ABSTRACT

SARKAR, ABHIK. Predictable Task Migration Support and Static Task Partitioning for Scalable Multicore Real-Time Systems. (Under the direction of Frank Mueller.)

Multicores are becoming ubiquitous, not only in general-purpose but also embedded computing. This trend is a reflection of contemporary embedded applications posing steadily increasing demands in processing power. On such platforms, prediction of timing behavior to ensure that deadlines of real-time tasks can be met is becoming increasingly difficult. While real-time global/semi-partitioned multicore scheduling approaches help to assure deadlines based on firm theoretical properties, their reliance on task migration poses a significant challenge to timing predictability in practice. Task migration actually (a) reduces timing predictability for contemporary multicores due to cache warm-up overheads, (b) renders locking of cache lines infeasible for multicore real-time systems and (c) increases traffic on the network-on-chip (NoC) interconnect. Additionally, prior work in static task partitioning on multicore architectures focuses on shared cache organization with a fixed number of cores. Such schemes are not suitable for static partitioning on scalable multicore architectures that feature private cache organization. We attempt to address these limitations in this dissertation.

The following are the key contributions of this work:

1. First, a task migration into two cores imposes cache warm-up overheads on the migration target, which can lead to missed deadlines for tight real-time schedules. We propose a novel push-assisted cache migration model to pro-actively migrate cache lines through novel software and micro-architectural support. Our mechanism imposes cache migration delays at a fraction of the task's execution time. This delay can be steered to fill idle slots in the schedule, *i.e.*, it does not contribute to the execution time of the migrated task. We also propose micro-architectural modifications that further reduce the delay and bus traffic.

2. Second, locked cache lines that are predominantly used in real-time systems are immobile during task migration. We address this issue by extending the push-assisted migration model with several cache migration techniques to efficiently retain locked cache lines on a bus-based chip multi-processor architecture. We also provide deterministic migration delay bounds that help schedulers to decide which migration technique(s) to utilize while migrating a single or multiple tasks. This information also allows the scheduler to determine feasibility of task migrations, which is critical for the safety of any hard real-time system. Such proactive migration of locked cache lines in multicores is unprecedented to our knowledge.

3. Third, we further the use of locked caches on scalable multicore architectures by analyzing its impact on static task partitioning algorithms. In shared cache architectures, a single resource is shared among *all* the tasks. However, in scalable cache architectures with private caches, conflicts exist only among the tasks scheduled on one core. This calls for a cache-aware allocation of tasks onto cores. Here, we propose a novel variant of the cache-unaware First Fit Decreasing (FFD) algorithm called the Naive locked First Fit Decreasing (NFFD) policy. We propose two cache-aware static scheduling schemes: (1) Greedy First Fit Decreasing (GFFD) and (2) Colored First Fit Decreasing (CoFFD) for task sets where tasks do not have intra-task conflicts among locked regions (Scenario A). NFFD is capable of scheduling high utilization task sets that FFD cannot schedule. CoFFD consistently outperforms GFFD requiring a lower number of cores and lower system utilization. For a more generic case where tasks have intra-task conflicts, we split the task partitioning into two phases: Task Selection and Task Allocation (Scenario B). Instead of resolving conflicts at a global level, these algorithms resolve conflicts among regions while allocating a task onto a core and perform unlocking at region-level instead of task-level. We show that a combination of our novel Dynamic Ordering (Task Selection) with Chaitin's Coloring (Task Allocation) scheme reduces the number of cores required considerably over a basic scheme (combination of Monotone Ordering and Regional FFD).

Overall, this dissertation suggests that deployment of locked caches and hardware support for cache migration on scalable multi-processors can enable more predictable and efficient multi-processor scheduling.

Predictable Task Migration Support and Static Task Partitioning for Scalable Multicore Real-Time Systems

by
Abhik Sarkar

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2012

APPROVED BY:

James Anderson	Xuxian Jiang
Helen Gu	James Tuck

Frank Mueller
Chair of Advisory Committee

DEDICATION

To Ma and Baba.

BIOGRAPHY

Abhik Sarkar was born and raised in New Delhi, India. He attended St. Xavier's high school, Delhi. Thereafter, to pursue a degree in Electronics and Communication Engineering he attended Delhi University. After completing his undergraduate degree, he worked for two years as an Associate Systems Lab Engineer in Advanced Systems Technologies group at STMicroelectronics, India. While working in a research group led by Dr. Kaushik Saha, he developed a keen interest in parallel processing and parallel computer architectures. During his tenure at ST, he worked on parallel implementations of audio/video codecs.

In fall 2005, he moved to North Carolina as a graduate student in the Department of Electrical and Computer Engineering at North Carolina State University (NCSU). There, he worked under the guidance of Dr. Yan Solihin and Dr. Joe Defenderfer (Tekelec Corp.). As a part of his Master's degree in Computer Engineering, he developed libraries to assist development of lock-free data structures. Thereafter, he joined the Department of Computer Science at NCSU in 2008 to continue his graduate studies as a doctoral student in systems research group led by Dr. Frank Mueller. Since then he has been pursuing his research interests in hardware/software co-design for predictable multicore real-time systems. During his doctoral studies he also interned in research labs at Intel and VMware. After finishing his doctoral studies at NCSU, he will be joining Intel Corporation.

ACKNOWLEDGEMENTS

It had drama, excitement, periods of boredom, moments of brilliance, dozes of good calls and poor judgments, tests of patience, lack of concentration, desperate dives, constant pressure, words of confidence, disappointing runs, lots of prayers, back and wrist problems, distance from family, but at the end a tryst with tearful celebrations. As every cricket enthusiast would call it the description of a single test inning played by the god of cricket, Sachin Ramesh Tendulkar. For me, that is the description of my PhD. For non-cricket playing souls, please enlighten yourself with a copy of May'12 edition of TIME magazine. During this period, one of the lessons I have learnt is to be grateful, which makes the acknowledgments an integral part of this dissertation.

Firstly, I would like to thank my parents for teaching me the importance of perseverance, for inspiring me with their selflessness, and for all the heart-felt wishes that have made it possible. In their lives, my father wanted to have an engineering degree and my mother wanted to be a doctor. This dissertation strikes two birds with one stone. How? This is left as an exercise for the reader. Ma/Baba, I dedicate this dissertation to you.

Secondly, I would like to thank my advisor Dr. Frank Mueller. His commitment towards me and my research has always motivated me to do better. Throughout my research, he has given me invaluable advice and guidance, both in research and life. His constant flow of ideas has amazed me but at the same time, challenged me to think beyond the conventional. His method of operating with doodles on discussion boards has enabled me to appreciate the concept of controlled chaos. It has been an invaluable experience to work with him and it is impossible for me to express my gratitude towards him in words.

I am grateful to Paula D'Onofrio for being the motherly figure to me when I needed the most. Her unconditional support for my dreams and aspirations is the integral part of maintaining my focus in this long drawn process. Her furry feline kids Bella, Willoughby, and Oliver have been great stress busters. Their research on the importance of slumber as a mechanism to relieve stress has motivated me to follow suit. I am grateful to Dr. Robert Jenks whose words of encouragement have imbibed self belief. He made me realize that wooden block puzzles can have a humbling effect. I thank Corrine Shuster for making me realize the importance of gratitude and savoring life in every situation.

Special thanks to Dr. James Tuck for helping me with simulation environments and the discussions that have helped me understand computer architecture at depth. I sincerely thank Dr. James Anderson, Dr. Helen Gu and Dr. Xuxian Jiang for their advice, suggestions and questions that have contributed substantially to making this thesis comprehensible.

I would like to thank Nikhil Deshpande, with whom I have shared a large part of my graduate

student life. We started by being disgruntled room mates but over the period of seven years, I have developed immense respect for this class act. Then there were two, Suresh Thummalapenta and Gaurav Bawa, who are the embodiment of hard work and Indian community networking. Special thanks to John Whitlow for accommodating me under his roof that I have been calling home during my final year of PhD. You can only wish to have a landlord who ships Garra Rufa fish to get you manicured and pedicured while you write your dissertation. I would like to thank this small family of spiritual beings at Ganesh Place in Durham, who have made me smile through the tough times either with profound love or with gifted goofiness. I am grateful to Dorothy for including me in her annual "feastivities" with the "Mazzitellis", a musical family with a big heart. Thanks Joe for making this world a little more musical. Ferrrrnando Rubio, I thank you for painting my graduate student life with shades of fun loving salsa attitude.

And then comes the turn of the Beagle Boys at the Systems Research Lab in NC State. Christopher Zimmer is the "Ma Beagle", who runs the show and keeps an eye on the door whenever anyone walks in. David Fiala is the "Bigtime Beagle", who leads the pack to everything mischievous. Arash Rezaei is the "Bebop Beagle" wearing jazzy outfits. Dr. James Elliotte is the personification of "Babyface Beagle" for the obvious reasons. Yangpeng Zhang and Xing Wu represent the "Bankjob Beagle" with exceptional soccer and publishing skills. And I am the "Burger Beagle" for obvious reasons again. As a Beagle Boy I have loved the passionate discussions on research and everything besides that. Without them I would not have learnt the importance of laying low and keeping the banter-guns loaded. Thanks to all of you for being a part of my clique.

Over the years many graduate students have left a brush stroke or two on the canvas of my life. Amit Awekar, Ravi Ramaseshan, Prasad Wagle, Sandeep Navada, Niket Chaudhary, Devesh Tiwari, Xiaowei Jiang, Siddharth Chhabra, Brian Rogers, Nicky Mahilani, Bala Subramanya Bhat, Karthik Babu, and many more. I thankful to all of you for being a part of my journey.

Since this all started when I was a student at Delhi University and an employee at STMicroeletronics in India, I would like to thank Dhananjay Gadre, Dr. Raj Senani, Saibal Dutt, Dr. Mona Mathur, Dr. Kaushik Saha, Srijib Narayan Maiti, Surinder Pal Singh, Yogesh Kumar Soni, Sumit Johar and Sandeep Dabas for motivating me with their brilliance. Special thanks to Rahul Gandhi for being there when I needed him the most.

Last but not the least, I would like to thank the staff of Office of International Students, Department of Computer Science, and Graduate School Office, at NC State and to everyone else behind the scenes for all their help. Special thanks to Dr. Douglas Reeves and Dr. David Thuente for helping me through the process of graduate studies at NC State.

To all of you who have been mentioned, I wish you the very best in life and I hope to keep running into you. To those whom I have inadvertently omitted, I sincerely apologize and

request you to let me know how I can make up for it.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Real-Time Systems

Real-time Systems are computing environments where temporal correctness is as important as functional correctness. A real-time system is temporally correct if all real-time tasks meet their timing constraints. Control systems use real-time software to operate in a temporally correct fashion in automobiles, aerial vehicles, audio-video devices, medical radiation devices, etc. A single real-time system can be composed of periodically executing real-time tasks that collect information from their environment and trigger a response. A delayed response could result in artifacts that can impact the quality of service or even lead to catastrophic consequences depending upon the purpose of the system. In a mobile device supporting audio/video applications, the frame rate leading to a "flicker-free" visual experience governs the amount of time in which a video frame should be decoded. A delayed response will result in dropping video frames, which in turn impacts the visual experience of the customers. However, such failures do not render a system nonfunctional. Such tasks are called soft real-time tasks and have soft deadlines. In contrast, when tasks in Anti-lock Brake System(ABS) fail, it causes the wheels to lock-up, which could lead to catastrophic consequences like loss of life. Such tasks are called hard real-time tasks and have hard deadlines.

In a system, real-time tasks can be executed periodically, aperiodically or sporadically. Periodic tasks have regular arrival times. Aperiodic and sporadic tasks may have irregular arrival times. Aperiodic tasks have soft deadlines while sporadic tasks have hard deadlines. This work assumes tasks to be periodic. Each task i is periodically released after a time interval of P_i. Every instance of task i is called a job and denoted as J_i^n where n is the nth instance of task i. Every real-time task is given a relative deadline D_i. Every real-time job has to respond before a pre-calculated instant of time called its deadline d_i^n, which is computed by adding the task's relative deadline, D_i, to the time instant t at which the nth job of task i was

released. If J_i^n does not finish execution before d_i^n, then the job is deemed to have missed a deadline. The relative task deadline can be less than, equal to or greater than a task's period. In our work, we assume that $D_i \leq P_i$, which means that J_i^n has to finish prior to the release of J_i^{n+1}. The consequence of missing a deadline depends on the criticality of a task. As stated earlier, a task that renders an encoded stream of video frames has a soft deadline. A video rendering task failing to meet a deadline leads to a poorer visual experience. However, if a task that releases brakes fails to meet its deadline in an ABS system, it can cause catastrophic consequences. Such critical tasks have hard deadlines. Tasks with hard and soft deadlines are called hard and soft real-time tasks, respectively. A real-time system is composed of at least a soft/hard real-time task while it can run multiple real-time or non-real-time tasks. In order to reduce operational costs and to leverage the computational capacity of modern processors, real-time systems are usually multi-programmed.

Task scheduler(s) running on these systems are responsible for invoking jobs of periodic tasks. For soft real-time systems, task schedulers may also incorporate a responsive action when a deadline miss is detected. A deadline miss can be detected in several ways. For example, a timer-based interrupt can be scheduled by the task scheduler at d_i^n for the invocation of J_i^n. The interrupt will initiate a interrupt routine that checks whether J_i^n has finished its execution. In case a job misses its deadline, the system has to respond to prevent/reduce subsequent jobs from missing their deadlines. These actions may vary depending upon the implementation of the system. For example, the task scheduler may choose to terminate J_i^n or skip job J_i^{n+1}. In case certain subsequent jobs are dependent upon J_i^n, then those jobs need to be skipped as well. In contrast, for hard real-time systems it becomes imperative that each of the task meet their deadlines for it to be functional. For such systems, the developer needs to analyze the tasks and make sure that all the tasks meet their deadline prior to launching the system. This is called *schedulability analysis*. Schedulability analysis requires the following information:

1. The Worst-Case Execution Time (WCET) of each the task represented as E_i. This can be obtained by

   (a) Static analysis of the source code. This has been a research topic for the real-time systems community for more than three decades. With the advent of multicores, multi-level caches and network-on-chip (NoC), it is still an open research area. A static analysis tool would require certain parameters of a processor architecture, e.g., execution latency of instructions in an Instruction Set Architecture (ISA), access latency at different levels of memory hierarchy, number of cycles lost due to a branch instruction on a system with branch prediction turned off etc. The tool then parses the instructions and, using an intermediate representation (IR), computes the worst case number of cycles needed for each basic block to execute. Parsing the IR, like an

abstract syntax tree, allows it to converge to a worst case execution path (WCEP) that delivers the WCET. Since programs usually execute in control flow of tight loops, the static analysis tool needs the information on the maximum number of iterations of each loop. Several static analysis tools are available though they are prone to deliver high upper bounds. However, if static analysis of a task delivers a WCET lower than the period, the bound is considered safe since the real-time task cannot exceed this bound.

(b) Dynamically observing the execution time of the task with an exhaustive input set. This scheme is widely used as certain parameter values assumed by static analysis tools, like using an upper bound on load-dependent branches, might just be too high than the ones observed in practice.

This allows us to compute the task density as $\frac{E_i}{D_i}$. This is the computational demand of a task on a computational resource. $\frac{E_i}{P_i}$ gives us the task utilization, which is the same as task density when $P_i = D_i$.

2. A schedulability test that governs whether a task set is schedulable or not on a given computing resource. If a task set passes the schedulability test, then it guarantees that all the tasks will meet their deadlines. However, a schedulability test is dependent upon the scheduling algorithm. Over the past four decades, various task scheduling algorithms have been proposed on a computing resource. They are broadly characterized as

   (a) Clock driven: These are also known as off-line schedulers as these schedulers fix the schedules before the system starts. Table Driven and Cyclic scheduling are two commonly known algorithms under this category [47].

   (b) Event driven: These schedulers dynamically schedule tasks at run-time where the scheduling points are governed by job invocation and job completion. These schedulers can be preemptive or non-preemptive. A preemptive scheduler could suspend the execution of an already executing task in order to execute a higher priority task. A non-preemptive scheduler will only grant the computing resource on termination of an already executing task. Rate Monotonic (RMA) and Earliest Deadline (EDF) are examples of Event driven scheduling algorithms [54, 29].

