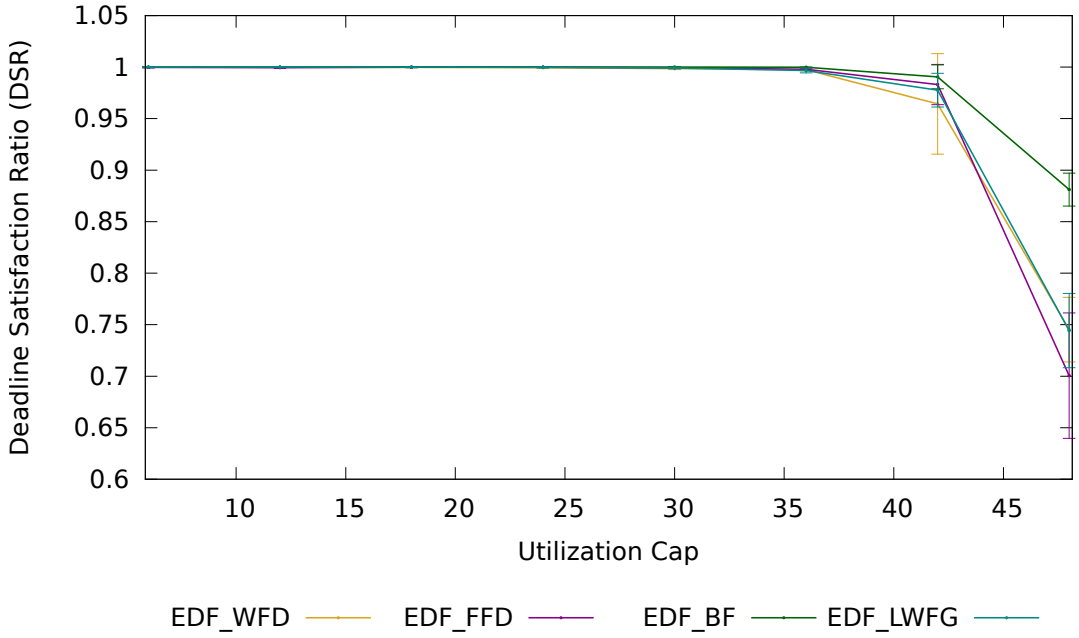


Figure 6.22: DSR for MWHP Distribution

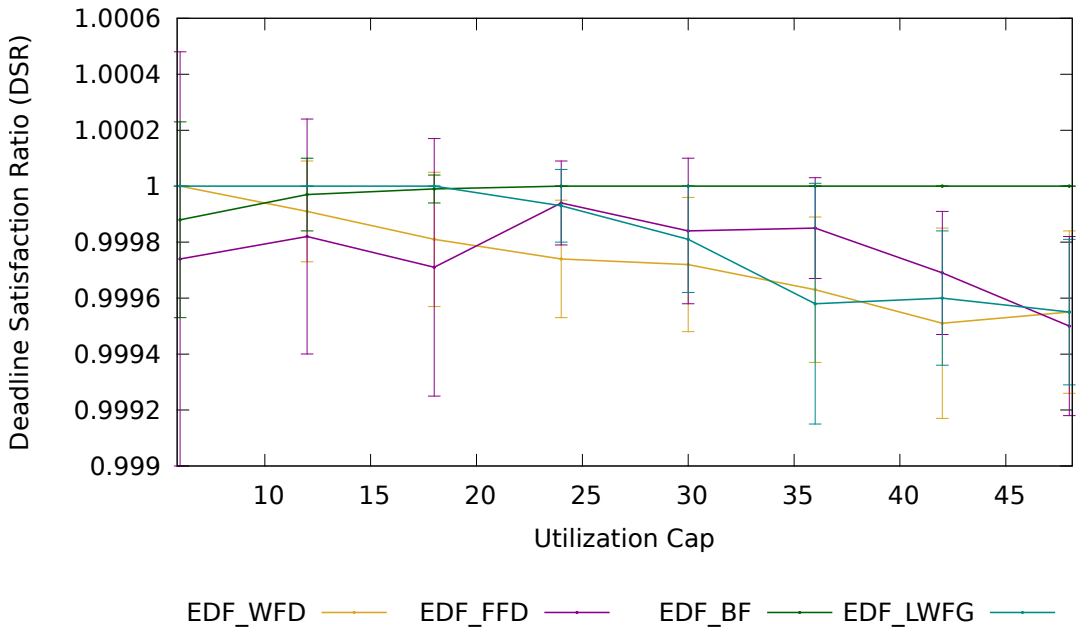


Figure 6.23: DSR for MWLU Distribution

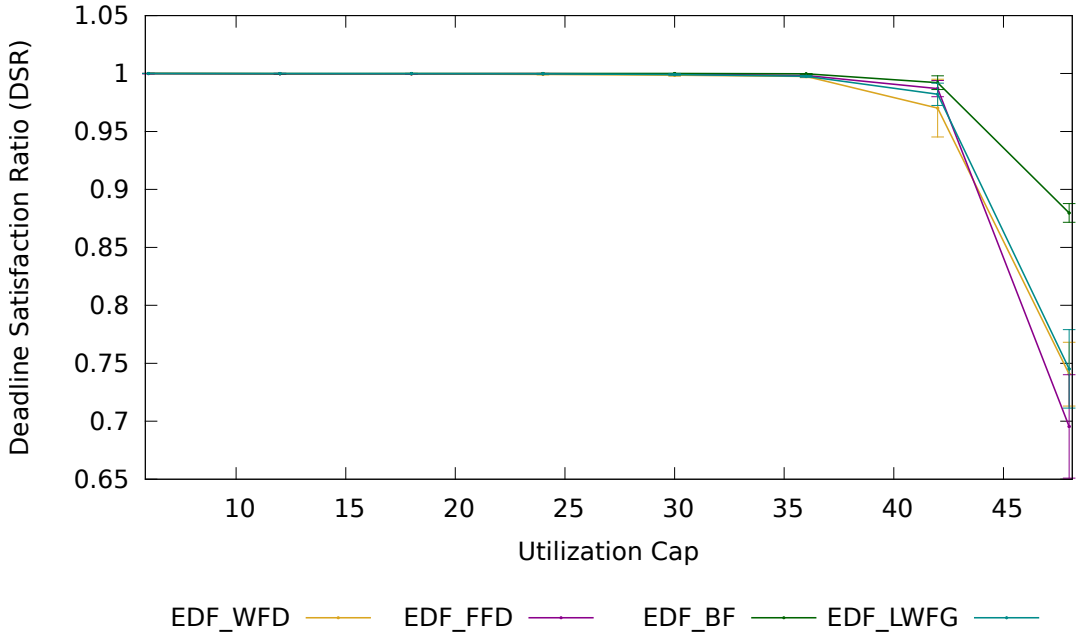


Figure 6.24: DSR for MWHU Distribution

Table 6.19: DSR for MWL Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	DSR	Stdev	DSR	Stdev	DSR	Stdev	DSR	Stdev
6	0.99999	0.00003	0.99985	0.00031	1.00000	0.00000	0.99996	0.00021
12	0.99999	0.00002	0.99997	0.00009	1.00000	0.00000	0.99999	0.00004
18	0.99999	0.00003	0.99998	0.00006	1.00000	0.00000	1.00000	0.00000
24	1.00000	0.00001	0.99997	0.00014	1.00000	0.00000	1.00000	0.00002
30	0.99997	0.00003	1.00000	0.00000	1.00000	0.00000	0.99999	0.00003
36	0.99997	0.00006	1.00000	0.00001	1.00000	0.00000	0.99999	0.00002
42	0.99999	0.00002	1.00000	0.00000	1.00000	0.00000	1.00000	0.00002
48	0.99997	0.00006	1.00000	0.00001	1.00000	0.00000	1.00000	0.00001

Table 6.20: DSR for MMU Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	DSR	Stdev	DSR	Stdev	DSR	Stdev	DSR	Stdev
6	1.00000	0.00000	0.99979	0.00054	0.99998	0.00012	1.00000	0.00000
12	0.99991	0.00024	0.99993	0.00019	0.99999	0.00007	1.00000	0.00000
18	0.99955	0.00052	0.99999	0.00008	1.00000	0.00000	0.99988	0.00024
24	0.99911	0.00093	0.99999	0.00005	0.99996	0.00020	0.99850	0.00125
30	0.99785	0.00100	0.99943	0.00062	1.00000	0.00000	0.99585	0.00144
36	0.97169	0.02292	0.99412	0.00434	0.99400	0.00544	0.98685	0.00664