   A feasible schedule on a single computing resource will not miss any deadline even when all the tasks are running at their WCET. In other words, a feasible uniprocessor schedule is the one that satisfies the following condition

   $U = \sum_{i=1}^{n} \frac{E_i}{D_i} \leq 1$ *, where n is the number of tasks [54]*

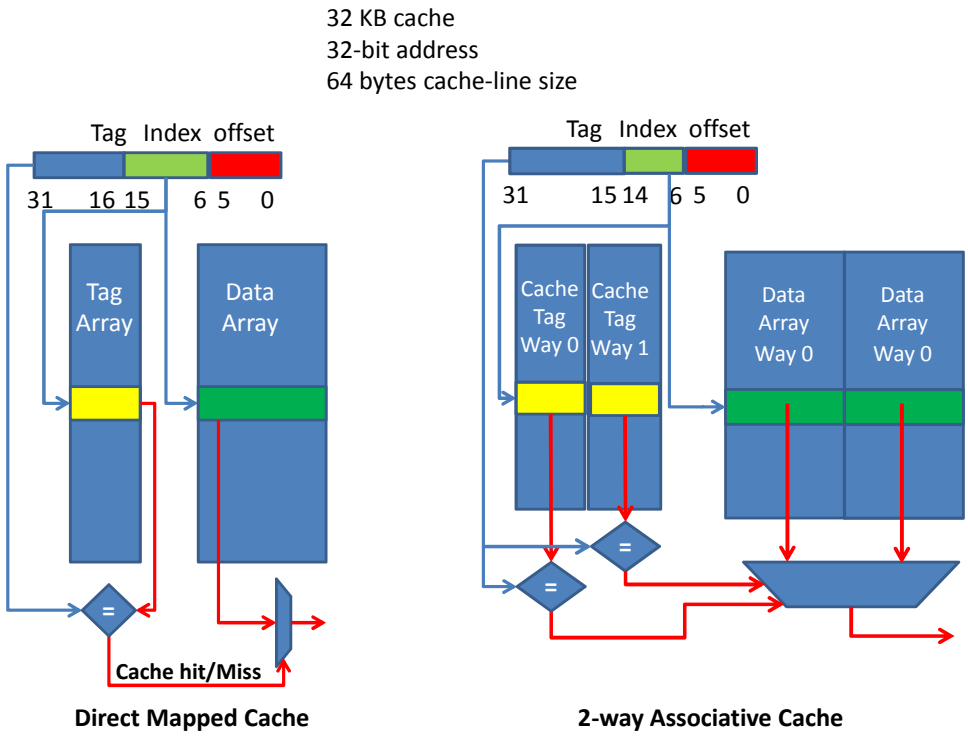
Here, U is the aggregate of task densities of all the tasks scheduled on a computing resource. If we assume $P_i = D_i$, then U can be called the aggregate utilization. The preemptive Earliest Deadline First (EDF) algorithm obtains this optimality for uniprocessor real-time systems. If $U > 1$, then the task set cannot be scheduled on a given computing resource. When U < 1, the system is under-utilized and multiple tasks can share the computing resource. This needs multiple real-time tasks to be scheduled by a real-time scheduler that enforces a scheduling policy to share the computing resource.

Most contemporary processors incorporate complex micro-architectural enhancements. Out of order execution [58, 83], branch prediction [99, 100, 82] and speculative execution with value prediction [53, 78] are some of the key features that enable these uniprocessors to deliver high throughput. However, all these features hamper the predictability of WCET for real-time applications, which is the first requirement in order to perform a robust schedulability analysis. Thus, for real-time systems, the focus has been to keep the micro-architecture simple by using in-order processor cores without branch prediction or speculative execution. Another aspect that throttles the throughput is the memory access delay. With higher throughput processors the memory access delay gets worse. Even with lower processor frequency the memory access delays are expensive, e.g., with processor cores running at 700 MHz it takes 70 cycles to access memory for Tilera processors [36]. To reduce memory accesses, on-chip caches are used to store the recently accessed memory locations. For general-purpose computing, cache performance is measured by observing the rate at which the memory references are not found in cache, called the cache miss-rate. The lower the cache miss-rate, the more effective a cache is. The cache miss rate is relevant for assessing the average performance. This information is not sufficient for predictability for real-time systems. Abstract Cache Analysis (ACA) is a mechanism by which the worst-case cache behavior is modeled [71, 79, 68]. Abstract Cache Analysis is incorporated with the aforementioned static analysis tools. Instead of assuming all memory references to cause off-chip memory accesses, an abstract cache is assumed. It is called abstract because the state of a memory address is abstract in this cache. The absolute state of the cache is known only at run-time as it is dependent upon the run-time control flow. However, abstract cache analysis statically analyzes the control-flow graph and determines the abstract state of the cache at any point in the program, which allows one to predict all cache hits that "must" exist regardless of the path taken by the program [89]. Static analysis then follows and calculates the execution time of basic blocks while assuming cache hits for "must" hit references predicted by ACA. Eventually, it converges by determining the WCEP and the WCET. In order to perform ACA, static analysis needs to know some of the key aspects of cache organization.

The following are the features of cache organization that will be referred in this dissertation:

1. Cache line: The smallest granularity at which a cache is accessed. Theoretically, cache line

Figure 1.1: Generic view of Cache Architecture



size can be anywhere from the smallest granularity at which the memory is being accessed to the whole cache size. So, for a 32-bit RISC architecture with 8KB cache may have cache line size from 32 bits to 8KB. However, in practice neither of them are considered suitable. With the former cache line size the spatial locality is sacrificed [43]. Spacial locality suggests that there is a high probability that subsequent memory accesses go to neighboring memory locations. E.g., this behavior is inevitable especially for instruction accesses within a basic block. Having a single cache line is not a good idea either. If only a single cache line is available, then at any time only one block of 8KB can be present in the cache. When a new block is referenced, the new block replaces the older block. If every memory reference refers to different 8KB blocks, then every access will cause a cache miss. Such misses are called conflict cache misses. Thus, cache line sizes in contemporary micro-processors are 32/64 bytes in size [94].

2. Associativity, Cache sets and Cache ways: In order to explain these characteristics, we explain the mechanism by which caches are referenced using a physical memory address. Figure 1.1 shows two types of organizations: A direct-mapped cache and a 2 way set-associative cache. We assume a 32KB cache being accessed by a 32-bit address. We also

assume that 64 bytes as the cache line size. In both the cases, the address is broken into 3 parts:

(a) Offset: This is used to address the individual bytes within a cache line. Since the number of bytes in a cache line are 64 bytes, the number of bits required to address a 64 byte sized structure is 6 bits. In both cache organizations, the offset uses the low order 6-bits.

(b) Index: These are the bits that identify if the data corresponding to an address is located in a cache. For a direct-mapped cache, there is room for only one cache line per index. With a 2-way set associative cache, there are two entries per index. The associativity of a direct-mapped cache is 1. It is 2 for a 2-way set-associative cache. The set of locations that are specified by an index in a set-associative cache is called a cache set. $\log_2 \frac{CacheSize}{Associativity \times Cache\ lineSize}$ number of bits are used as an index.

(c) Cache tag: The higher order bits left excluding the aforementioned bits are used as the tag. When a cache line is read into a cache, the tag of the cache line is stored in the cache-tag arrays.

When a cache is accessed, the index bits are used to select a cache set in two arrays: the tag array and the data array. Comparisons between the cached tags and the reference tag give us a cache hit/miss signal. On a tag match, it is a cache hit; otherwise, it is a cache-miss. These signals are then used as select lines for the multiplexer that forwards the data corresponding to the tag that has matched. Figure 1.1 shows that our 2-way set-associative cache has 2 memory arrays for tags and 2 memory arrays for data. Each group of memory arrays, i.e., one tag array and its corresponding data array, is called a cache way. A set associative cache is called fully associative when there is a single cache set and any cache line can map within that set. So a 32KB fully associative cache will have a single cache set holding 1024 entries. Usually, fully associative caches are used for smaller cache structures. Also, a 8/16-way set-associative cache approximates a fully associative cache [43]. There are no index bits for fully associative caches because there is only one set and tag bits are all high-order bits excluding offset bits.

3. Cache addressing: The memory address used for addressing a cache can be a physical or virtual address depending upon whether the memory management unit (MMU) for virtualizing memory is used or not. Nearly all implementations of virtual memory divide a virtual address space into virtual pages that map to physical memory regions called the frames. This requires virtual to physical address translation before the physical memory is referenced. A cache can be addressed using either of these addresses. If the virtual address is used then on every context switch, all the contents of the cache need to be flushed. If

a physical address is used with the MMU unit turned on, the translation from virtual to physical address needs to be done prior to accessing the cache. This makes cache access a multiple cycles process. It is worse for real-time systems as dynamic allocation of physical frames to pages makes it impossible to perform ACA as a virtual memory reference can be mapped anywhere in the cache. In high performance systems, the caches are addressed virtually indexed and physically tagged. The virtual indexing allows virtual to physical memory translation to take place in parallel to the retrieval of cache tags and cache lines from indexed cache sets. However, this suffers from aliasing, which happens when some of the bits in the virtual index are changed during translation. In contemporary systems, pages are usually 4KB. A 4KB page can be addressed by using the 12 low-order bits of a given virtual address. These are called page offset bits. The page offset bits remain the same but any of the remaining 20 address bits can be altered during translation. This poses a problem. In Figure 1.1, we see that the index bits went well past the 12th bit. There are several ways to tackle this during run-time but for real-time systems it causes indeterminacy. Since contemporary memories are becoming denser, real-time systems can run with the MMU turned off so that a flat physical memory address is used directly [94]. The physically indexed and physically tagged cache is a viable and predictable option for real-time systems.

4. Types of Cache-Misses: When a program starts execution, it does not have any of the program's address space cached. This is a state when the cache is said to be "cold". As the program resumes execution it populates the cache, i.e., the cache becomes "warm". When a cache miss occurs due to accessing a cache line for the first time, it is called a cold or compulsory cache miss. A capacity cache miss occurs if the program's address space does not fit within a given size of the cache. Another form of cache miss is called the conflict cache miss. Conflict cache misses are those that occur in a cache with a given associativity and that could have been prevented by a fully associative cache of same size.

5. Replacement Policy: On a cache miss, a new cache line is brought in from memory. In a set-associative cache, a cache miss occurs if all the cache lines in the cache set are marked invalid or none of the valid cached tags match with the address' tag. Invalidations of cache lines occur due to events like cache flushes and invalidations to maintain coherence between processor and I/O devices or between processors on a multicore system (to be described later). If invalid cache lines exist in the indexed set, the incoming cache line is placed there and the cache entry is marked valid. In case all the cache entries are valid, one of the cache entries needs to be replaced. Replacement policy decides which cache line to replace. The replacement policies have widely been studied and proposed. The most commonly used policy is Least Recently Used (LRU). LRU chooses the cache entry

that has been residing in the cache set without being referred for the longest time. The most efficient use of LRU requires a set of counters per cache entry. All the counters need to be updated at every cache hit/miss. This increases the design complexity of caches. Hence, most micro-processor use a pseudo-LRU policy [28]. Locked caches provide some user control over replacement. Locking a cache line prevents a cache line from being replaced. In the context of real-time systems, locked caches have been used for achieving predictability. This is the key reason for cache locking to be prevalent among embedded processors, like the IBMPowerPC 460S,MotorolaMPC7400, Intel 960, ARM940T etc. This will further be discussed in the later chapters.

6. Multi-level Caches: One would want to design a large cache structure to reduce cache-misses considerably. Larger caches have higher access time. A processor pipelines instructions to improve throughput but in order to pump a new instruction in the pipeline every cycle, it should be able to read an instruction every cycle. Therefore the cache access latency impacts the critical path of a processor. However, having smaller caches will lead to a higher cache-miss rate. A middle ground is reached with multi-level caches. The cache that is closest to the core processor is called the Level-1 cache or the L1 cache. The farther it goes from the processor, the size and the level of the cache increase. This allows the L1 cache to be smaller and fast enough to match the throughput of the processor. An L2 cache can be larger but is still 1-2 orders of magnitude faster than the memory access latency. Normally, a cache miss at any level leads to an access to the next higher level of cache. By higher level, we mean farther away from the processor. The traversal of a cache line from one level to another is another important aspect of multi-level caches. In an *inclusive cache* structure, the higher level cache retains the cache line while forwarding the content to the lower level. In contrast, an *exclusive cache* structure invalidates a cache line at the higher cache level. Inclusive caches are useful as they reduce complexity of cache design. And later in our discussion on multicore cache hierarchy, we will see that it helps in maintaining coherence at a higher cache level. Exclusive caches are capable of using the aggregate cache capacity as they do not maintain additional copies.

7. Split and Unified caches: L1 caches are small structures and they are commonly accessed by two different pipeline stages (with the exception of self modifying code). Also, access patterns of instructions are different from that of data. A unified cache holds both instruction and data cache lines. A small unified cache would result in data cache lines and instruction cache lines to evict each other. This phenomenon is known as thrashing. To minimize the thrashing effect and to be able to design caches customized to instruction and data access behavior separately, vendors use split L1 caches (L1 instruction and L1 data cache). Higher-level caches are designed to be unified as they have large capacity.

8. Write-through and Write-back caches: A write-through cache sends updates to its higher levels whenever a cache line is being updated. A write-back cache does not send an update, but updates a dirty bit associated with the cache line instead. Cache lines at higher cache levels get updated when they are evicted from lower levels. If a cache connected to the memory controller is a write-back cache then it considerably reduces the off-chip traffic. We will discuss more pros and cons of these schemes when we discuss multicores in the following section.

Even though the ACA determines lower WCET bounds, with increasing processing demand on real-time systems, task sets might not be schedulable over uni-processors. One of the intuitive mechanisms to improve the throughput of real-time applications would be to increase the processor frequency. Moore's Law has been holding for a long time in microprocessor design, yet single-processor designs have reached a clock frequency wall due to fabrication process and power/leakage constraints [5]. This has led designers to investigate the option of chip multiprocessors (CMPs), a.k.a. multicores, to ensure that performance increases at past rates.

1.2 Multicore Architectures

Application specific cores or accelerators have been used in many embedded systems. In the past decade the advancement in making homogeneous multicore platforms had been the central focus of many micro-processor vendors. Symmetric Multi-Processors (SMPs) were developed with the idea of having multiple processor chips connected via an external shared memory bus, which is used to interface the processors and the memory controller. There are numerous examples of such systems, ranging from the high-end SGI Power Challenge, AMD Opteron, Sun Ultraserver and DEC Alpha Server to the low-end NVidia Tegra and intel Xeon processors. Requests issued by all the cores are serialized on the SMP bus by a bus arbiter. If the cores use a write-through cache, each write would then be sending a request to the memory. This would incur a large amount of traffic that could throttle the performance of all the programs significantly due to lack of memory bandwidth for simple read requests. The other option is to use a write-back cache. This will prevent the updates to go to memory in introducing every time there is an update. This would reduce the traffic but it introduces a problem of a non-coherent view of data. With write-through cache the memory is always coherent with regard to the contents of the caches. And every time an update occurs it becomes visible on the bus. A snoop control unit, which is a part of the cache controller, is responsible for snooping the visible update requests and, in case the updated cache line is present in local cache, then update it. This requires a cache to have one read/write port for local processor requests and another one for the snoop requests. Such a design impedes the performance as a large portion of memory bandwidth is consumed by the update traffic. Modern processors with write-back caches absorb this update traffic but

leave memory with stale contents. This also prevents any of the updates to be visible to remote snoop controllers. But it leaves the cache incoherent. In order to ensure coherency, various cache-coherent protocols for SMPs were developed. One of the most common protocols is the MESI protocol. It has four different states that cache lines can be in. The following are the coherency states:

1. Modified (M): This means that the cache line is present only in local cache, the contents of the cache line are different from the contents of the memory, and the local processor is permitted to modify the contents.

2. Exclusive (E): This state dictates that the cache line is resident only in local cache but its contents are coherent with that in memory. The local processor does not have the right to modify the contents.

3. Shared (S): This suggests that there are copies held by remote caches besides the local cache. The contents of the cache line are coherent with memory.

4. Invalid (I): The content of the cache line is not valid.

A cache line's state can be modified by local/remote accesses. The memory accesses generated by the local processor are either a read or a write. The key objective of the coherence protocol is to serialize the writes to a cache line (write serialization) and to make the write visible to all the processors (write propagation). This requires a signal to be issued by the cache controller when a local write request finds a cache line in a state other than Modified. We call this signal Read With Intent to Modify (RWITM). When a remote snoop controller snoops an RWITM, it accesses the cache to check if it holds a copy of the cache line. On finding a match, the state of the cache line is transitioned to the Invalid state. If the line in the remote cache is found in Modified state, the contents of the cache line are "flushed". Flushing of a cache line means that the memory is updated with the contents of the cache line. On receiving acknowledgments from all the processors, the snoop controller on the requesting processor updates the state of the cache line to modified and the contents of the cache line are modified.

There are several state transitions that take place due to local and remote requests. Figure 2b shows the state transitions that take place due to locally initiated accesses. The following is a description of the state transitions:

1. The cache line is in the Modified State:

   (a) Read Access : Read the contents and stay in the Modified State;

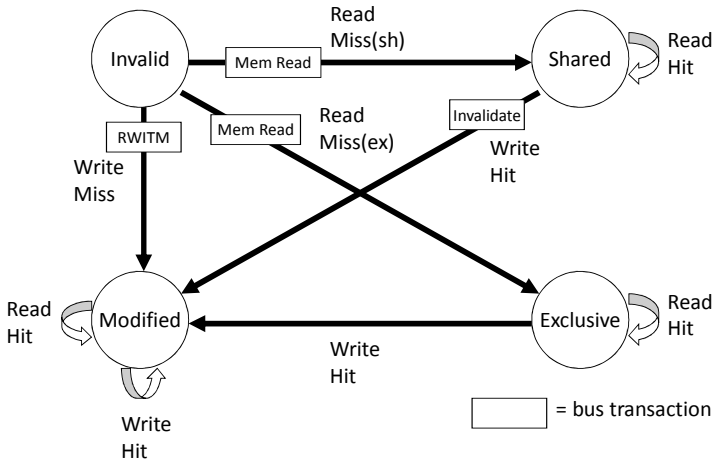
   (b) Write Access: Update the contents and stay in the Modified State.

2. The cache line is in the Exclusive State:

(a) Read Access : Read the contents and stay in the Exclusive State;

(b) Write Access: Transition to the Modified State and update the contents.

Figure 1.2: MESI-Locally Initiated Accesses



3. The cache line is in the Shared State:

   (a) Read Access : Read the contents and stay in the Shared State;

   (b) Write Access: Issue a RWITM bus transaction; on receiving an acknowledgment from the rest of the processors, transition to the Modified State and update the contents.

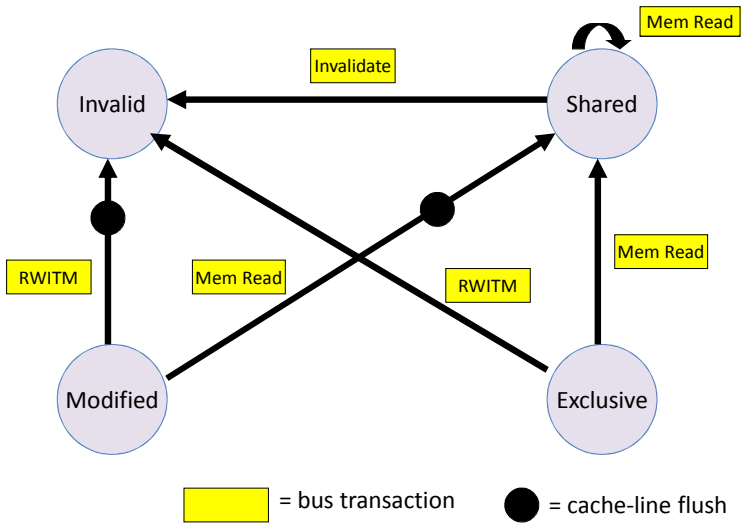
4. The cache line is in the Invalid State:

   (a) Read Access : This means that the cache line is not present in the local cache. Hence, a mem read bus transaction is issued. When none of the remote caches hold a copy, the state of the cache line is changed to the Exclusive State. If one or more remote caches hold a copy of the cache line then the cache line has to be allocated in the Shared State.

   (b) Write Access: In response to a cache miss, a RWITM bus transaction is issued. The state of the cache line is changed to the Modified State.

Figure 1.3 exhibits the state transitions caused by remotely initiated accesses. The remotely generated bus transactions are RWITM and Mem Read by the cache controllers in response to

Figure 1.3: MESI-Remotely Initiated Accesses



locally generated accesses. As we explained before, the snoop controller snoops the bus at every bus cycle for remote bus transactions. The following is a description of the state transitions caused by the snoop controller:

1. The cache line is in the Modified State:

   (a) Mem Read : The state of the cache line switches to Shared State. But this cache line also has the latest updated value, so the contents of the cache line are flushed to update the memory.

   (b) RWITM : The cache line is invalidated and the contents are flushed.

2. The cache line is in the Exclusive State:

   (a) Mem Read : The state of the cache line switches to Shared. Note that there is no cache to cache exchange of cache line content. This is because cache-to-cache and memory-to-cache latencies are comparable for SMPs. Thus, the contents can very well be supplied by the memory. However Chip Multi-Processors (CMPs) have on-chip cores connected via an on-chip interconnect. In such architectures, it is efficient to have cache-to-cache transfers.

   (b) RWITM : Invalidate the cache line and the contents are read from memory.

3. The cache line is in the Shared State

(a) Mem Read : The cache line stays in the Shared State

(b) RWITM : Invalidate the cache line and the contents are read from memory.

4. The Cache line is in the Invalid State: This means that remote cache does not hold a copy of the accessed cache line.

Advancement in fabrication technology has enabled vendors to pack a larger number of processor cores with private and shared caches within a single chip. This has motivated researchers to investigate different multi-processor cache structures.

1. Shared higher-level caches (like an L2 cache) with coherent lower-level caches (like an L1) that are connected via a coherent bus have been proposed. They have become prevalent in contemporary embedded system designs, e.g., the NVidia Tegra 3 has four ARM processor cores with 32KB L1 Instruction/data caches and a 1MB shared L2 cache [4]. The primary concern looking forward with shared L2 caches is that they do not scale with large numbers of cores. This is due to the coherency traffic as a single RWITM causes all the cores to check their caches and send an acknowledgment. Also, large L2 caches impact the access latency thereby hurting the common case scenario. Large L2 caches contribute to 50% of power dissipation due to leakage current, which contributes significantly to overall power consumption [91].

2. Multi-level private caches. The private caches in multi-programmed workloads exhibit lower conflicts and reduced access latency for L2 caches [16]. These processor cores with private L1+L2 caches may be connected via a shared bus or point-to-point interconnects. The Intel Pentium D is an example of such a cache design [2]. For large scale CMPs, processors with mesh interconnects have been in production, e.g., the Tile64Pro from Tilera Corporation. These cores are called tiles. Each tile has private L1+L2 caches and an interconnect router. In a mesh interconnect a tile is connected to its neighbors using a high bandwidth network. The frequency of tile-to-tile transfer typically matches the processor frequency. The routing can also be software controlled and message passing between cores is supported.

3. Directory-based caches: A hybrid of the two schemes has also been proposed. The tile based architecture supports a directory-based coherence protocol. A directory entry contains the tag of a cache line and its coherency status. The directory entry also holds an array of bits with each bit representing a tile. A bit value of 1 indicates that the corresponding tile contains a copy of the cache line. There is only one directory entry for each cache line and thus it has to be at a fixed location. Early designs suggested these to be located at the memory controller assuming an on-chip memory controller. But this

would generate a lot of traffic over the interconnect. Like a single shared cache, a single directory becomes an unscalable structure. This motivated the design of a distributed directory. A directory is distributed among tiles. Each cache line has a unique home tile. The home tile of a cache line is dereferenced by certain bits in address similar to the mapping of a cache line to a cache set. Each tile now contains an additional tag array and a corresponding directory entry array. Please note that the home tile can be a remote tile. On a cache miss, a requester tile sends a message to the home tile. Based on the coherency status, additional messages are generated by the home tile to other tiles that hold a copy of the cache line. Those tiles then respond accordingly to the home tile and the requester tile. The Tilera architecture can be configured to support a Private L1+L2 with message passing or a Private L1+L2 with a distributed shared L3 cache.

There has been a substantial amount of research that has improved the average case execution of high performance applications. This has helped multicores to become ubiquitous in several devices that are used on a daily basis, from laptops, cellular phones to supercomputers. However, feasibility of multicores in real-time systems is still under investigation for different cache hierarchies, interconnects and memory interfaces. The following subsections present those reasons in detail.

1.3 Multicore Real-time Systems

As the number of tasks increases, the processing demand in terms of overall processor utilization keeps increasing. Thus, a complex system like an automotive subsystem that increases the number of tasks with every new release puts more pressure on computing resources. As discussed before, due to high power leakage with high frequencies, multicores support the execution of multi-programmed workloads. This trend already extends to embedded designs with heterogeneous multicores (*e.g.*, for cell phones) and also homogeneous multicores at a larger scale [6]. Thus, it is evident that multicores will become ubiquitous over the next few years.

1.3.1 Scheduling Real-time Tasks on Multicores

As discussed before, schedulability of a real-time task set is dependent upon the predictability of WCET on a given architecture under a given scheduling policy. In the past decade, several multi-processor scheduling policies have been proposed. These scheduling policies can be widely be characterized as:

1. Static Partitioned Scheduling: Under this type of scheduling, tasks are assigned to a core and are not allowed to migrate [30, 18]. Each core runs an independent scheduler. Though optimal assignment of tasks to partitions has been proven to be an NP-complete problem,

simple heuristics like first fit decreasing (FFD) have been found to be quite efficient under task partitioning[44]. In Chapter 4, we will be discussing this algorithm in detail.

2. Global Scheduling: The major drawback with Static Partitioning is that it is unable to distribute tasks that have high density/utilization, i.e., tasks with a utilization greater than 0.5. To address this limitation, global scheduling techniques have been and are being proposed. The fundamental premise of these techniques is that tasks may migrate between cores. Some global scheduling techniques suffer from reduced system utilization [30, 50]. A whole class of fair scheduling algorithms have recently been developed for multicore scheduling that give better utilization bounds [15, 59, 10, 11, 84, 14]. However, under global/fair scheduling it is hard to achieve a tighter bound on the number task migrations. Therefore, these algorithms were proposed as non-work-conserving schedules. A non-work-conserving schedule is one where the computing resource remains idle even though there are tasks ready to be scheduled.

3. Semi-Partitioned Scheduling: Another way of limiting the number of task migrations is to use Semi-partitioned scheduling. Here, some of the tasks are statically scheduled while others are allowed to migrate. In such a system, there is a hierarchy of schedulers. There may be a single global scheduler that is invoked at certain events to allocate migratable tasks on different cores. While each core runs an independent scheduler that is responsible for scheduling allocated tasks on a core only. The original work on semi-partitioning was directed at soft real-time systems [12]. Subsequently, several algorithms have been proposed for hard real-time systems [32, 37].

1.3.2 Challenges with Multicore Schedulability

As mentioned in Section 1.3.1, the basic assumption of partitioned/semi-partitioned scheduling is that the tasks are migratable. However, the schedulability analysis for these scheduling policies considers the overhead of task migration to be either zero or some constant value. In order to assume zero migration costs, these algorithms assume processor architectures without any caches, i.e., a memory reference will cause a memory access regardless of the migration. Such an assumption might render tasks unschedulable due to a prohibitively high WCET. Research work conducted upon using caches in multicores has resorted to shared caches. This simplifies the cache analysis as all the tasks are assumed to be sharing a single cache resource. This allows the tasks to have lower utilizations. Pessimistic migration costs are within the allowable range such that the task sets are still schedulable. However, as we have explained, shared caches are unscalable. With increasing processing requirement in every computing domain, including real-time systems, scalable multicore architectures are bound to become ubiquitous. These scalable architectures use private caches. Not much research has been conducted on the usage of private

caches in multicores real-time systems. These assumptions for multicore scheduling algorithms may lead to an inaccurate analysis. Also, static partitioning on scalable multi-processors with private caches have not been studied.

1.4 Problem Statements and Contributions

This dissertation assesses if scalable multicore architectures with private caches are suitable for real-time systems. The following are the issues presented and solutions proposed in this document:

1. In Chapter 2, we present a push-assisted cache migration scheme [77]. We illustrate that task migration actually (a) reduces timing predictability for scalable multicores due to cache warm-up overheads while (b) increasing traffic on the network-on-chip (NoC) interconnect. This chapter puts forth a fundamentally new approach to increase the timing predictability of multicore architectures aimed at task migration. A task migration between two cores imposes cache warm-up overheads on the migration target, which can lead to missed deadlines for tight real-time schedules. We propose novel micro-architectural support to migrate cache lines. We present two micro-architectural schemes (a) Whole Cache Migration (WCM) and (b) Regional Cache Migration (RCM). We illustrate that our mechanisms are capable of providing predictable cache migration over a conventional Prefetch Task Model (PTM). Our experimental results show that RCM reduces overheads considerably. This overhead can be steered to fill idle slots in the schedule, i.e., they do not contribute to the execution time of the migrated task. This migration overhead can also be used by schedulers to select migratable tasks during semi-partitioning of task sets [80].

2. In Chapter 3, we illustrate the idea of migrating locked cache lines. Locking cache lines in hard real-time systems is a common means of achieving predictability of cache access behavior and tightening as well as reducing worst-case execution time, especially in a multi-tasking environment. Tasks with locked cache lines need to proactively migrate these lines before the next invocation of the task. Otherwise, cache locking on multicore architectures becomes useless as predictability is compromised. This chapter proposes enhancements to hardware-based push-assisted cache migration as a means to retain locks on cache lines across migrations [74]. We improve our RCM and WCM techniques with pipelined cache migration schemes that significantly reduce the cache migration delay of an individual task migration. We also present parallel migration mechanism that maximizes the bus utilization in the midst of multiple cache migrations. We further provide deterministic migration delay bounds that help the scheduler decide which migration

technique(s) to utilize to relocate a single or multiple tasks.

3. In Chapter 4, we further the concept of locked caches on scalable multicore architectures by analyzing its impact on static task partitioning algorithms. In shared cache architectures, a single resource is shared among *all* the tasks. The objective of task partitioning on a shared cache architecture is to reduce response time of tasks. With scalable architectures, there are abundant computing and cache resources and the focus shifts toward efficiently utilizing computing resources. Also, in scalable cache architectures with private caches, conflicts exist only among the tasks scheduled on one core. This calls for a cache-aware allocation of tasks onto cores. Here, we propose a novel variant of the cache-unaware First Fit Decreasing (FFD) algorithm called the Naive locked First Fit Decreasing (NFFD) policy. We propose two cache-aware static scheduling schemes: (1) Greedy First Fit Decreasing (GFFD) and (2) Colored First Fit Decreasing (CoFFD) for task sets where tasks do not have intra-task conflicts among locked regions (Scenario A) [75]. NFFD is capable of scheduling high utilization task sets that FFD cannot schedule. Experiments also show that CoFFD consistently outperforms GFFD resulting in a lower number of cores and lower system utilization. For a more generic case where tasks have intra-task conflicts, we split task partitioning into two phases: task selection and task allocation (Scenario B) [76]. Instead of resolving conflicts at a global level, these algorithms resolve conflicts among regions while allocating a task onto a core and perform unlocking at the region level instead of the task level. We show that a combination of our novel Dynamic Ordering (task selection) with Chaitin's Coloring (task allocation) scheme reduces the number of cores required considerably over a basic scheme (combination of Monotone Ordering and Regional FFD). Regional unlocking allows this scheme to outperform CoFFD for medium utilization ($0.40 > \textit{locked task utilization} \geq 0.25$) task sets from Scenario A. However, CoFFD performs better than any other scheme for high utilization ($0.55 > \textit{locked task utilization} \geq 0.40$) task sets from Scenario A.

1.5 Hypothesis

Advancement in processor technology provides an opportunity to avail large numbers of computational resources in the form scalable multiprocessors. With the ever increasing computational demand of real-time systems, we foresee the deployment of such systems on scalable multiprocessors. The primary limitation in multiprocessor real-time scheduling is that they are agnostic of private cache organization. Such cache organization results in:

1. Unpredictability of task migration costs in global/semi-partitioned scheduling; and
2. Ineffective use of computational resources during static partitioning.

We attempt to address these limitation in this dissertation. Hence, the hypothesis of this dissertation is the following:

Multiprocessor real-time scheduling algorithms can more effectively utilize scalable multicore platforms when global/semi-partitioned scheduling techniques predict cache migration costs accurately and static partitioning considers the impact of private caches. Deployment of locked caches and hardware support for cache migration on scalable multi-processors can enable more predictable and efficient multiprocessor real-time scheduling.

1.6 Organization

Chapter 2 analyzes the impact of task migration and presents a hardware mechanism for migrating cache lines from one core to another during the slack available within the schedule. Chapter 3 defines the problem of task migration for hard real-time systems using locked cache lines and presents mechanisms to handle individual/multiple cache migrations efficiently. Chapter 4 presents static partitioning mechanisms for private locked caches on scalable multiprocessor architectures. Chapter 5 discusses open problems for future work and presents the conclusions drawn from this work.