42	0.68131	0.05165	0.84569	0.10211	0.85425	0.04093	0.73541	0.10042
48	0.60234	0.05888	0.54500	0.08476	0.77499	0.05385	0.56820	0.07823

Table 6.21: DSR for MWL Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	DSR	Stdev	DSR	Stdev	DSR	Stdev	DSR	Stdev
6	1.00000	0.00000	0.99994	0.00022	0.99989	0.00032	1.00000	0.00000
12	0.99989	0.00033	0.99974	0.00067	0.99981	0.00062	1.00000	0.00000
18	0.99938	0.00099	0.99992	0.00020	0.99993	0.00026	1.00000	0.00000
24	0.99909	0.00096	0.99992	0.00016	1.00000	0.00000	0.99999	0.00006
30	0.99761	0.00208	0.99978	0.00030	0.99997	0.00013	0.99967	0.00066
36	0.99474	0.00807	0.99953	0.00072	0.99999	0.00003	0.99932	0.00111
42	0.98432	0.01594	0.99835	0.00187	1.00000	0.00000	0.99891	0.00158
48	0.94060	0.06395	0.99747	0.00393	0.99992	0.00021	0.99927	0.00072

Table 6.22: DSR for MWH Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	DSR	Stdev	DSR	Stdev	DSR	Stdev	DSR	Stdev
6	1.00000	0.00000	0.99977	0.00112	0.99986	0.00072	1.00000	0.00000
12	0.99997	0.00014	0.99979	0.00041	0.99978	0.00058	1.00000	0.00000
18	0.99962	0.00068	0.99978	0.00051	0.99998	0.00008	0.99996	0.00011
24	0.99935	0.00091	0.99991	0.00018	0.99998	0.00008	0.99972	0.00051
30	0.99923	0.00077	0.99943	0.00088	0.99989	0.00031	0.99886	0.00118
36	0.99816	0.00147	0.99797	0.00181	0.99998	0.00008	0.99774	0.00155
42	0.99429	0.00659	0.99582	0.00248	0.99894	0.00058	0.99420	0.00323
48	0.96265	0.02684	0.95895	0.03375	0.99085	0.00865	0.96275	0.04431