Chapter 2

Push-Assisted Migration of Real-Time Tasks in Multicore Processors

2.1 Timing Predictability and Micro-architectural Challenges

Multicores have been a research topic in micro-architecture, which has led to rapid industry adoption (Intel Core, AMD Barcelona, Sun Niagara, IBM Power) and even advanced scalable multicore designs [5, 6] due to the frequency wall. Past research has focused on improving parallelization strategies [23, 95, 73, 90, 39] to increase average performance and to provide scalability on the memory path [22, 57, 46, 33, 87, 56], including capacity-oriented schemes that scavenge unused neighboring cache lines [34, 24, 26, 101]. Yet, timing predictability actually deteriorates in multicores. Thus the results and inferences drawn from contemporary high-performance computer architecture research are not directly applicable to real-time systems. High performance applications do not have deadlines so that deviations from average performance do not render a system dysfunctional. Real-time systems largely consist of multiple tasks, often periodically scheduled, such that each task is able to meet its deadline. Jobs of a periodic task are then released at regular intervals to obtain sensor readings, perform processing actions of often control-theoretic nature and then engage in actuator actions. Jobs of real-time tasks systems have to complete by their deadline. These jobs require accurate timing predictability to ensure deadlines can be met, which could easily be affected by minor deviations on complex multicore architectures that dilate execution. The dilation of interest in this work is due to migration of a task or an application between processor cores. Task migrations in contemporary architectures are followed by cache warm-ups on the migration target. Cache warm-ups are usually ignored or have insignificant impact for high-performance computing. For

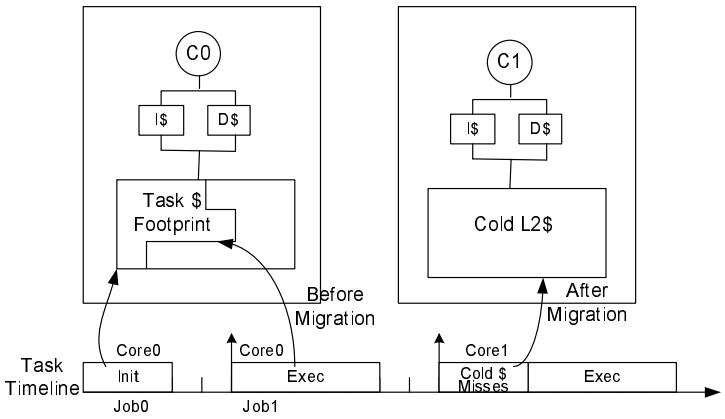
example, architectural simulations make use of so-called simpoints, i.e., simulation points that delineate program phases. The idea is to reach a stable architectural state at which average case can be studied [40]. In contrast, cache warm-ups have considerable impact on real-time systems because any dilation in execution time of real-time tasks could lead to system failure.

Multicore research is actively working towards providing scalable multicores with large L2 caches. Distributed shared L2 caches, private L2 caches, and hybrids involving both designs are being actively compared by the high-performance research community. The size of such L2 caches often provides real-time systems with enough resources to fit all tasks within the L2. Consequently, tighter WCETs help tasks to become schedulable by allowing them to meet their deadlines. However, task migrations may cause deadlines to be missed due to dilations in execution time, *e.g.*, due to cache warm-up. Thus task migration on multicore architectures deters the predictability of the WCET of real-time tasks significantly. In order to evaluate the impact of cache warm-ups without any extraneous effects, we simulated a multicore architecture. Our experimental results on the simulated multicore architecture over a set of WCET benchmarks show that task migration can dilate the execution time anywhere from less than a percent to up to 56.6%.

2.2 Introduction

We propose a novel software mechanism to highlight the impact of pro-active migration of cache lines in order to increase the predictability of real time tasks. We follow it up with a novel hardware/software mechanism to dramatically increase predictability in the presence of migration by providing micro-architectural support for task migration that is further suitable for static timing analysis, an area not covered by previous work on WCET analysis for multi-level caches in multicores and shared-memory multiprocessors [60, 98, 42]. In this work we are assuming a multicore architecture with private L1 and L2 caches as explain in the previous chapter. Our solution is based upon transfer of L2 cache lines of a migrated task from a source core to a target core over the coherence interconnect. This migration is initiated between a task's job completion and the next job's invocation, preferably in idle slots of the real-time schedule. Research has been conducted to keep relative deadlines much shorter than periods for tasks in control systems to reduce variation in response time [13, 49]. This leads to composition of tasks with high density but low utilization as shown in Figure 2.2 that also leave idle slots due to slack available in the system. We propose a software scheme, where the scheduler launches a task at the target core that prefetches cache lines. Thus this scheme is called Pre-fetch Task Migration (PTM). It reduces the time dilation to which the task is subjected on the target core as a result of its migration. However, it is not able to prefetch instruction cache lines and imposes high overhead that cannot be reduced. Hence, we investigate a pure hardware scheme,

Figure 2.1: Task Execution amid Task Migration



called Whole Cache Migration (WCM), that restores the whole cache context of the task onto the target core. However, this mechanism involves high overhead, leaving not enough time to migrate all lines before the next job invocation on the target core. Hence, we propose a software assisted hardware scheme, called Regional Cache Migration (RCM), that allows the developer to transfer knowledge about the memory space to be migrated to the cache controller and then initiate a push operation (instead of a pull) of the lines to the target. Thus the cache controller can identify a subset of cache lines subject to migration, which reduces migration overhead. RCM reduces the overhead for all tested benchmarks, for some by an order of magnitude, while constraining the dilation caused by task migration close to the dilation experienced by a task under WCM. The objective of this work is to provide hardware support that enables predictability and reduction in cache-related migration delay (CRMD).

2.3 Problem Analysis

This section presents the performance impact of migration of tasks on predictability of their WCET.

Experimental Architecture: Our experimental model assumes a CMP architecture with private L1+L2 caches. These processor cores are connected via a bus-based interconnect. This choice was made to exhibit similar properties to contemporary tile-based architectures with the assumption of a QoS-based interconnect [48]. It then excludes the complexity introduced by the interconnects and uncovers the predictability challenge caused by cache misses only. The simulated environment is composed of a two-core CMP. Each core is composed of an in-order processor with a private 32KB L1 cache and a private 2MB L2 cache. The standard MESI coherence protocol is used to maintain coherence across L2 Caches. SESC [72], an event driven

simulator is used to design this architecture.

Experimental Model: Figure 2.1 shows the experiment methodology of this work. We assume the real-time applications to be composed periodically invoked real-time tasks. The timeline shows the instances of execution of a task (jobs). Job 0 initializes the task at core 0. The initialization warms the local cache of core 0 which is shown as the Task Cache Footprint. In our experimental model, we treat the first job (job 0) of a task to be part of an initialization phase where the cache is warmed up. The next release of the task, job 1, is invoked on the same core while the cache is warm. As the task footprint is within the local cache, job 1 does not incur any L2 cache misses. In a uniprocessor scenario, if the footprints of all tasks fit within the local cache, then static timing analysis for each load can use the L2 cache hit latency instead of the memory latency as the upper bound. This approach gives a much tighter bound for the WCET of a task. However, in a multicore environment, this WCET does not hold anymore as subsequent jobs of the task might be scheduled on a different core as shown in Figure 2.1. Job 2 executes on core 1 whose local L2 does not contain the task footprint. Hence, job 2 spends a substantial amount of time in bringing the cache lines from core 0 to core 1. However, the extent of impact can be unbounded. This is the subject of the analysis of our experimental study.

Table 2.1: WCET Benchmarks

Benchmark Name	Functionality	Algorithmic Complexity
bs	Binary search on a given array of records	$O(logn)$
crc	Cyclic redundancy check computation on 40 bytes of data	$O(n)$
cnt	Counts non-negative numbers in a matrix	$O(n)$
stats	Statistics program uses floating point operations	$O(n)$
bsort	Bubble sort program	$O(n^2)$
matmult	Matrix multiplication of two matrices	$O(n^3)$

Impact of Migration on WCET: Experiments were performed over a subset of the WCET benchmarks from Malardalen [7]. Table 2.1 lists those benchmarks along with their functionality and their algorithmic complexity. Each of these benchmarks were executed as described in Experimental Model. The results of the experiments as listed in Table 2.2. The first column lists the name of the benchmark and the second column shows the size of the dataset in terms of kilobytes. The third and fourth columns provide the execution time in cycles for tasks before migration (with warm caches), and after migration (with cold caches), respectively, and the fifth column expresses the increase in execution time due to migration as a

Table 2.2: Task migration & dilation in WCET

Benchmark Name	Dataset Size (kbytes)	Before Migration (cycles)	After Migration (cyles)	WCET Dilation (%)
bs	1024	1595	2497	56.55
crc	1	6972	8292	18.93
cnt	308	2014528	2309082	14.62
stats	781	11676261	12428101	6.43
srt	2	4002752	4006507	0.09
matmult	2.6	954106	963203	0.95

Table 2.3: Matrix multiplication WCET v/s matrix size

Matrix Dimension	Before Migration (cycles)	After Migration (cyles)	WCET Dilation (%)
15x15	189116	192231	1.64
30x30	1456796	1468135	0.78
60x60	11448656	11484769	0.31

percentage. The results show that the increase in execution time over WCET due to migration varies from 56.6 percent for binary search to less than a percent for matrix multiplication. However, it is important to understand the characteristics of the tasks that are more susceptible to dramatic changes in execution time. One such characteristic is the algorithmic complexity. Our experiments show that accurate knowledge of algorithmic complexity of a real-time task can help to identify tasks that are more affected by the migration than others. Benchmarks whose complexity is $O(n)$ tend to get affected by migration the most. This is because the critical operations (compare, multiply etc.) and number of load operations grow proportionally with the number of data elements. Since L2 miss latency is much higher than any CPU operation, a substantial number of cold misses can have a significant impact on the execution time of a task. This is true for the crc and cnt benchmarks, shown in Table 2.2. However, the stats benchmark, which contains algorithms with complexity $O(n)$, shows lesser impact than cnt and crc. There are two reasons for this behavior: (a) Stats has floating point arithmetic that takes a significantly larger number of cycles than integer arithmetic. Hence, the ratio between L2 miss latency and critical floating point operations is significantly reduced, and (b) Stats is composed of multiple algorithms of linear complexity that reuse the same dataset. Benchmarks matrix multiplication and sort are of complexity $O(n^3)$ and $O(n^2)$, respectively. The increase in their execution cycles due to task migration is less than one percent due to heavy data reuse.

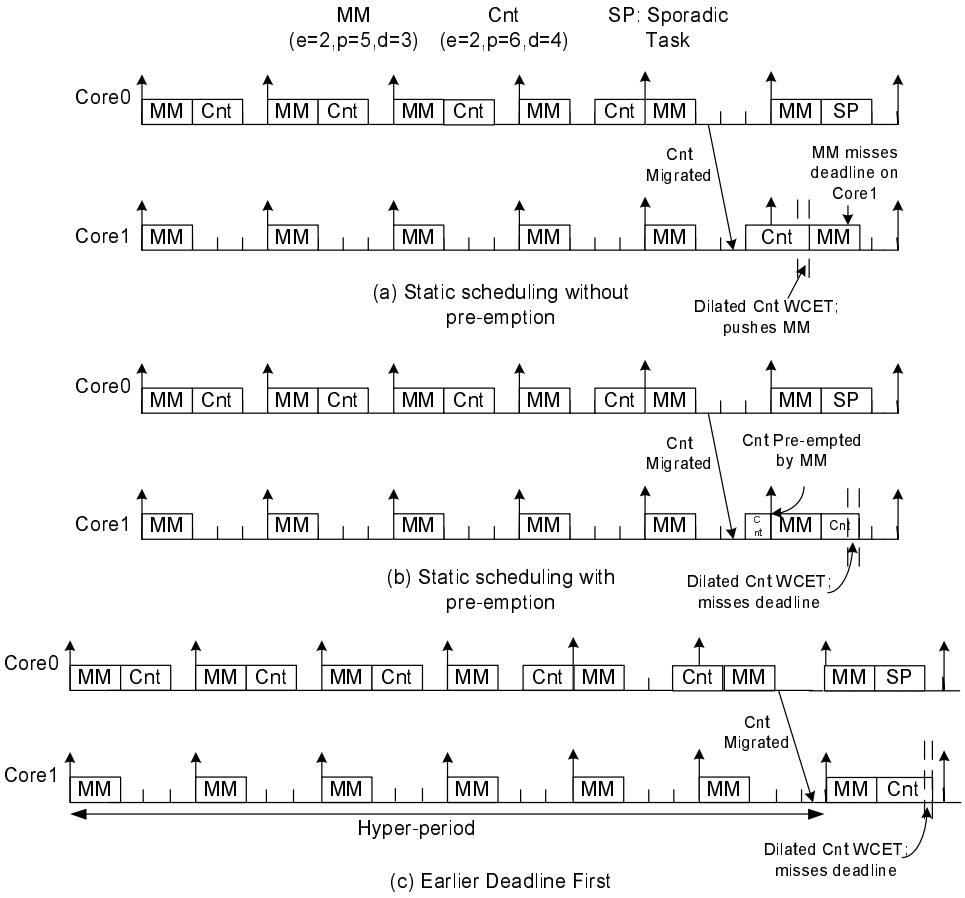
Table 2.4: Bubble Sort WCET v/s Number of Elements

Array Elements	Before Migration (cycles)	After Migration (cyles)	WCET Dilation (%)
50	161160	162400	0.8
100	641155	643040	0.29
250	4002752	4006507	0.09

We conducted experiments to obtain the execution cycles for different sizes of data sets for Matrix Multiplication and Sort. Tables 2.3 and 2.4 show the results for Matrix Multiplication and Sort, respectively. The first column gives the size of the data set, the second and third columns list the execution time that the tasks takes to complete before migration (with warm caches) and after migration (with cold caches), respectively, and the fourth expresses the increase in execution time due to migration as a percentage. The results show that even these benchmarks experience greater impact when the data set is small.

Task migration and impact on schedulability: Since the impact of task migration can be quite large, it can adversely affect the schedulability of tasks. Using a simple task set, we demonstrate that a single task migration can force the migrated task or other tasks within the task set to miss their deadlines under both static and dynamic scheduling. We chose matmult and Cnt, termed as MM and Cnt, respectively, to construct a task set as shown in Figure 2.2. We selected matmult for its high complexity among the benchmarks studied (implies high data re-use). We chose Cnt as it was among the benchmarks whose execution time increased by 14% due to migration. We then examined the impact of task migration using both the (a) static and (b) dynamic scheduling schemes. Figure 2.2(a) shows a static schedule across two cores. On core 0, a task set composed of matmult and cnt is deployed. On core 1, a task set composed of only matmult is running. The scheduler grants matmult higher static priority. This system is non-preemptive. The system progresses as expected until the scheduler experiences a request from a sporadic task to be scheduled on Core 0. There may be several reasons for a task to be scheduled on a specific core. For example, the core may be connected to a sensor or an actuator. Interrupts triggered by a device can invoke a sporadic task to execute on core 0. As a consequence, at some point on the timeline the scheduler decides to migrate task cnt to core 1. This decision is marked by an arrow from core 0 timeline to core 1. This allows cnt to be released on core 1 one unit before the release time of matmult on core 1. Since the system in Figure 2.2 is non-preemptive cnt executes to completion. However, due to a cold cache encountered by cnt on core 1, there is a 14% dilation in the execution time of cnt on core 1. This delays the release of matmult that misses its deadline as shown in Figure 2.2(a).

Figure 2.2: Task migration and scheduling anomalies



One might argue that if the system was preemptive then matmult would have preempted the migrated task and met its deadline. However, Figure 2.2(b) exhibits that if preemptive scheduling allows matmult to meet its deadline on core 1, it results in a deadline miss when cnt resumes execution after matmult's completion. This is again attributed to the extra latency added to the execution of cnt while encountering cold cache misses on core 1. Next, we investigate how a dynamic scheduling algorithm such as Earliest Deadline First (EDF) behaves in the wake of task migration. First, we confirmed whether the task set is schedulable under EDF. In order to accomplish that, we computed the utilization and the density of task set running cnt and matmult on core 0. The utilization of the task set is less than one but the density is greater than one. Thus we assessed the schedulability of the task set on a per-job level within the hyperperiod. The task set is schedulable as cnt and matmult meet all their deadlines within their hyperperiod as shown in Figure 2.2(c). However, towards the end of hyperperiod, a sporadic task forces the scheduler to migrate cnt from core 0 to core 1 for the next hyperperiod. As

per EDF, matmult's job with higher priority gets to execute before cnt. However, as in static scheduling, cnt fails to meet its deadline under EDF.

Hence we obtain the following:

1. Cold cache misses incurred by a migrated task substantially dilate the WCET of a task. Such dilation may prevent the migrated task and other tasks from meeting their deadlines.

2. Theoretical and simulated schedules show that the impact of task migration on schedulability is not restricted to any particular type of scheduling algorithm (static or dynamic) or the system aspect (preemptive or non-preemptive).

3. Aspects like algorithmic complexity of the program, size of the data set, data set reuse and latency of critical operations performed by the task are important factors that can help in gauging the impact of task migration on WCET.