Table 6.23: DSR for MWLP Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	DSR	Stdev	DSR	Stdev	DSR	Stdev	DSR	Stdev
6	1.00000	0.00000	0.99963	0.00084	1.00000	0.00000	1.00000	0.00000
12	0.99988	0.00020	0.99990	0.00027	0.99997	0.00014	1.00000	0.00000
18	0.99973	0.00038	0.99982	0.00033	1.00000	0.00000	1.00000	0.00000
24	0.99960	0.00036	0.99995	0.00016	1.00000	0.00000	0.99991	0.00021
30	0.99937	0.00050	0.99989	0.00018	1.00000	0.00000	0.99969	0.00042
36	0.99949	0.00043	0.99971	0.00029	1.00000	0.00000	0.99940	0.00071
42	0.99914	0.00097	0.99933	0.00106	1.00000	0.00000	0.99929	0.00078
48	0.99873	0.00193	0.99871	0.00210	1.00000	0.00000	0.99881	0.00285

Table 6.24: DSR for MWHP Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	DSR	Stdev	DSR	Stdev	DSR	Stdev	DSR	Stdev
6	1.00000	0.00000	0.99966	0.00055	0.99994	0.00019	1.00000	0.00000
12	0.99994	0.00013	0.99947	0.00086	1.00000	0.00000	1.00000	0.00000
18	0.99978	0.00032	0.99983	0.00039	1.00000	0.00000	1.00000	0.00000
24	0.99941	0.00042	0.99983	0.00047	1.00000	0.00000	0.99972	0.00044
30	0.99853	0.00085	0.99945	0.00046	1.00000	0.00000	0.99909	0.00076
36	0.99743	0.00113	0.99789	0.00121	0.99988	0.00018	0.99672	0.00234
42	0.96423	0.04875	0.98305	0.01949	0.99057	0.01151	0.97759	0.01635
48	0.74521	0.03139	0.70058	0.06090	0.88114	0.01594	0.74426	0.03603

Table 6.25: DSR for MWLU Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	DSR	Stdev	DSR	Stdev	DSR	Stdev	DSR	Stdev
6	1.00000	0.00000	0.99974	0.00074	0.99988	0.00035	1.00000	0.00000
12	0.99991	0.00018	0.99982	0.00042	0.99997	0.00013	1.00000	0.00000
18	0.99981	0.00024	0.99971	0.00046	0.99999	0.00005	1.00000	0.00000
24	0.99974	0.00021	0.99994	0.00015	1.00000	0.00000	0.99993	0.00013
30	0.99972	0.00024	0.99984	0.00026	1.00000	0.00000	0.99981	0.00019
36	0.99963	0.00026	0.99985	0.00018	1.00000	0.00000	0.99958	0.00043
42	0.99951	0.00034	0.99969	0.00022	1.00000	0.00000	0.99960	0.00024
48	0.99955	0.00029	0.99950	0.00032	1.00000	0.00000	0.99955	0.00026

Table 6.26: DSR for MWHU Distribution

	EDF_WFD		EDF_FFD		EDF_BF		EDF_LWFG	
Load	DSR	Stdev	DSR	Stdev	DSR	Stdev	DSR	Stdev
6	1.00000	0.00000	0.99989	0.00027	1.00000	0.00000	1.00000	0.00000
12	0.99994	0.00013	0.99984	0.00038	0.99998	0.00010	1.00000	0.00000
18	0.99972	0.00030	0.99982	0.00031	1.00000	0.00000	0.99999	0.00005
24	0.99930	0.00041	0.99992	0.00016	1.00000	0.00000	0.99978	0.00026
30	0.99860	0.00081	0.99948	0.00051	1.00000	0.00000	0.99918	0.00056
36	0.99748	0.00100	0.99832	0.00103	0.99983	0.00015	0.99763	0.00075
42	0.97018	0.02487	0.98702	0.00699	0.99217	0.00594	0.98215	0.00963
48	0.74070	0.02752	0.69554	0.04458	0.87974	0.00810	0.74518	0.03386

6.3 Further Exploration

Though the previous study and results show that our partitioning algorithm generally improves execution efficiency, they do not demonstrate improvement on any real-time metrics. This is in part because the vast majority of the tasksets are feasibly-partitioned (therefore, they always achieve a DSR above one), and in part because those tasksets which are infeasible are partitioned poorly due to the partitioning scheme's attempts to place the considered tasks on the core with the smallest current sum utilization.

As described in Section 4.2, it is often impossible, and nearly always difficult and/or cost-prohibitive to determine or prove absolute WCET values for real-time applications. Many real-time systems therefore rely on experimentally-measured WCETs, which may not be truly worse-case.

For these two reasons, we conducted additional tests to show that the LWFG partitioning algorithm is capable of improving real-time metrics in the face of WCET under-estimation and overload conditions. For these tests, we vary the ratio between the actual WCET and the WCET reported to the scheduling system from 1.0 to 2.0. Unlike before, the maximum taskset utilization is fixed at 95% of the theoretical hardware bound. We have focused this exploration on the MWL taskset distribution. We chose the MWL distribution because it seems to be fairly indicative of the remainder of the tasksets, and choosing one distribution allowed us to focus our time on understanding one distribution's behavior in-depth rather than that of many shallowly.

Figures 6.25 and 6.26 present the DSR and per-taskset maximum task tardiness metrics. We see from these plots that while the Baruah-Fisher partitioning algorithm manages to obtain slightly higher DSR, LWFG clearly does a much better job of bounding task tardiness. This is because LWFG misses deadlines by smaller amounts of time than does BF.

Figure 6.27 presents a histogram of task tardiness of the MWL taskset distribution with an actual/reported WCET ratio of 1.6. If a task meets its deadline, its tardiness is recorded as zero. Although BF meets more deadlines than LWFG at a load of 1.6 (see Figure 6.25), many more of LWFG's tasks have tardiness closer to zero. These results show that the LWFG partitioning scheme not only minimizes cache misses, but decreases task tardiness.

Figure 6.25: DSR for MWL Distribution with Varied Actual/Reported WCET

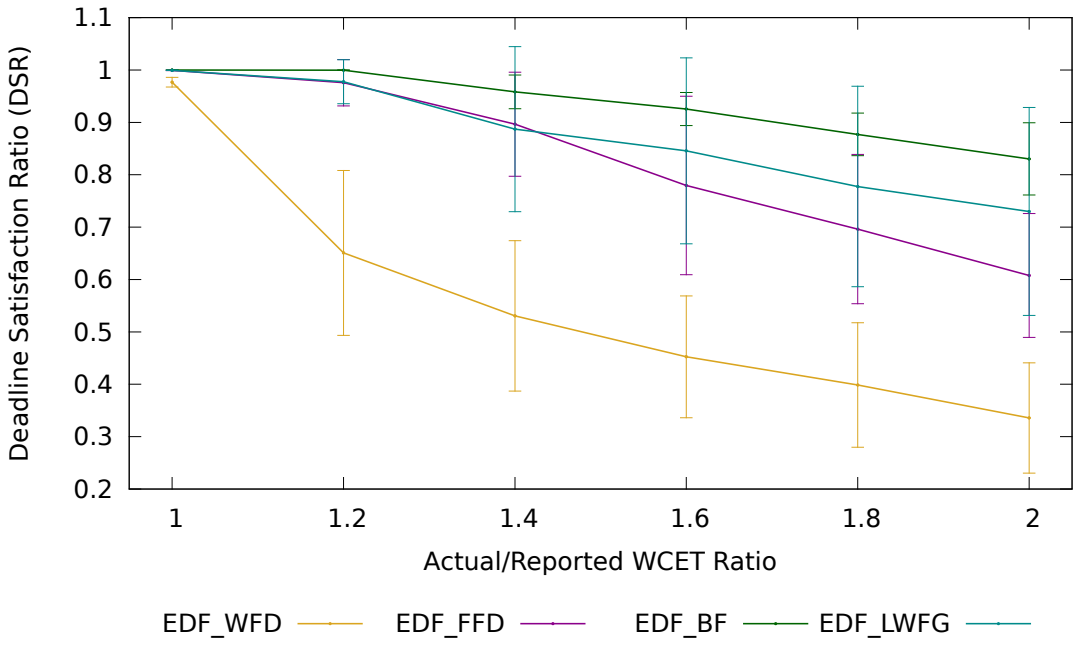
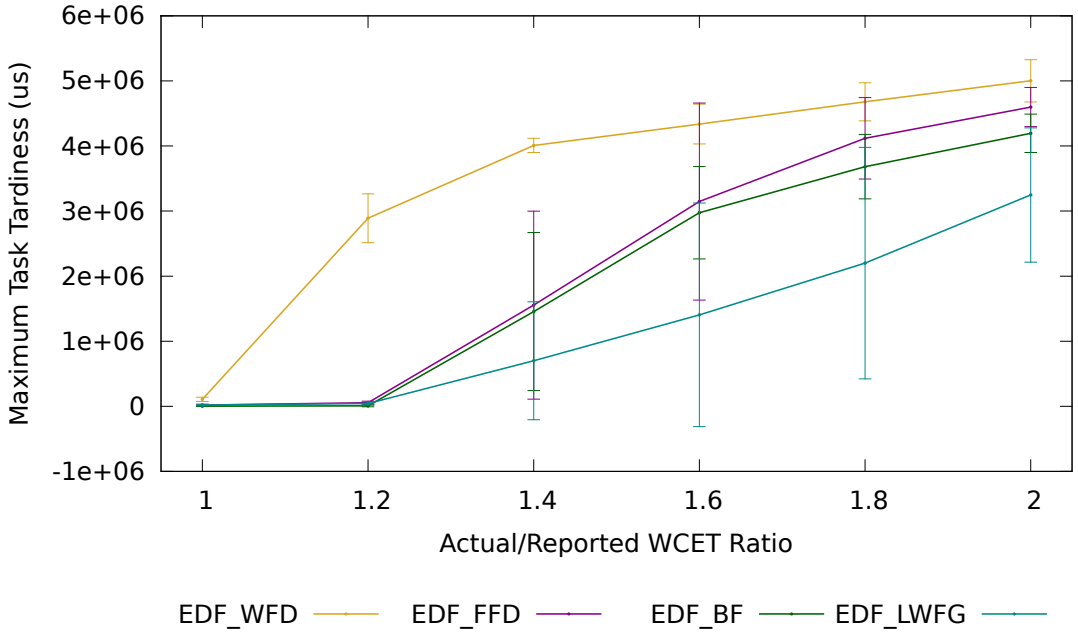