2.4 Proposed Solution

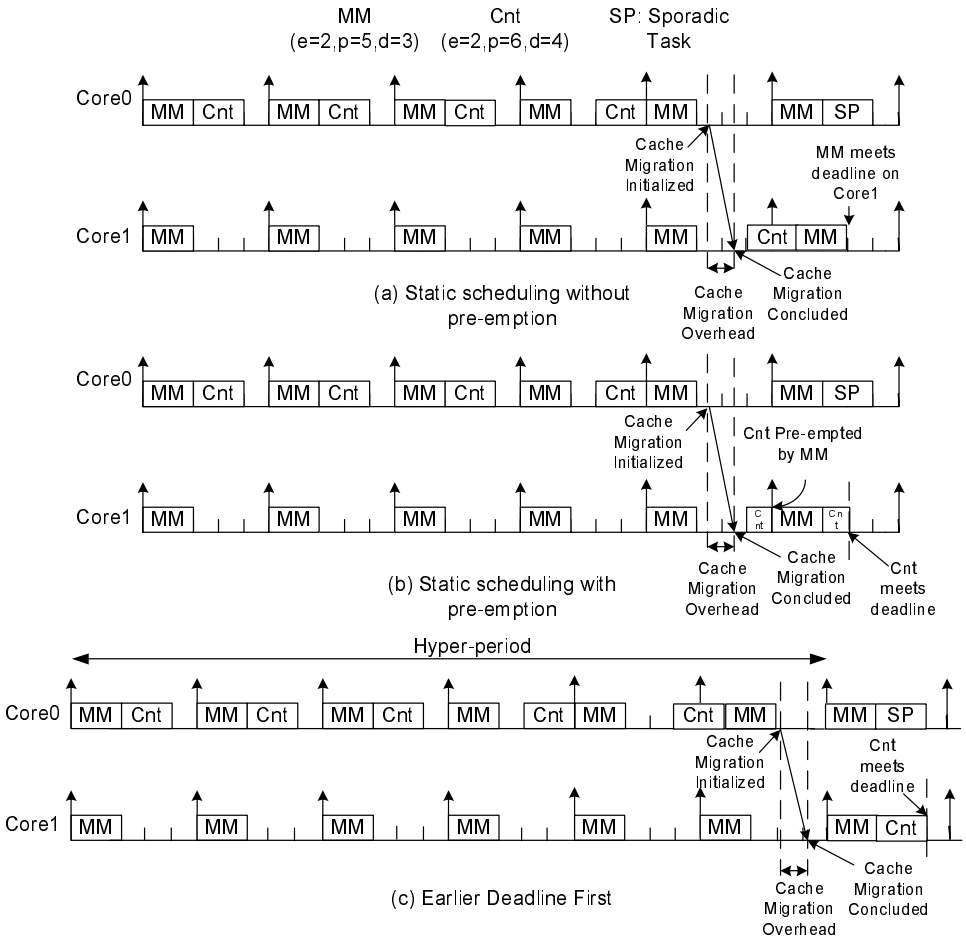
In this section, we present the proposed solution at a high level and discuss any assumptions made for the experimental model. As shown in the previous section, task migration can significantly dilate the WCET of a task.

The architecture of contemporary multicore designs is completely ignorant of task migration. A simple experimental scenario was constructed (see Figure 2.2). It illustrates this inability inherent to task migration leading to a deadline failure of scheduled tasks. However, since scheduler is aware of the task migrations it can also initiate cache migrations from source core to target core. The same task set is modeled with our proposed solution shown in Figure 2.3.

The key difference in these scenarios is that when the scheduler decides to migrate a task, it actively forces the migration of cache lines from source (core 0) to target (core 1). Such active movement of cache content comes at the cost of up-front migration overhead in contrast to the traditional delayed overhead at the next job activation. In presently discussed scheme, this overhead does not contribute to the execution time of the task's next job because cache migration takes place *prior* to this next job's invocation at the target (core 1). In the depicted case, cache migration also *completes before* the next job of Cnt is invoked at the target (core 1). Such a scheme allows a scheduler to decide dynamically if a task is migratable or not. Our proposed migration techniques can also be used to facilitate predictable Cache-Related Migration Delay (CRMD) that can further be used for determining schedulability statically [80]. Here, we investigate two schemes: The **Pre-fetch Task Model** and the **Push-assisted Cache Migration Model**. These schemes will be discussed in the following sections.

In this chapter, we use an experimental model with the following underlying assumptions. We assume separate communication channels for processor-to-memory and processor-

Figure 2.3: Task Migration Coupled with Cache Migration



to-processor traffic on a network-on-chip interconnect, like the ones supported in the TilePro64 (Tile Dynamic Network (TDN) and Memory Dynamic Network (MDN)) [6]. We also assume the processor-to-processor communication to support some form of QoS that bounds the processor-to-processor delay. This allows us to gauge the potential of cache migration in terms of the cache footprint of tasks and their execution complexity. We also assume that the caches are physically index and physically tagged. To understand the effect of migration on individual tasks, we evaluate single task migration. Our experimental model requires that cache migration take place during idle slots for both source and target cores (or any other bus activity). Such isolation has been proposed for systems that alternate between computational and communication phases [64]. In our recent publication, we have proposed a semi-partitioned scheduling algorithm that also shows the availability of slack [80]. If migration were to take place during a task's execution on any of the cores, the network-on-chip mentioned earlier isolates memory

requests from processor-to-processor communication. Furthermore, this chapter does not focus on inter-task cache conflicts as such conflicts are orthogonal to this study. In our recent publication, we have proposed a semi-partitioned scheduling algorithm that resolves cache conflicts and exploits our cache migration mechanism to improve the response time for real-time tasks [80]. We will discuss this work briefly in Section 2.8.

2.5 Migration Models

In order to explain the migration model, we use "source" and "target" to refer to the core/cache where the task is currently running and where the scheduler migrates the task to before the next invocation, respectively.

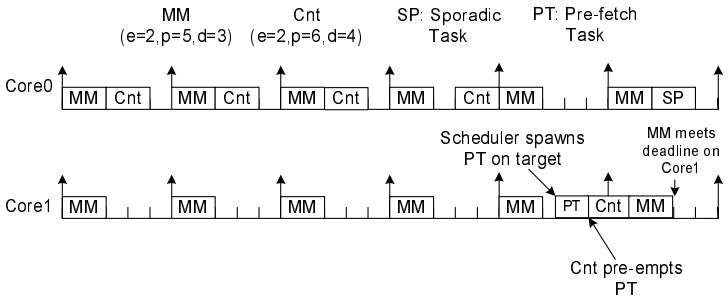
2.5.1 The Conventional Pull Model

Task migrations are scheduler events. Contemporary processors do not have any support to notify the cores of such events. The source core stops executing the task while the target core starts executing the same. Hence, when a task is migrated by the operating system from a source core to a target core, it suffers from L1 (and L2) cold misses as it warms up its working set. These misses in the L1+L2 caches of the target core are resolved one at a time by a shared L3 cache or the L1/L2 caches of the source core for MESI and MOESI coherence protocols, respectively. Hence, each cache miss results in multiple coherence messages being sent over the on-chip interconnect (bus or point-to-point) and competes with memory references from other cores. Initial read references on the target core force the use of coherence messages resulting in transitions into a shared cache state on both ends while subsequent writes inflict additional coherence traffic for invalidations. This is termed a *pull-based scheme* as it reacts to misses one at a time on demand before pulling data over the NoC interconnect.

2.5.2 The Pre-fetch Task Model(PTM)

The pre-fetch task model is a software scheme. Each task registers a set of regions that define the anticipated cache footprint. Each region is defined by the starting address and size of region in terms of bytes. When the scheduler decides on migrating a task, it consults the region specifications and spawns a pre-fetch task on the target as shown in Figure 2.4. Since this is an execution driven mechanism and occupies processor, the pre-fetch task has the lowest priority at the target core. Thus it is active only when there is an idle slot available at the target. A pre-fetch task issues a sequence of read requests at the granularity of a cache line, which on completion warms both L1 data and L2 unified caches. On completion of this task, all the cache lines within the critical regions of a migrated task have been pre-fetched. The prefetch

Figure 2.4: Task Migration with Pre-fetch Task support



task is terminated at the target core once the migrated task resumes execution at target core regardless of the state of the pre-fetch task. Since it has the lowest priority, it is descheduled by tasks that run prior to the invocation of the migrated task as shown in Figure 2.4 where Cnt pre-empts the pre-fetch task PT.

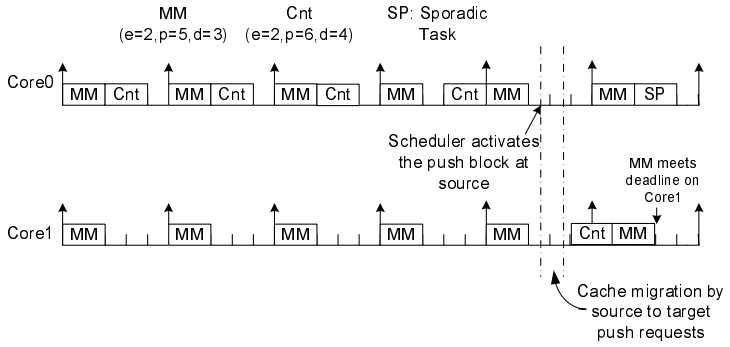
A software scheme like the Pre-fetch Task Model seems simple to implement but descheduling of the pre-fetch task prevents any prefetching to take place while tasks other than the migrated tasks resume execution. While a task executes on a core, it consumes processor L1 cache resources extensively. However, L2 cache resources may have idle cycles available. Thus a hardware solution may use the idle cycles at L2 cache controller to prefetch lines. However, a pre-fetch model is eventually a pull model with requests actively issued by the target rather than passively. Therefore, such a scheme, irrespective of a software or hardware solution, will have the disadvantages that apply to any pull model scheme. A pull model is not aware of the cache lines that are available at the source. If a line is not in the source cache then the read request is futile. Such unnecessary read requests dilate the time required for pre-fetching to complete and cause additional traffic on the NoC. Additionally, a pull based scheme will always issue requests for one cache line at a time. This is because it is not aware of locality of the cache lines present at the source. Hence, there is no scope for improvement in reducing the overhead of cache migration. Thus we describe a hardware scheme in the following section, which exhibits the following improvements over PTM:

1. It delivers tighter WCET of the migrated tasks,
2. It eliminates any futile transactions for cache lines that do not exist at sources,
3. It has the capability to use idle L2 cache cycles at source and target while the processors is busy serving other tasks,
4. It Parallelizes the transactions to reduce migration overhead, and

5. It also provides the opportunity to reduce the coherency related transaction after migration

2.5.3 The Push-assisted Cache Migration Model

Figure 2.5: Task Migration with Push-assisted Cache Migration support



In a push model, memory requests are initiated by the source core in order to warm up the target cache instead of demand-driven requests issued by the target. We propose such a novel push-based hardware mechanism initiated by the Operating System on the source core that transfers warm cache lines to the target core. Such movement is proactive in the sense that it is initiated prior to task reactivation on the target core and continues in parallel once this task resumes execution on the target. The present implementation exploits the slack time that a task has prior to the next invocation of the task on the target. The sequence of events in the process of cache migration then is as follows.

1. The task notifies the OS of its completion.

2. The OS invokes the scheduler routine to determine target core for the task.

3. The OS initiates cache migration by setting the "target register" within the special-purpose hardware, called the push block.

4. The push block, in conjunction with the source cache controller, migrates the cache contents by placing memory requests, called the push requests, for each migrated cache line. This is shown in Figure 2.5.

The Push Block

The push block has the responsibility to identify migratable lines. Only a subset of valid cache lines are migratable. A migratable line has to be associated with the task being migrated. In case all the valid lines are migrated, it has the following drawbacks:

1. The push block will take a longer time to migrate the task since it will require a larger number of cache line migrations.

2. The target will experience cache pollution because of the displacement of the valid lines of some other task running on it.

3. On re-invocation of remaining tasks at the source, the system bus experiences a large number of coherence messages.

Thus to identify a migratable line, the push block may need support from the cache or the OS. We will discuss the implementation requirements of the push block in the discussion of the variants of the push model.

Whole Cache Migration (WCM)

WCM replicates the cache context of the migrated task at the target such that after re-invocation, the migrated task's execution behaves as though the task had not migrated. It is a complete hardware mechanism where the push block scans each and every cache block in order to find the migratable lines. However, the push block requires the cache lines to hold a task identifier (task ID) in order to prevent it from migrating the valid lines of other dormant tasks as discussed in the section above. This mechanism has the advantage of replicating all the cache contents of the task at the target. However, in case the cache footprint of the task is very small as compared to the size of the L2 cache, a lot of cycles are wasted while accessing non-migratable cache lines.

Regional Cache Migration (RCM)

RCM borrows the concept from PTM of allowing developers to specify expected cache footprint of the task, in terms of regions, and uses that information to push cache lines. This can be derived by abstract cache analysis or the user may define a set a locked cache lines as presented in Chapter 3. This reduces the number of cache lines being visited by the push model and also eliminates the requirement for the cache to store task identifiers in order to determine migratable cache lines. In our experimental model, this is accomplished by adding an instruction that allows the programmer to fill a limited number of registers called "region registers" within the push model. Each region register contains a start address and the size of a frequently reused region

expressed in bytes. A programmer can identify contiguous locations of memory that form the majority of the memory footprint by updating the region registers. At the set-up phase tasks save the region register values as part of task's context. When a task is activated on a core, the region registers of the push block at the core are updated as part of updating the task context.

When a migration is triggered, the push block identifies the migratable lines as valid lines belonging to the regions specified by region registers. RCM has the advantage of scanning fewer cache lines than WCM. However, since one can specify only a limited number of regions, this could limit the programmer from identifying the whole cache footprint. Thus some cold cache misses may be retained.

Push Request

We next discuss the integration of the memory request generated by the push model (the push request) with the MESI protocol on a CMP architecture. Once the push block identifies the cache line to be migrated, it issues the push request. Unlike a read/write request, the push request is only referenced by the target cache but never by the memory controller. The push request is issued by the push block. It does not constitute a demand request but rather a write request on behalf of the target made by the source. However, the implementation of a push request may have effects on the cache performance and the bus bandwidth requirement. The push model might find a migratable line in shared, exclusive or modified state. The shared cache lines are read at the source and allocated to the target and initialized with shared state. The task replacing the migrated task may or may not use those lines. If the task uses the lines then it may save on cache misses. In case it does not, a line in shared or invalid state does not change the task's cache performance. However, the choice of implementation might make a difference for migratable lines that are in exclusive or modified state. If a line is in modified state, the re-activation of the task at the target may write to those lines again. Hence, if the lines in the modified or exclusive state are changed to shared state, then there is a possibility that when the task resumes execution at the target, the system experiences a number of invalidation requests posted by the target. However, in case that does not happen, those lines could be used by the tasks that get scheduled on the source core following the migrated task. Therefore, it is worthwhile to analyze the two scenarios for the migration of lines in exclusive or modified state.

1. Modified/Exclusive state to Shared State: change the state to Shared at the source cache, issue a write back to the memory if the state is a modified line, issue a push request to target, allocate a cache line in shared state at target cache.

2. Modified/Exclusive state to Exclusive State: change the state to invalid at source cache, issue a write back to the memory if the state is modified, issue a push request to target, allocate a cache line in exclusive state at target cache.

Bulk Cache Migration (BCM)

The push models discussed in WCM and RCM allow only a single push request to be generated by a source at a time. A push block waits for a push request to complete (i.e. wait for the acknowledgment from the target) before issuing a subsequent push request. This behavior is similar to the read performed in a pull model.

However, as the push requests are generated at the source core, there is an opportunity for overlapping multiple transactions. Subsequent push requests may be initiated without waiting for the acknowledgment from the target to arrive. Thus as the push block places a request on the bus, it can initiate another cache read. This pipelines the processing of multiple push requests and overlaps the overheads caused by multiple transactions. Thus in this work we couple the RCM with BCM to exploit the potential of BCM to decrease the overhead significantly.

2.6 Simulation Platform

Table 2.5: Simulation Parameters

Component	Parameter
Processor Model	in-order
Cache Line Size	32B
L1 I-Cache Size/Associativity	32KB/2-way
L1 D-Cache Size/Associativity	32KB/4-way
L1 hit latency	1 cycle
Replacement Policy	LRU
L2 Cache Size/Associativity	2MB/8-way
L2 Hit Latency	20 cycles
L2 Replacement Policy	LRU
Coherence Protocol	MESI
Push Request Latency	50 cycles
Network Configuration	Bus based
Processor To Processor Delay	2 cycles
External Memory Latency	480 cycles

Our proposed solution requires micro-architectural modifications to a stable bus-based mul-

ticore CMP architecture. We chose SESC [72], a light-weight event driven simulator that implements a stable bus-based CMP supporting the MIPS instruction set. Our base experimental model consists of a multicore CMP architecture where each core has private L1 and L2 caches. This design is very close to the tile-based architecture [101] except that we assume a private L2 cache for each core while they consider each L2 cache to have two sections. In their design, one acts like local L2 while the other is a part of a shared L2 distributed across the cores. In contrast to their work, the focus of our work is towards the timing predictability of the system, for use in real-time systems. Simulator has been modified to provide scheduling capabilities. However, the scheduling and migration overhead is minimal as the scheduler is implemented using emulated system calls. This is because our work does not concentrate upon scheduling overheads.