Figure 6.26: Tardiness for MWL Distribution with Varied Actual/Reported WCET



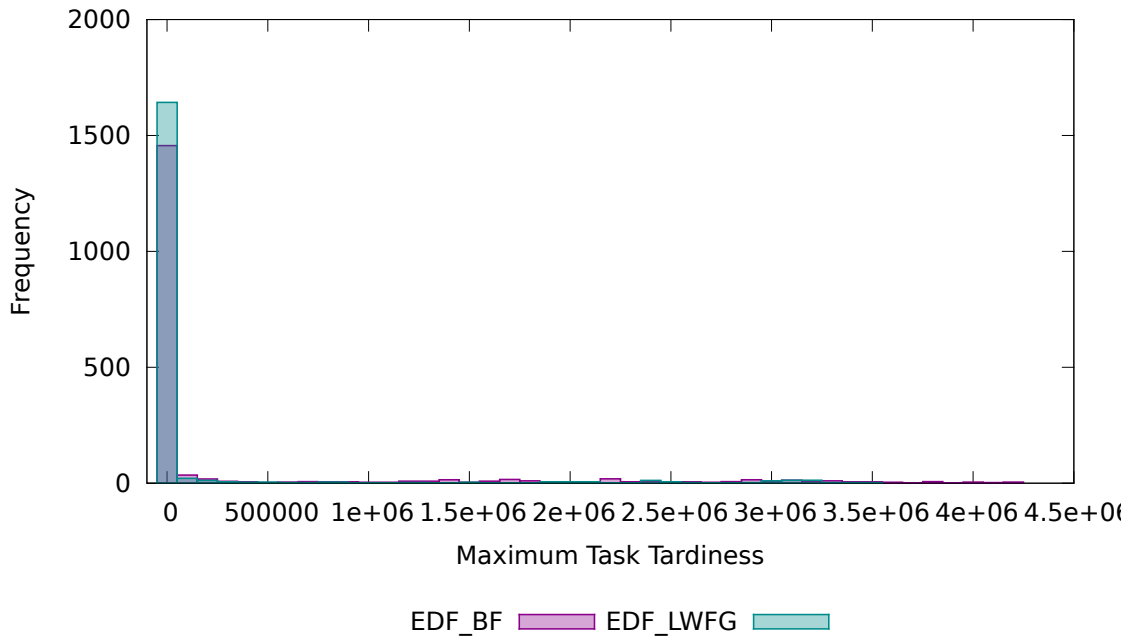


Figure 6.27: Tardiness Histogram for MWL Distribution with 1.6 Actual/Reported WCET Ratio

6.4 Summary

In this chapter, we presented an experimental evaluation of the LWFG partitioning algorithm: our methods, results, and further exploration. We find that LWFG improves IPC (execution efficiency), and is effective at reducing the average maximum task tardiness when a system is in overload conditions (potentially due to WCET underestimation). LWFG does not out-perform all partitioning algorithms in all areas, but it does so in many cases, and certainly achieves our goal of developing a cache-aware partitioning strategy.

Chapter 7

Conclusions

In this thesis, we addressed the problem of cache-aware real-time scheduling algorithms, focusing on partitioning algorithms. There has been no previous research in this area. We first developed the principles behind "cache-awareness", then applied them to the creation of a cache-aware partitioning algorithm, which we termed LWFG (Least Working set size First, Grouping).

Furthermore, we evaluated the LWFG partitioning algorithm in comparison to three other commonly-used partitioning heuristics: WFD (worst-fit, decreasing utilization order), FFD (first-fit, decreasing utilization order), and BF (first-fit, increasing relative deadline order). Our evaluation included eight taskset distributions, covering a variety of working set sizes, utilizations, periods, numbers of memory-sharing tasks, and whether periods and utilizations were per-task or per-group of memory-sharing tasks. We conducted this experiment on a 48-core AMD *Magny-Cours* platform.

Our experimental evaluation of the LWFG partitioning algorithm shows that it is effective at reducing cache misses for a variety of taskset distributions, and improves execution efficiency (as measured by IPC) by up to 15%. Furthermore, we show that the LWFG partitioning algorithm is capable of decreasing the average maximum taskset tardiness metric by up to 60% in some cases.

We draw several conclusions based on our experiences and observations.

Studying cache-aware algorithms is an important area of research for computing systems in general, and especially so for those with high core counts and deep, hierarchical caches. This was made apparent by the drastic differences in the number of cache misses and general execution efficiency that different partitioning algorithms effected. The extent and directions of what we believe to be the most immediate future work are covered in the following chapter.

We support the partitioning and cache-related conclusions posited by Dellinger in his thesis [29]. Namely, that cache-miss overheads are an important consideration when working with real-time systems in practice, and that task partitioning is a viable method for improving the scalability of real-time applications to many-core platforms.

The LWFG partitioning algorithm reduces the number of cache misses suffered by the tested taskset distributions. In most cases, this reduction led to an increase in IPC and/or a decrease in mean maximum tardiness. We therefore conclude that the LWFG partitioning algorithm is effective in improving real-time performance for applications in which tasks share memory with each other.