Contemporary static timing analyzers predict the performance of a task assuming no contention for system resources. There has been research work on providing QoS-like cache partitioning [45] that dynamically changes the performance of the running applications over the period of their execution. The focus of their research is to increase the throughput or to provide fairness to admitted tasks. However, none of the objectives address predictability and not subjected to real-time systems. Static cache partitioning may be a means to reduce the inter-task cache contention. However, it has not been very popular because of potentially waste of critical cache space. Hence, we chose to use a system that keeps the cache contention to the minimum while also providing enhanced predictability.

The system architecture specifications are presented in Table 2.5. A unique parameter in this system is the "push request" latency. A push request incurs latencies of L2 cache access at source and at destination, round trip of push and acknowledgment messages on the bus and delays involved in setting up the request of those messages on the bus. These costs contribute to the aggregate latency of one push request with an assumed cost of 50 cycles.

2.7 Evaluation

2.7.1 Prefetch Task Model

Prefetch Task Migration and Potential to Bound WCET: The evaluation results for the Prefetch Task Model are depicted in Table 2.6. The first column lists the name of the benchmarks. The second, third and fourth columns provide the execution time in cycles for tasks before migration (with warm caches), after migration without PTM and after migration with PTM, respectively, and the fifth column expresses the increase in execution time due to migration (with PTM) as a percentage. These results exhibit the potential of pro-actively warming up the target cache before the migrated task's subsequent invocation. We see that

Table 2.6: Prefetch Task Model & Potential to Bound WCET

Benchmark Name	WCET (Warmed Cache) [cycles]	Exec. Time w/o "PTM " [cycles]	Exec. Time with "PTM" [cycles]	Dilation in WCET (%)	PTM Overhead [cycles]	PTM Overhead vs. WCET (%)
bs	1595	2497	2109	32.22	2001363	125477
crc	6972	8292	7925	13.67	8183	117.36
cnt	2014528	2309082	2015194	0.033	599811	29.77
stats	11676261	12428101	11678209	0.017	1528712	13.09
srt	4002752	4006507	4003610	0.02	6582	0.16
matmult	954106	963203	954496	0.041	13914	1.45

PTM has a significant impact on reducing the dilation in execution time due to migration. However, it is less effective for bs and crc. This is because PTM is capable of prefetching only data which causes L1 instruction cache misses. The benchmarks bs and crc have different characteristics. Benchmark bs has a large memory footprint. However, the number of data accesses are of O(log N), i.e. only a subset of the data is accessed. Thus the number of data accesses is approaching the same order of the number of instruction accesses. Hence, L2 misses on instruction accesses contribute significantly to the dilation in execution time of the migrated task. On the contrary, crc has a very small data footprint that is smaller than the instruction footprint. In this case as well, L2 misses on instruction accesses contribute significantly. Since, the execution time for all the other benchmarks is driven by the data accesses, PTM delivers the desired outcome of approaching the execution time with warmed cache. Thus we can deduce that migrating the instructions become equally important when executing benchmarks that have low data accesses.

PTM and Prefetch Task Overhead: The cache migration scheme hides the latency incurred by making cache lines available on the target core before they are referenced. Hence, it is important to accurately predict the number of cycles required to migrate the cache footprint of the task so that the process of migration completes before the task is invoked at the target. In Table 2.6, the **sixth** column shows the overhead incurred by prefetching, which is the execution time of the prefetch task, and the **seventh** column represents the overhead with respect to the WCET. The migration overhead of the prefetch task is composed of not only cache-to-cache migration but also the instructions executed to generate the prefetch requests. Since the prefetch task references the data region of the migrated task, the overhead is proportional to the data footprint. The overhead is less than 30% of the target WCET for most of the benchmarks. However, for bs and crc, it is very high. The benchmark bs has a large data footprint. Its overhead is high because the cache-to-cache transfer grows by O(N) while the execution cycles scale with O(log N). However, in case of crc, the cache misses incurred while accessing the

footprint of the prefetch task dominates. Hence, the cycles required to execute the prefetch task inflate the overhead and, in comparison to the small WCET of crc, the overhead seems large. Unfortunately, the overhead with prefetching may get worse if the user-specified regions were inaccurate as described in Section 2.5. Also, in the wake of an in-order core, which is prevalent in real-time systems, the requests will be serialized.

2.7.2 Push-assisted Cache Migration Model

Table 2.7: WCM Performance & Overhead

Benchmark Name	Execution Time "WCM" [cycles] [cycles]	Deviation from tight WCET (%)	WCM Overhead [cycles]	WCM Overhead vs. WCET (%)
bs	1842	15.48	1737657	108944
crc	7490	7.43	68617	984
cnt	2014631	0.005	566240	25.28
stats	11677428	0.099	1317796	11.28
srt	4004448	0.042	37035	0.92
matmult	958121	0.42	79543	8.34

Hardware Cache Migration & Potential to Bound WCET: We performed an evaluation on the simulation platform discussed in the previous section. Table 2.7 shows the potential of the push-assisted cache migration scheme in achieving performance close to the tighter WCET bounds for the migrated task. The first column shows the name of the benchmarks. The second and third columns express the execution time of migrated tasks with WCM in cycles and in percentage of deviation from tight WCET, respectively. The whole cache migration scheme maximizes the ability of cache migration in restricting WCET. However, our current implementation of the whole cache migration scheme pushes the lines from one L2 cache to another but does not push it up to the L1 cache of the target core. Due to the L1 misses suffered by the tasks, the results show an extra cost in cycles to complete compared to warmed-up L1+L2 caches. Therefore, there is a deviation experienced by all the benchmarks. However, bs and crc have deviate more from the tighter WCET bound provided by the warmed up cache, which is similar to PTM. We can deduce that L1 cache misses become significant for tasks that have an algorithmic complexity lower than $O(n)$ or tasks that have small data set size with close to linear algorithmic complexity. Data set size alone cannot be considered a sufficient criterion because the execution cycles consumed by computation can be the dominating factor as was the case for matmult and bubblesort. We will compare WCM and PTM along with the discussion

on RCM results. One might consider warming up the L1 cache, too. However, one of the key features of our hardware scheme is to enable tasks to execute concurrently to cache migration with minimum conflict. Since L1 cache accesses are very frequent, contention caused by push requests will be highly undesirable. Hence, we will keep our hardware push model to transfer cache lines among L2 caches only.

WCM and Cache Migration Overhead: The fourth and fifth columns in Table 2.7 shows the amount of overhead that WCM incurs in absolute number of cycles and as a percent of the target WCET, respectively. We find similar behavior to PTM where bs and crc had a considerably large overhead when compared to the target WCET. The remaining benchmarks experience migration overheads of less than 30% over the target WCET.

One of the primary drawbacks of WCM is the scanning of the whole cache to identify migratable lines. If a line is non-migratable, the tags or cache lines are not read. Also, the requests are not placed on the bus. Additionally, subsequent task ID checks only need index and column lines to be activated in an incremental fashion for WCM. Hence, we are assuming that a non-migratable line adds a cycle to the overhead instead of a cache miss latency. However, contemporary L2 cache sizes are in the order of MBs, which could exceed the task's cache footprint by orders of magnitude. Thus tasks that have a very small cache footprint still pay a high cache migration overhead as the overhead is then proportional to the cache size instead of the task's cache footprint. Such undesirable behavior is alleviated by RCM.

RCM vs. WCM: Due to the aforementioned practical issues associated with WCM that were confirmed by simulated performance results, we propose a software-assisted micro-architectural support technique. Real-time applications primarily perform computation on globally visible static buffers since WCET estimates for dynamic memory allocations are difficult to bound and allocations of memory blocks may cause unnecessary conflict misses. The utilization of large local buffers is discouraged because embedded environments have limited stack size that requires applications to conserve memory needs. Thus most embedded applications perform computations over buffers of static length used over the lifetime of the application. Hence, developers tend to have sufficient knowledge about the size of the data set so that each task can be associated with a vector containing the critical regions of the data set. This is explained in detail in Section 2.5.3.

Next, we present RCM results while comparing PTM, WCM and RCM. Table 2.8 compares the execution time of the benchmarks with different cache migration schemes. Each value in the table is a percentage of additional cycles over the WCET required by a migrated job to finish under the respective cache migration scheme. It shows that all the schemes discussed are effective in reducing the dilation in execution of the migrated tasks considerably. The results show that WCM approaches the tighter WCET bound closest for most of the benchmarks. However, PTM performs better for srt and matmult. This is because in these benchmarks,

Table 2.8: Migration Paradigms and Percent Additional Execution over WCET

Benchmark Name	No Cache Migration (%)	PTM (%)	WCM (%)	RCM (Data Only) (%)	RCM (Data + Instr.) (%)
bs	56.55	32.23	15.49	27.52	17.3
crc	18.93	13.67	7.4	14.26	8.34
cnt	14.62	0.033	0.005	0.03	0.02
stats	6.43	0.017	0.009	0.01	0.01
srt	0.09	0.021	0.042	0.058	0.044
matmult	0.95	0.04	0.42	0.44	0.43

the data fits within the L1 cache and has a smaller instruction footprint. Thus the number of cycles saved for L1 data cache hits outweigh the number of cycles spent on L2 cache misses for instructions. However, if a task other than the migrated task executes before the re-invocation of the migrated task, then the probability of lines evicting from L1 cache becomes high. Hence, PTM may lose its advantage over the push-based hardware mechanisms. The fifth and sixth columns exhibit the results for variants of RCM. RCM (Data Only) performs comparable to WCM for most benchmarks except bs and crc. This is because the cache misses caused by instructions become dominant. This underlines the significance of the results in the sixth column showing that, if the developer is able to specify both data and instructions clearly, it can match the effect of WCM. Push-assisted Cache Migration schemes show a significant reduction in execution time for bs too but the dilation in execution time is still exceeding 10%.

Whether or not a task can be migrated is based upon the overhead of migration. This overhead does not contribute to the execution of the application but rather constitutes the overhead for migrating the task's cache footprint such that the performance of the application

Table 2.9: Cache Migration Overhead cycles over WCET [percent]

Benchmark Name	PTM (%)	WCM (%)	RCM (Data Only) (%)	RCM (Data + Instr) (%)	Bulk RCM (Data + Instr) (%)
bs	125477	108944	106833	106868	47238
crc	117	984	21.3	40	18.38
cnt	29.77	25.28	25.29	25.32	11.20
stats	13.09	11.28	10.92	10.93	4.93
srt	0.16	0.92	0.081	0.099	0.045
matmult	1.458	8.34	1.39	1.43	0.64

is as stated in Table 2.8. As discussed earlier, the WCM scheme has a high overhead because it requires references to non-migratable cache lines. We have shown that RCM can make tasks approach the ideal performance obtained by WCM. Hence, if RCM is able to reduce the overhead of cache migration significantly compared to WCM, it may be considered a viable micro-architectural scheme.

This potential of RCM is evaluated against all other schemes in Table 2.9. The first column holds the benchmark names and the other columns show the ratio of the overhead to the WCET bound in percent for the respective cache migration schemes. Before investigating the RCM results, it is important to notice that WCM behaves poorly in terms of overhead against PTM for crc, srt and matmult. PTM has an overhead proportional to the cache footprint of the task while WCM's overhead depends on the cache size, which is significantly larger than the cache footprint for the aforementioned benchmarks. However, WCM has lower overhead than PTM for bs, cnt and stats whose cache footprint is large. This is because PTM has an extra overhead caused by execution of the prefetch task. RCM blends the advantages of PTM and WCM. It eliminates the execution overhead of PTM and the excessive number of look-ups of WCM. Hence, the overhead of RCM is the least among all the schemes for all the benchmarks.

The comparison between the RCM (Data Only) and RCM (Data + Instruction) shows that the developers have to critically choose the variant of RCM. For example, the migration overhead for crc reduces from 984% for WCM to 21.3% for RCM with data regions migrated and 40% for data+instruction region migration. However, the performance of RCM with data+instruction regions migrated closely approaches that of WCM. Thus for benchmarks like cnt and stats, one might choose only data region migrations while for crc, one should give preference to RCM with data+instructions region migrations.

Bulk Cache Migration: One of the key aspects of push-based cache migration scheme is to issue push requests in parallel as described in Section 2.5.3. The last column of Table 2.9 clearly shows that bulk transfer of push requests can reduce the cache migration overhead by more than 50%. Even though a detailed study of NoC is beyond the scope of this work, such bulk transfer promises to maintain low migration overhead even with increased network latency and multi-hop cache to cache transfers. This is because of the pipe-lining of the transactions. This aspect will be studied in detail in Chapter 3.

Cache Migration and Coherence States: Another aspect of overhead is the extra bandwidth requirements that cache migration imposes. This overhead may affect the performance of concurrently running tasks. Table 2.10 compares the requests issued to the bus during cache migration schemes against a conventional scheme where no cache lines are migrated. The second, third and fourth columns report the read, write and push requests issued onto the bus without cache migration, with cache migration for both the data and instruction regions, and with cache migration for both data and instruction region along with their state (shared if

shared at source, exclusive if exclusive or modified at source), respectively. It can be deduced that those tasks accessing their complete data set will have minor or no difference in the total accesses issued with or without cache migration. This is evident among benchmarks like crc, cnt, stats and srt. However, bs, which accesses only *log(n)* elements of its data set (array), suffers a significant bandwidth overhead. The forced migration of the whole data set issues more push requests than the number of read/write requests that the task issues in the absence of cache migration. Matmult has a resultant array that, if not migrated, causes only write misses. Since the developer specifies this resultant array to be migrated, cache migration scheme without *state migration support* migrates those modified cache lines as shared to the target. Thus they not only incur push traffic but also lead to write misses on subsequent writes. A similar behavior can be seen for srt. Hence, such write misses can be avoided by our state migration extension to our cache migration schemes, *e.g.*, when lines in modified or exclusive state at the source are migrated as exclusive lines to target. Thus subsequent writes do not require invalidations that are otherwise required for matmult and srt (third column of Table 2.10). It can be concluded that the push-assisted cache migration scheme imposes only minor pressure on bandwidth unless the data accessed is too small compared to the data set.

Cache migration and Task Migration Decision: Task migrations are performed by a Real-Time Operating System (RTOS) in response to certain events. Currently, an RTOS does not consider the impact of cold cache misses on the execution time of migrated tasks. Our push-based cache migration techniques overlap the cache transfers with slack time available for the task. However, it is not safe to assume that the RTOS can ignore the impact of task migration with the availability of cache migration mechanisms. We argue that an RTOS should utilize knowledge about the task's cache migration overhead and the available slack time in deciding when and if a task should be migrated. Hard real-time systems would require cache migration to complete within the available slack time. On the other hand, soft real-time systems may allow cache migration to continue even after the next instance of the task has started execution

Table 2.10: Cache Migration and Bandwidth Overhead[number of requests]

Benchmark	No Cache Migration			Cache Migration (Data + Instr.)			Cache Migration (Ex + Data + Instr)		
	Push	Read	Write	Push	Read	Write	Push	Read	Write
bs	0	27	1	32780	1	1	32780	1	1
crc	0	39	4	54	0	4	54	0	3
cnt	0	9808	2	9813	1	2	9813	1	2
stats	0	25027	7	25002	27	7	25002	27	6
srt	0	74	65	77	1	65	77	1	2
matmult	0	182	86	263	5	86	263	5	1

at the target. Furthermore, the RTOS may make an optimal decision of which task is to be migrated among a set of possible candidates based upon cache migration overhead and available slack time. While the development of such policies is out of the scope of the current work.

2.8 Related Work

Yan and Zhang have recently proposed techniques to calculate the WCET of tasks in real-time multicore systems [97, 96, 98], and other approaches develop multi-level WCET cache analysis as well [60, 42]. These approaches are limited to shared L2 instruction caches in their bounding of WCET and none of them consider task migration. Choffnes *et al.* propose migration policies for multicore fair-share scheduling in the context of soft real-time systems [27]. Their technique strives to minimize migration costs while ensuring fairness among the tasks by maintaining balanced scheduling queues as new tasks are activated. Li *et al.* discuss migration policies that facilitate efficient operating system scheduling in asymmetric multicore architectures [52, 51]. Their work focuses on fault-and-migrate techniques to handle resource-related faults in heterogeneous cores and does not operate in the context of real-time systems. In contrast, our work focuses on homogeneous cores and strives to improve system utilization by allowing migrations while providing timeliness guarantees for real-time systems. Calandrino *et al.* propose scheduling techniques that account for co-schedulability of tasks with respect to cache behavior [9, 19]. Their approach is based on organizing tasks with the same period into groups of cooperating tasks. While their method improves cache performance in soft real-time systems, they do not specifically address issues related to task migration. Other similar cache-aware scheduling techniques have been developed [35], but they do not target real-time systems and do not address task migration issues.