Cache-awareness in general is application-, platform-, and environment-specific. Care should be taken when adapting any cache-aware algorithm to a new context. The results we obtained for the LWFG partitioning algorithm are somewhat specific to applications whose characteristics match those of the applicable studied taskset distributions. As with any cache-aware algorithm, it would be purely wishful thinking to assume that the LWFG scheme would improve IPC or reduce tardiness for all applications in all situations. As evidenced by the results from a few of our taskset distributions, this is not even true for the subset of possible behaviors tested here. Therefore, as always, these techniques should be applied to a real-time system only after testing their effects for the specific application, platform, and environment in question.

The area of cache-aware real-time partitioning algorithms is an important area for future research. While the research contained in this thesis has answered several questions, it has raised far more. The amount of improvement seen from a combination of relatively simple heuristics (compared to other current non-cache-aware partitioning schemes) suggests that there is yet more improvement to be seen by building upon both the cache-aware principles and goals outlined in this thesis, as well as the algorithm itself. We hope our research is not seen as the final answer in this area, but rather as a starting point which will spur others to greater insights and improvements in the field of cache-aware real-time partitioning algorithms.

7.1 Contributions

To summarize, the major contributions of this thesis research are:

1. The creation of a set of goals obtaining cache-awareness in a scheduling algorithm and methods for applying them.

2. The application of these goals to real-time partitioning and the development of the LWFG partitioning algorithm with them in mind.

3. The experimental evaluation of LWFG in comparison to other popular partitioning heuristics, which reveals that it indeed minimizes cache misses and improves execution efficiency and real-time performance by as much as 15% and 60%, respectively.

Chapter 8

Future Work

There are several avenues we would like to pursue to further understand the potential costs and benefits of our partitioning algorithm, as well as other similar cache-aware partitioning algorithms. Because the field is relatively unstudied, and is becoming increasingly important as core counts increase and cache hierarchies become more complex, we hope our work is only the starting place of research in this area.

8.1 Task Dependencies

Our task model does not consider dependencies between tasks. We assume all tasks are capable of executing independently of one another and never must wait for another task. However, we recognize that real-time applications in enterprise environments typically contain at least some form of task dependencies. We therefore believe there would be benefit to extending our cache-aware partitioning algorithm to consider inter-task dependencies.

8.2 Aperiodic Task Releases

We make the claim that LWFG is easily adaptable for constrained sporadic tasks, in which each task has a known minimum arrival separation time. Often, this condition cannot be met, and a system which supports an aperiodic task model must be used. In this case, a static partitioning scheme will be insufficient, because the arrival times (or even the existence of tasks) is not known ahead of time. A cache-aware scheduler for such a system could be an online partitioning/clustering algorithm, or one of the cache-aware global scheduling variants. We note that the exploration of such an online cache-aware partitioning system would be immensely valuable to many real-world real-time systems which cannot provide assurances of task periodicity.

8.3 Wider Range of Taskset Distribution Parameters

We believe that there is still much room for exploration of taskset distributions in which more parameters are varied farther. In particular, we would like to study the effects of varying the percentage of total WSS shared between tasks in an MTT. In this thesis, we assumed that 100% of a task's WSS was shared with other members of its task group. This may be nearly true for some applications, but is most certainly not true for many others; it therefore warrants further study.

Furthermore, there is a combinatorial explosion of possibilities between the possible ranges and statistical distributions of all the taskset parameters: individual memory usage, task group memory usage, task period, task utilization, the relationship between the periods/utilizations/memory usage of tasks in the same group, the number of tasks in a memory-sharing task group, etc. While it would be impractical, if not impossible, to thoroughly study all possible combinations of these parameters, it would be beneficial to conduct an in-depth study of real-time applications in practice and their performance, and more closely align the taskset parameters with some subset or category of them.

8.4 Exploring Different Memory Access Patterns

We have primarily explored behavior with an in-order memory access pattern and a working set which did not vary in size or location in this thesis. This was done because it was necessary to find a fairly generic access pattern which would be somewhat generalizable to other applications. Moreover, not following these assumptions could introduce unpredictability in the tests we ran, especially if an application began accessing memory currently swapped out to disk. Unfortunately, as mentioned in Chapter 4, cache-awareness is very fickle and depends greatly upon many variables, one of which is the memory access pattern. For this reason, we propose exploration into three aspects of memory access in real-time applications:

1. Varying working set sizes. We assumed in this thesis that real-time applications contain the same WSS for the entire duration of their execution. This is, of course, an assumption violated by many applications.

2. WSS turnover. Though related to varying WSS, by this we mean that the physical region of memory being accessed is changing over time. For instance, in a multimedia-intensive application, the data being processed could change rapidly, and if each memory object is dynamically allocated, so could the physical memory backing it. The rate at which the working set memory regions change certainly has an effect on cache performance, and we believe this is worthy of future study.

3. Finally, we propose studying varying access patterns inside the current working set. For instance, we propose that instead of using an array as the data structure over

which accesses are iterated, that more advanced structures be used, such as linked lists or trees. In addition to swapping out the data structure, there are various patterns in which each of the data structures could be accessed. An example of this would be looping through an array versus performing an algorithm which makes several passes over the array at different strides (like a fast Fourier transform).

8.5 Partitioning With Non-EDF Algorithms

In this thesis, we restricted ourselves to studying partitioning algorithms for use with the uniprocessor EDF scheduling algorithm. There are many other uniprocessor real-time scheduling algorithms, with the most popular being rate-monotonic scheduling. We are interested to see how a static-priority uniprocessor scheduling algorithm such as RMS would work with LWFG, or a variant thereof.