Eisler *et al.* [34] develop a cache capacity increasing scheme for multicores that scavenges unused neighboring cache lines. They consider "migration" of cache lines amounting to distribution of data in caches while we focus on task migration combined with data migration mechanisms that keep data local to the target core. Acquaviva *et al.* [17, 8] assess the cost of task migration for soft real-time systems. They assume private memory and different operating system instances per core on a low-end processor. In contrast, we assume private caches with a single operating system instance, which more accurately reflects contemporary embedded multicores [6]. Their focus is on task replication and re-creation across different memory spaces while our work focuses on task migration within partly shared, partly private memory spaces.

In a recent publication [80], we use the push-assisted cache migration to bound the CRMD (Cache Related Migration Delay). In this paper, we present a predictable semi-partitioned strategy for scheduling a set of independent hard real-time tasks on homogeneous multicore platforms using cache locking and locked cache migration. Semi-partitioned scheduling strate-

gies form a middle ground between the two extreme approaches, namely global and partitioned scheduling. By making most tasks non-migrating (partitioned), run-time migration overhead is minimized. On the other hand, by allowing some tasks to migrate among cores, schedulability of task sets has been improved. Locked cache migration is used, which makes the migration overhead between job executions more predictable. We also assume that a migratable task occupies one cache way. If caches have k ways, then k-1 ways are used by purely partitioned tasks and one way is used by the migrating task. A migrating task migrates between two cores. In this work, the predicted CRMD is added to the WCET while the slack time on the two cores is used to fit the migrating task. Simulation results demonstrate the effectiveness of our approach in improving task set schedulability over purely partitioned approaches while maintaining real-time predictability of migrating tasks. The results shows an average increase in utilization of 37.31% and an average increase in density of 81.36% compared to purely partitioned task allocation.

2.9 Conclusions

In this Chapter, we identified task migration as a key contributor to unpredictability in determining WCET bounds of real-time tasks on multicore architectures. With larger L2 caches and increasing numbers of processing units, WCET bounds have the potential to become tighter in the future. Static timing analyzers can capitalize on large L2 caches in that, after the initial warm-up of the cache, execution of real-time tasks will become predictable. This work has shown that, in the wake of task migrations, dilation in execution time due to cache warm-up will become significant enough to occasionally prevent real-time tasks from meeting deadlines. It is thus imperative to develop real-time systems capable of tightly bounding — if not eliminating — the impact of task migration. Simulation results on a subset of WCET benchmarks experience a dilation in execution time ranging from of 6% to 56.6% for tasks whose algorithmic complexity does not exceed $O(n)$. Tasks with higher complexity show a significant dilation for small data set sizes.

We consolidate the idea of proactive cache migration as a means to diminish the dilation introduced by the target warm-up overhead using a software technique called PTM. PTM launches low priority prefetch tasks at the target core to prefetch the cache lines that belong to the address ranges specified by the programmer. The software approach shows that it can prevent the dilation in execution time but may result in high and potentially unbounded migration delay requirements. Hence, we propose two schemes of push-assisted cache-to-cache migration in multicores. (1) WCM, a hardware scheme replicating the cache context of the task onto the target L2 cache, reduces dilation in execution time to less than a percent for the majority of simulated tasks, except for those with the smallest data set sizes or an algorithmic complexity

lower than $O(n)$. WCM eliminates any execution overheads of PTM but still maintains a high overhead due to a complete scan of the cache. (2) RCM uses the address range specification capabilities of PTM with hardware-based pushing of cache lines of WCM. This allows RCM to achieve performance benefits similar to WCM while making the cache migration overhead proportional to the cache footprint of the task. This demonstrates that cache migration is a feasible solution for preserving execution times after task migration close to those tight WCET bounds otherwise only valid in the absence of migration.

We further enhance the MESI coherence protocol to significantly reduce or even eliminate the number of write misses due to task migration. This eliminates extra bandwidth requirements due to cache migration except for residuals of tasks with large data sets.

In our recent publication we use this migration mechanism to predict the Cache Related Migration Delay (CRMD) and improve task set schedulability by using it in combination with a semi-partitioned scheme. Locked cache migration has been used to improve the predictability of cache migration. We delve in detail on providing support to migrate locked caches for tasks in the next chapter.

Chapter 3

Deterministic Task Migration for Hard Real-Time Tasks in Multicore Processors

3.1 Introduction

Locking cache contents in uniprocessor hard real-time systems has been a popular option for hard real-time systems designers. Locked cache contents are immune to cache replacement, which improves the predictability of cache access behavior of a hard real-time task. Additionally, large caches with high associativities reduce intra-task conflicts [43], which can enable shorter worst-case execution time (WCET) for hard real-time tasks by using locks. Conventionally, the programmer incurs a cost of loading the locked cache lines prior to the execution of the task [66]. Subsequently, execution of the task assumes cache hits for locked lines. However, the impact of locked cache lines has not been studied on multicore platforms, where their immobility may render global multicore scheduling policies unusable. Also, under such policies, the migration delay is assumed to be constant and added to the WCET of the task. Such assumptions may lead to highly conservative migration delays. This problem worsens with simultaneous task migrations that are prevalent in such scheduling mechanisms.

Pinning hard real-time tasks onto cores and using partitioning across multicores is a sub-optimal solution as we discussed in Chapter 1. Another approach to the problem is static analysis of the migration delay by modeling cold caches after migration. However, static analysis may result in loose WCET bounds that potentially render a hard real-time system unschedulable that otherwise would have been schedulable had cold misses been avoided due to cache locking.

Thus we identify locked cache line mobility as a hindrance to task migration in real-time systems. We propose task migration support that can render task migrations efficient and

predictable for multicore scheduling techniques. Any solution to preserve predictability for locked cache lines has to be proactive in nature to guarantee that these lines are also locked at the target core after migration. Previously, we proposed a cache migration scheme that push cache lines of a migrating task from a source core to the target core during idle slots in the schedule. This work technically contributes in the following ways:

1. This work is the first one to consider mobility requirements of locked cache lines for task migration. Otherwise impractical theoretically optimal multicore scheduling techniques thus become realistic in the context of hard real-time systems. Prior work is not directed towards locked cache lines.

2. In Chapter 2, we provided a hardware/software mechanism called Regional Cache Migration (RCM) to identify and move large contiguous memory regions specified by a limited number of Region Registers. We extend the identification of migratable cache lines to locked cache lines. We then expose the potential of reducing individual cache migration delays by pipe-lining the cache-to-cache transfers. We propose two such schemes that work with RCM. These schemes, called Controlled Cache Migration Pipelining (CCMP) and Streamed Cache Migration Pipelining (SCMP), reduce migration delays by 48% and 56%, respectively in our experiments.

3. In Chapter 2, we proposed a hardware mechanism called Whole Cache Migration (WCM) for cache footprints that are sparse w.r.t. memory address space. However, it imposes an overhead that becomes proportional to the cache size instead of the number of migratable cache lines. In this work, we present another hardware based mechanism called Set-Scan Cache Migration that presents an efficient and practical solution to sparse locked cache footprints.

4. in Chapter 2, we considered single task migration. Such an assumption leads to multiple task migrations to take place sequentially, thereby under-utilizing the bus bandwidth. In this work, we propose a novel mechanism that can be used while generating a work conserving static schedule that may incur multiple cache migrations. We introduce a mechanism that can support multiple cache migrations in parallel without any conflicts and provide an alternative to pipelined cache migration schemes.

5. While SSCM that works efficiently for single task migration, we show that it cannot be synchronized for parallel migration with RCMs. We fill this gap by proposing Slotted-SSCM that allows multiple cache migrations based on RCM and Slotted-SSCM to progress in parallel without any bus conflicts.

6. We also propose a Slotted-SSCM Pipelining model that reduces the migration delay for

single task migrations by 46.7% over SSCM on average for a subset of Malerdalen WCET benchmarks [7].

7. In Chapter 2, migration costs are assessed by assuming migration of the total cache footprint of a task. Thus, the overheads stated in Chapter 2 for RCM are too loose. Here, we allow pre-determined locked cache lines to migrate. We also establish deterministic migration delay bounds for each scheme irrespective of whether cache locks are used to lock contiguous (RCM like) or sparse (SSCM like) memory locations.

3.2 Problem Analysis

In this section, we exhibit the importance of task migration in the presence of cache locking in multicore platforms as a problem for real-time systems.

Cache Locking in Hard Real-time Systems: Hard real-time tasks have stringent deadlines that have to be met or the system may fail. Recent research works have considered systems that co-schedule non-real-time along with real-time tasks while sharing common resources [81, 62]. In such systems, one such time-critical shared resource is the on-chip cache. Soft real-time or non-real-time tasks may have large memory footprints that lead to intra-task cache contention. In contrast, hard real-time tasks are constructed to have smaller memory footprints. Especially with the current trend of large L2 and L3 caches, we believe that hard real-time tasks can fit within the cache with low or no intra-task contention. Nonetheless, inter-task cache contention with other non-real-time tasks hampers the cache behavior predictability of hard real-time tasks. Thus cache locks are useful to improve the predictability of cache behavior. As a side effect of memory regions at lower levels of the memory hierarchy, execution time of hard real-time tasks may be reduced significantly since intra-task cache contention may be low if not eliminated.

Cache locks have long been studied for uniprocessor systems. Locks can be applied statically or dynamically. In static cache locking, the system locks the entries pertaining to a task into the cache at start-up phase. These locked entries are resident within the cache during the lifetime of the task. On the other hand, dynamic locking requires reload points to be identified. At these reload points, cache lines pertaining to a certain region are locked. Study of dynamic and static cache locking analysis is not the focus of our work. Though in this work we are considering static cache locking.

Impact of Cache locks on multi-tasking systems: We conducted experiments on a uniprocessor model with and without cache locks for hard real-time tasks to substantiate the impact of cache locking for hard real-time systems. Table 3.1 shows a subset of Malardalen WCET Benchmarks [7] used in our experiments as hard real-time benchmarks. Each of these

Table 3.1: Experimental Benchmarks

Benchmark	Functionality / Hard Real-Time Routines
fft1	1024-point Fast Fourier Transform (Cooly-Turkey algorithm)
jfdctint	Discrete-cosine transformation (8x8 pixel block)
bs	Binary search (array of records)
crc	Cyclic redundancy check (40 bytes of data)

benchmarks was run individually along with a non-real-time Cnt benchmark.

Assumptions of the study: We model a processor with an in-order core and multi-level inclusive caches in SESC. WCET benchmarks have small memory footprints. Therefore, to model inter-task contention, we used configurations with small caches as usually is the case in real-time systems literature [55, 65, 69]. Our experimental cache hierarchy has 2KB 4-way associative L1 data and instruction caches, and a 8KB 8-way associative unified L2 cache with a cache line size of 32 bytes at both cache levels. We have used inclusive caches as they are common in commercial processors because they prevent write-backs of non-dirty lines during replacement from L1 to L2 that are otherwise required in exclusive caches. Also, multi-processors tend to constrain the coherence protocol to a single level, .i.e., at L2 to simplify design. The simulated caches were modified to support locks at both L1 and L2 levels. For inclusive caches, a line locked in L1 is also locked in L2. One may argue that such a design would lead to a waste of L2 cache space. However, we consider that such a provision allows an application developer to choose the level of cache that benefits the application. In order to perform that we implemented a *lock* instruction to the Instruction Set Architecture used by SESC processor model. The lock instruction uses two arguments: memory address and the level at which a line is to be locked. Static locking has been used to lock cache lines. Unfortunately, there is no static analysis tool available with unified multi-level cache locking to select cache lines for locking to that would derive optimal WCET bounds. Thus we lock instruction and global data pertaining to all the paths within the hard real-time tasks that fit within the large L2 caches. An optimal selection of cache lines to lock reduces inter and intra task conflicts induced by locking. The work presented in this chapter focuses on providing support for migrating locked cache lines on multicore architectures. Shekhar et al. use our multicore locked cache line migration support to semi-partition the tasks while focusing on resolving inter- and intra-task conflicts [80]. The L1, L2 and Memory access latencies have been set to 1, 10 and 100 cycles, respectively. As for the benchmarks, bs and crc benchmarks are being executed as hard real-time tasks. The inner loops of fft1 and jfdctint have been re-factored as stand-alone hard real-time tasks.

Table 3.2 shows the impact of cache locking when each of the benchmarks are run in con-

Table 3.2: Impact of Cache Locking (when contented with cnt)

Benchmark	WCET (No lock) [cycles]	WCET (lock) [cycles]	Number of Lines locked (Level)	Reduction in WCET
fft	13922	8302	47 (2)	59.6 %
jfdctint	6125	2143	36 (2)	34.9 %
bs	1842	590	10 (1)	32.03 %
crc	12936	9423	41 (1)	72.8 %

tention with the cnt benchmark that uses streaming input data. The first column contains the benchmark names, second and third columns show the WCET of the benchmarks when run without locks and with static locks. The fourth column shows the number of cache lines statically locked in L2 cache and the level at which they were originally locked. The fifth column quantifies the reduction in WCET percentage obtained by using static locks. The WCETs in this work are experimentally observed maximum execution times obtained by feeding different inputs due to a lack of static WCET analysis tools. All benchmarks show a considerable drop in WCET with locked lines versus without them. It is evident that the streaming nature of the input data in cnt leads to inter-task conflicts, which leads to eviction of the lines in these benchmarks. This is prevented when the lines are being locked. However, it should also be noted that locks show high improvements in observed WCET because they do not have much intra-task cache contention as it is assumed that the L2 caches have enough associativity to accommodate for the footprint of hard real-time tasks.

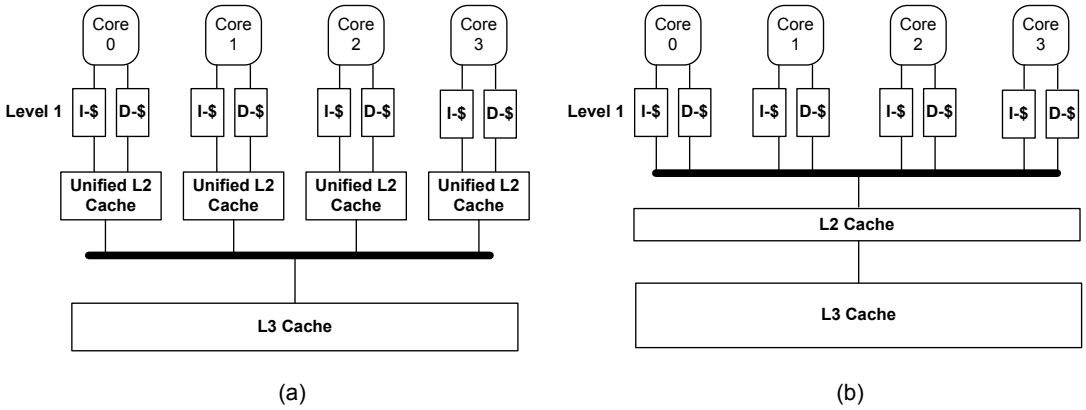
We also take this opportunity to show that multi-level cache locking can be beneficial when L1 cache space is scarce. fft and jfdctint have large instruction memory footprints. Therefore, they have been locked at the L2 level. bs and crc have small instruction footprints. Their instructions have been locked in L1. The total number of lines locked for crc is comparable to fft and jfdctint because crc uses a buffer that is equivalent to the size of the instruction footprint that gets locked in L1 data cache.

These results pose the primary advantages of cache locking namely; immunity from inter-task contention and improvement in execution time for short hard real-time tasks. Those are the key reasons for cache locking to be prevalent among embedded processors, like the IBM PowerPC 460S, Motorola MPC7400, Intel 960, ARM 940T etc. Studies on intra-task cache conflicts, and work on identification of locked cache lines is orthogonal to ours and studied elsewhere [85, 86, 70, 66, 65, 92]. These results also emphasize that without locks, dilation in execution time of hard real-time tasks can have adverse effects as documented in Chapter 2.

Multicore Cache Architecture Assumption: The cache organization is similar to that

of Chapter 2. Figure 4.2 exhibits the two SMP designs that use the private caches. Both architectures have three levels of caches except that in Figure 3.1(a) the L3 cache is shared among processors while in Figure 3.1(b) sharing begins at the L2 cache. Tile processors from Intel and Tilera with on-chip message passing capabilities have recently been advocated. Therefore, we choose the design shown in Figure 3.1(a) based on current industrial trends.

Figure 3.1: Symmetric Multi-processors



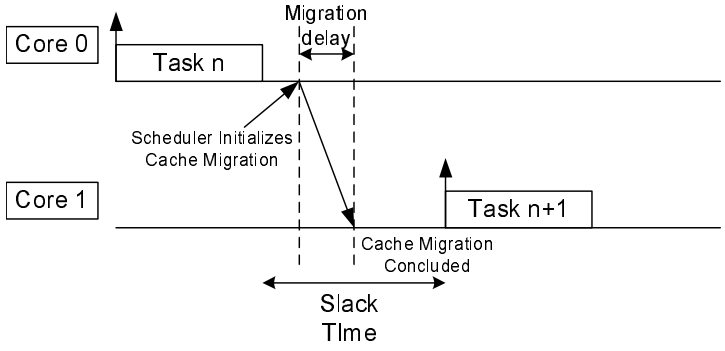
3.3 Proposed Solution and Assumptions

In this section, we present the problem scenario and justify our proposed solution at a high level.