8.6 Combination with DVFS Scheduling

Though we have shown our partitioning scheme can improve overall system efficiency by increasing IPC, further study is needed to take advantage of this increased execution efficiency. One potential avenue for research is studying how we can use the increased efficiency gained from LWFG and cache-aware scheduling in general to reduce power consumption with dynamic voltage and frequency scaling. This is very important for power-constrained computing environments such as autonomous vehicles, but could also be useful for reducing data center energy costs.

8.7 Cache-Aware Clustered Scheduling

Finally, we would like to test our partitioning scheme with clustered real-time scheduling approaches. Clustered scheduling is similar to partitioning except that tasks are assigned to a cluster of cores instead of an individual core. We believe that a clustered version of the LWFG partitioning algorithm may achieve some of the same cache-awareness benefits, while also retaining higher deadline satisfaction rates (and higher numbers of feasibly-partitioned tasksets) due to the larger bin size in the underlying bin-packing problem.

Bibliography

[1] CPU shielding using /proc and /dev/cpuset. https://rt.wiki.kernel.org/articles/c/p/u/CPU_shielding_using__proc_and__dev_cpuset_4449.html. Accessed February 6, 2012.

[2] Intel research :: Teraflops research chip. http://techresearch.intel.com/ProjectDetails.aspx?Id=151. Accessed February 17, 2012.

[3] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/. Accessed December 5, 2011.

[4] Real-time linux wiki. https://rt.wiki.kernel.org/. Accessed February 6, 2012.

[5] Tile processor architecture overview for the Tile*Pro* series. www.tilera.com/scm/docs/UG120-Architecture-Overview-TILEPro.pdf. Accessed April 25, 2012.

[6] TSE launches next-generation "arrowhead" trading system. http://www.fujitsu.com/global/news/pr/archives/month/2010/20100108-01.html. Accessed February 6, 2012.

[7] James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi. Real-time scheduling on multicore platforms. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 179–190, Washington, DC, USA, 2006. IEEE Computer Society.

[8] H. Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 9 pp., april 2003.

[9] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors., 2005. Technical Report TR-051101, Florida State University.

[10] S. Baruah. The partitioned EDF scheduling of sporadic task systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 116 –125, 29 2011-dec. 2 2011.

[11] Sanjoy Baruah and N. Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *Computers, IEEE Transactions on*, 55(7):918 – 923, july 2006.

[12] Sanjoy Baruah and Nathan Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 321–329, 2005.

[13] A. Bastoni, B.B. Brandenburg, and J.H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 14 –24, 30 2010-dec. 3 2010.

[14] A. Bastoni, B.B. Brandenburg, and J.H. Anderson. Is semi-partitioned scheduling practical? In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 125 –135, july 2011.

[15] Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proceedings of the Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, OSPERT 2010, July 2010.

[16] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for NUMA-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM.

[17] Martin J. Bligh, Matt Dobson, Darren Hart, and Gerrit Huizenga. Linux on NUMA systems. In *Proceedings of the Linux Symposium, Volume One*, pages 89–102, 2004.

[18] A. Burns, R. I. Davis, P. Wang, and F. Zhang. Partitioned EDF scheduling for multiprocessors using a c=d task splitting scheme. *Real-Time Syst.*, 48(1):3–33, January 2012.

[19] J.M. Calandrino, J.H. Anderson, and D.P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 247 –258, july 2007.

[20] John M. Calandrino and James H. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 299–308, Washington, DC, USA, 2008. IEEE Computer Society.

[21] John M. Calandrino and James H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 194–204, July 2009.

[22] B. Chattopadhyay and S. Baruah. A lookup-table driven approach to partitioned scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 257 –265, april 2011.

[23] Houssine Chetto and Maryline Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Trans. Softw. Eng.*, 15:1261–1269, October 1989.

[24] Sangyeun Cho, Lei Jin, and Kiyeon Lee. Achieving predictable performance with on-chip shared l2 caches for manycore-based real-time systems. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 3 –11, aug. 2007.

[25] R.K. Clark, E.D. Jensen, and N.F. Rouquette. Software organization to facilitate dynamic processor scheduling. In *IPDPS '04*, page 122, April 2004.

[26] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In D. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 46–93. PWS Publishing, Boston, 1997.

[27] D. Dasari, V. Nelis, and B. Andersson. WCET analysis considering contention on memory bus in COTS-based multicores. In *ETFA '11*, pages 1 –4, Sept. 2011.

[28] M. Dellinger, P. Garyali, and B. Ravindran. ChronOS linux: A best-effort real-time multiprocessor linux kernel. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 474 –479, june 2011.

[29] Matthew Dellinger. An experimental evaluation of the scalability of real-time scheduling algorithms on large-scale multicore platforms. Master's thesis, Virginia Tech, April 2011. http://scholar.lib.vt.edu/theses/available/etd-05122011-142219/.

[30] Ulrich Drepper. What every programmer should know about memory. `http://people.redhat.com/drepper/cpumemory.pdf`, November 2007. Accessed March 15, 2012.

[31] Carsten Emde. OSADL celebrates one-year anniversary of the QA farm... and presents long-term data of the determinism of mainline linux real-time. `https://www.osadl.org/Single-View.111+M59e3481cdfe.0.html`. Accessed April 20, 2012.

[32] J. Erickson, G. Coombe, and J. Anderson. Soft real-time scheduling in google earth. In *RTAS '12*, Bejing, China, 2012. IEEE Computer Society.

[33] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53:49–57, February 2010.

[34] Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. WCET analysis with MRU cache: Challenging LRU for predictability. In *RTAS '12*, Bejing, China, 2012. IEEE Computer Society.

[35] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, pages 245–254, New York, NY, USA, 2009. ACM.

[36] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *EMSOFT '09*, pages 245–254, New York, NY, USA, 2009. ACM.

[37] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 220–229, New York, NY, USA, 2008. ACM.

[38] Yunlian Jiang, Kai Tian, Xipeng Shen, Jinghe Zhang, Jie Chen, and R. Tripathi. The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions. *Parallel and Distributed Systems, IEEE Transactions on*, 22(7):1192 –1205, july 2011.

[39] M. Tim Jones. Anatomy of real-time linux architectures. `http://www.ibm.com/developerworks/linux/library/l-real-time-linux/`, April 2008. Accessed February 7, 2012.

[40] S. Kato and N. Yamasaki. Portioned static-priority scheduling on multiprocessors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –12, april 2008.

[41] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 249 –258, july 2009.

[42] Shinpei Kato and Nobuyuki Yamasaki. Portioned EDF-based scheduling on multiprocessors. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 139–148, New York, NY, USA, 2008. ACM.

[43] D.B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 229–237, December 1989.

[44] D.B. Kirk and J.K. Strosnider. SMART (strategic memory allocation for real-time) cache design using the mips r3000. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 322–330, December 1990.

[45] D.B. Kirk, J.K. Strosnider, and J.E. Sasinowski. Allocating SMART cache segments for schedulability. In *Real Time Systems, 1991. Proceedings., Euromicro '91 Workshop on*, pages 41–50, June 1991.

[46] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 239 –248, july 2009.

[47] G. Luculli and M. Di Natale. A cache-aware scheduling algorithm for embedded systems. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 199–209, December 1997.

[48] Zoltan Majo and Thomas R. Gross. Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 11–20, New York, NY, USA, 2011. ACM.

[49] Zoltan Majo and Thomas R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 12:1–12:10, New York, NY, USA, 2011. ACM.

[50] Jason Mars, Lingjia Tang, and Mary Lou Soffa. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 167–176, New York, NY, USA, 2011. ACM.

[51] David C. Snowdon Martin Lawitzky and Stefan M. Petters. Integrating real time and power management in a real system. In *Proceedings of the 4th Workshop on Operating System Platforms for Embedded Real-Time Applications*, Prague, Czech Republic, July 2008.

[52] Paul McKenney. Attempted summary of "RT patch acceptance" thread. `https://lkml.org/lkml/2005/6/7/256`. Accessed February 5, 2012.

[53] Paul McKenney. A realtime preemption overview. `http://lwn.net/Articles/146861/`. Accessed January 10, 2012.

[54] Larry McVoy. LMbench - tools for performance analysis. `http://lmbench.sourceforge.net/`. Accessed March 16, 2012.

[55] Matteo Monchiero, Ramon Canal, and Antonio González. Design space exploration for multicore architectures: a power/performance/thermal view. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 177–186, New York, NY, USA, 2006. ACM.

[56] Drew Prairie. New AMD opteron(tm) processors deliver the ultimate in performance, scalability and efficiency. Press Release, November 2011. `http://www.amd.com/us/press-releases/Pages/new-amd-opteron-processor-2011nov14.aspx`.

[57] J.C. Saez, J.I. Gomez, and M. Prieto. Improving priority enforcement via non-work-conserving scheduling. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 99 –106, sept. 2008.

[58] J. Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 195–204, 2000.

[59] S.K. Shekofteh, H. Deldari, and M.B. Khalkhali. Reducing cache contention in a multi-core processor via a scheduler. In *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, volume 6, pages V6–555–V6–558, aug. 2010.

[60] Tara Sims. Tilera announces the world's first 100-core processor. `http://www.tilera.com/about_tilera/press-releases/tilera-announces-worlds-first-100-core-processor`, October 2009. Accessed April 23, 2012.

[61] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 41–48, July 2005.

[62] Per Stenström, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, pages 80–91, New York, NY, USA, 1992. ACM.

[63] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 12–21, New York, NY, USA, 2011. ACM.

[64] Weixun Wang and P. Mishra. Dynamic reconfiguration of two-level caches in soft real-time embedded systems. In *VLSI, 2009. ISVLSI '09. IEEE Computer Society Annual Symposium on*, pages 145–150, May 2009.

[65] Weixun Wang, P. Mishra, and A. Gordon-Ross. SACR: Scheduling-aware cache reconfiguration for real-time embedded systems. In *VLSI Design, 2009 22nd International Conference on*, pages 547–552, January 2009.

[66] Yan Wang, Lida Huang, Renfa Li, and Rui Li. A shared cache-aware hybrid real-time scheduling on multicore platform with hierarchical cache. In *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*, pages 208 –212, dec. 2011.

[67] Wanghong Yuan and Klara Nahrstedt. Energy-efficient CPU scheduling for multimedia applications. *ACM Trans. Comput. Syst.*, 24(3):292–331, August 2006.

[68] Yi Zhang, Nan Guan, Yanbin Xiao, and Wang Yi. Implementation and empirical comparison of partitioning-based multi-core scheduling. In *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*, pages 248 –255, june 2011.

[69] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 129–142, New York, NY, USA, 2010. ACM.