Chapter 2 presented push-assisted cache migration with micro-architectural enhancements that migrate the cache lines pertaining to a task proactively. Such active movement of cache content comes at the cost of up-front migration delay in contrast to the traditional delayed overhead at the next job activation [77]. We assumed tasks are statically scheduled on every core, with each core running its own scheduler. We presented a single task migration scenario that triggers a task migration following a dynamic task admittance. A task that has ample slack time before its next invocation is chosen to migrate. The slack time that the task has is overlapped with proactive migration of cache lines from source core to target core. Thus the overhead does not contribute to the execution of the task when it resumes on the target core, as shown in Figure 3.2. The proposed solution is not applicable when multiple tasks may be migrated. Also, if the decision of task migration happens just prior to the migration, there is no slack that can be used. In this case, one needs to add the migration cost to the schedule.

One such scenario is when using schedulers like PFair [59]. This schedules tasks on quantum basis. The scheduler is invoked periodically. The scheduling is globally synchronous across all cores, i.e., execution of tasks are stalled and the scheduler selects the tasks to be scheduled on the given computing resources for next time quanta. E.g., Figure 3.3 shows a set of tasks executed (A,B,C,D,E,F,G) on a quad-core. At every scheduling point, the scheduler is invoked. In the first two time quanta the task selection does not cause any task migrations. In the third time quanta, scheduler decides to schedule B,D,E,G on the 4 cores. At least 2 tasks are to be migrated as tasks B and D were previously executing at the same computing resources as E and G, respectively. The scheduler decides to schedule B and D at a different computing resource. This incurs task migrations of B and D and the migration cost gets added to the schedule as shown in the figure. However, multiple task migrations can be highly unpredictable as will be illustrated later. Also, the addition of the migration cost to the schedule motivates us to reduce the migration cost. As the PFair algorithm has been proven to be theoretically optimal for multicore task scheduling, we assume that these algorithms can be used to generate off-line static schedules while retaining predictable migration costs with our migration support.

Figure 3.2: Scheduler Initiated Cache Migration Overlaps Slack Time



The simplest way of performing proactive cache migration is to unlock the lines at the source core. Thereafter, the target core can load and lock these lines. Both unlock and lock sequences can be encapsulated into threads spawned by the scheduler at the source and the target cores, in that order. Please note that this will contribute to the migration delay as shown in Figure 3.2. Such a scheme has following drawbacks:

1. In real-time systems, threads execute on a simple in-order core. This means that only one demand request can be issued at a time. This serializes all the demands.

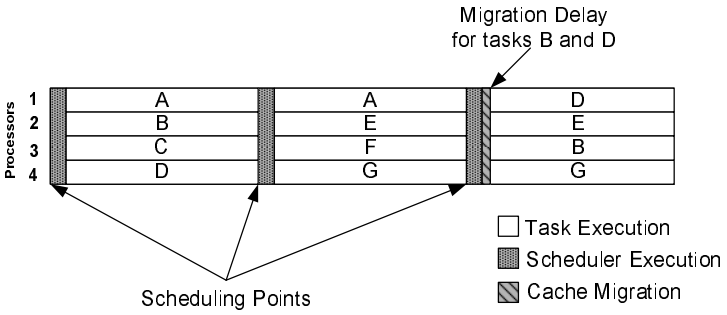
2. Each demand request in a contemporary bus-based SMP will be snooped by all caches.

This will induce multiple response actions on all cache controllers of the system. This not only affects the bus bandwidth but causes useless cache accesses. This, in turn, makes parallel transactions highly unpredictable and inefficient.

3. As demand requests originate from an executable, these executables have their own execution overheads to calculate simple addresses.

4. Unlocking at the source before locking lines at the target, does not prevent off-chip accesses when re-loads are issued.

Figure 3.3: Cache Migration follows Pfair scheduling



These inadequacies led us to exploit push-based micro-architectural cache migration presented in Chapter 2. Firstly, we enhance the current state of the art of push-assisted cache migration to improve the performance of individual cache migrations. Then, we propose a novel scheme that allows the scheduler to orchestrate multiple cache migrations to synchronize and proceed in parallel seamlessly. We also show that we have a pre-calculated bound within which they complete.

We assume the scheduler can identify if the target core has enough cache lines for all lock requests of the migrated task. Such issues are orthogonal to our work because scheduler interaction is required irrespective of the type migration (thread-based pull or hardware-based push). In a recent publication, we proposed a mechanism that resolves locked region conflicts before using our migration model to deduce the migration cost of statically determined task migrations in order to semi-partition the task on a multicore architecture [80]. Here, we intend to show that if a hard real-time task has been chosen for migration, then we can deliver an efficient and predictable cache migration delay.

3.4 Migration Models

In order to explain the migration model, we use "source" and "target" to refer to the core/cache where the task is currently running and where the scheduler migrates the task to before it resumes execution, respectively.

3.4.1 Push Model

A Push Model is one where memory requests are initiated by the source core in order to warm up the target cache instead of demand-driven requests issued by the target. We introduced two such migration schemes: Whole Cache Migration (WCM) and Regional Cache Migration (RCM). WCM is a hardware mechanism, where every cache controller has a push logic block and each cache line has a PID associated with it. When the task migrates, the target is initialized to start pushing the cache lines with a push request. The push request is a point-to-point request. This means that a push request will be issued by a source core and referenced by the target core only. This prevents any useless acknowledgments or cache accesses by cores besides source and target. Therefore a push request carries the information about the target core, along with the data and the tag of the cache line that is being migrated. However, WCM scans through the whole cache in order to identify each cache line that needs to be migrated. Thus the migration overhead is directly proportional to size of the cache. RCM is a hardware/software approach that uses support of dedicated registers called Regional Registers (RR). RRs hold regions of consecutive memory locations as pairs of start and end addresses. Each Task Control Block (TCB) will hold region information for RRs as identified by the programmer. The scheduler fills these limited number of registers before migration. Thereafter, a push block unit computes addresses sequentially that fall within these regions, searches for them in the cache and pushes them to the target. Since this scheme uses addresses to search the cache, cache lines do not need a PID. Refer Chapter 2 for further details on the hardware complexity of the push block, and integration of push requests with coherence.

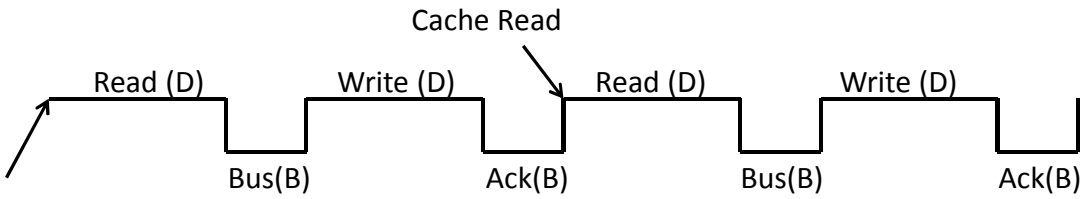
3.4.2 Regional Cache Migration (RCM)

Figure 3.4 depicts the diagrammatic representation of two cache line transfers performed consecutively using RCM. Each cache line transfer has a cache read at the source, followed by a bus transaction to the target succeeded by a write at target, which finally concludes with an acknowledgment request that is also monitored by the memory controller (in case it carries a write-back message from the target). We use RCM scheme as our base scheme, yet with a minor modification: we add a lock-bit check with each cache read as we migrate only locked cache lines. In RCM, a cache read is initiated by the push logic only after the acknowledgment

has been received. This serializes each transaction. Next, we deduce the migration delay T_m for RCM based cache migration. Let us assume that the worst-case cache access time is D cycles, uncontended cache-to-cache migration takes B cycles and the number of locked cache lines is C_n. Each cache line takes two cache accesses (incurring $(2 \times D)$ cycles delay), and one push request put onto the bus by the source tile/core (incurring B cycles delay) and another transaction placed on the bus by target tile/core to acknowledge the push request, which can be piggybacked with a write-back of the evicted cache line (incurring another B cycles delay). Thus, each cache-to-cache transaction takes $(2 \times (B + D))$ cycles. The serialized nature of RCM gives us the migration delay T_m incurred by a single task migration as

$T_m = C_n \times (2 \times (B + D))$,

Figure 3.4: Regional Cache Migration Operational Sequence

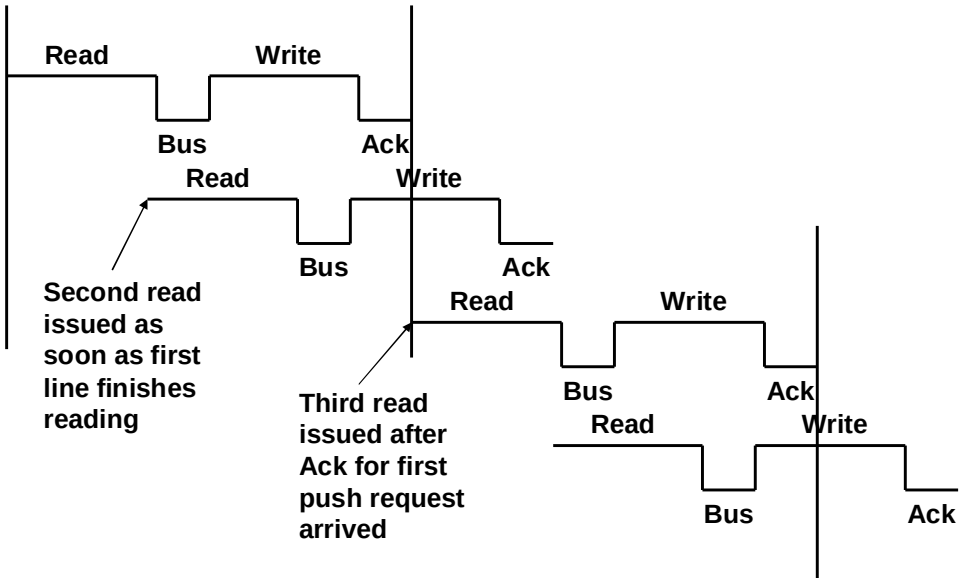


RCM in its current form is inefficient. Each cache read waits for the acknowledgment to arrive. When a cache is being read or written to, the bus is idle. When the bus is being accessed, the cache controllers are idle. To mitigate this under-utilization of cache and bus resources, we present our first novel scheme called Controlled Cache Migration Pipe-lining.

Controlled Cache Migration Pipe-lining (CCMP)

Subsequent cache reads can be serviced while a request has been placed on the bus as caches are supported to issue multiple requests. Thus a pending request buffer is created that holds two pending requests at a time. This means that, at any time, there can be only two pending push requests and no cache read can be performed until one of the acknowledgments reaches the source core. This is shown in Figure 3.5. One unique observation of such pipelining is that bus transactions of push requests and acknowledgments do not interfere with each other as seen in Figure 3.5. Individually, each push request requires $(2 \times (B + D))$ to complete as explained previously. In Figure 3.5, for the purpose of evaluating migration delay, we assume that the first and second push requests form the first pair, and the third and fourth push requests form

Figure 3.5: Controlled Cache Migration Pipe-lining Operational Sequence



the second pair. In the figure, we see that the first push request in the second pair begins when the first request in the first pair finishes. Thus, one can say that each new pair of push requests is at an offset of $(2 \times (B + D))$ cycles. So, assuming that C_n is an even number of cache lines, the last pair of push requests begin at $((\frac{C_n - 2}{2} \times 2 \times (B + D))$. The last pair would finish migration when the second push request of the last pair finishes. Since the second push request lags behind the first one by D cycles, the last pair takes $(2 \times (B + D) + D)$. So for an even number of C_n, the migration delay is $(\frac{C_n}{2} \times (2 \times (B + D)) + D)$ cycles. If C_n is an odd number, then the last push request is not paired but rather constitutes a single push request. The cache migration is considered to be finished as soon as the last transaction finishes. This last request takes $(2 \times (B + D))$ to finish. Since C_n is an odd number, the number of pairs considering the last transaction pair is $\lfloor \frac{C_n}{2} \rfloor$. So, for an odd C_n, the migration delay is $(\lfloor \frac{C_n}{2} \rfloor \times (2 \times (B + D)))$. Hence, T_m for all C_n is

$T_m = \lfloor C_n/2 \rfloor \times 2 \times (B + D) + V$

where V=0 if C_n is odd and V=D if C_n is even.

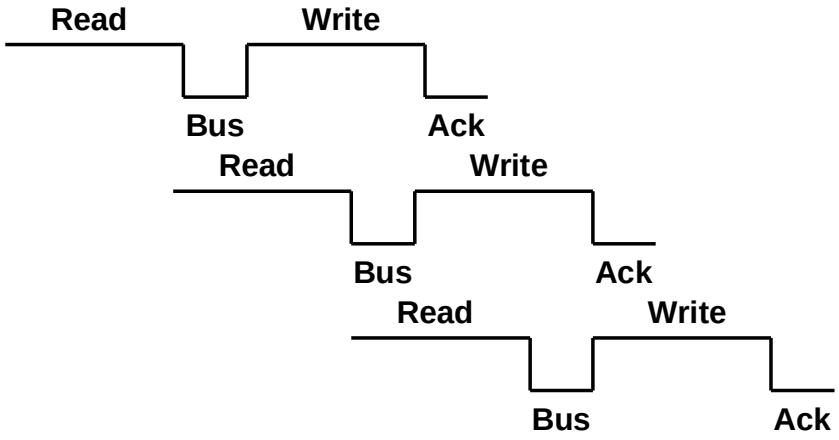
Figure 3.5 further illustrates that if the third cache read is allowed just after the completion of the second cache read, then the push transaction issued by the third cache read can only conflict if the bus delay is greater than half of the cache read access. Thus the CCMP model is valid for processors where the bus delay is less than or equal to the cache access. With the advent of HyperTransport [1] and Intel Quickpath interconnect [3], we believe the cache-to-cache transport delays will remain smaller than the cache accesses. The observation mentioned

above lead us to develop our next scheme called Streamed Cache Migration Pipe-lining.

Streamed Cache Migration Pipe-lining (SCMP)

This migration model allows every subsequent transaction to resume with cache reads without waiting for any acknowledgments to arrive. This scheme is extremely efficient when the bus delay is shorter than half of the cache read access time. Figure 3.6 depicts the migration of three cache lines from a source to a destination under SCMP. As shown in Figure 3.6, none of the push requests conflict effectively resulting in streaming behavior of cache migration.

Figure 3.6: Streamed Cache Migration Pipe-lining Operational Sequence



Since each subsequent request lags behind the previous request by D cycles, the last transaction starts $((C_n - 1) \times D)$ cycles after the first push request is issued. The last transaction takes $(2 \times B + D)$. Thus, the migration delay incurred by SCMP is

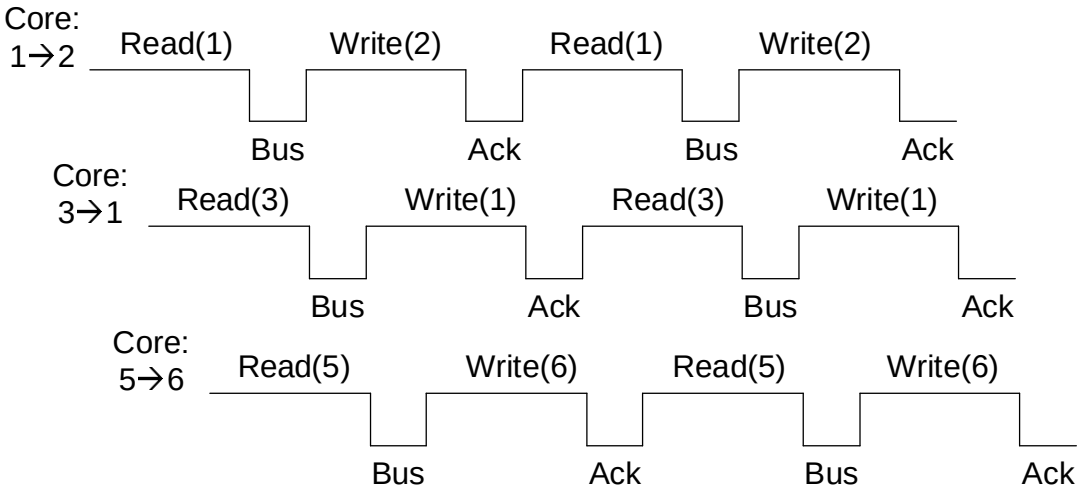
$T_m = C_n \times D + (2 \times B + D)$.

The above schemes improve the performance of individual cache migrations. But in situations where multiple cache migrations are required, there is room to engage in seamlessly parallel cache migrations.

3.4.3 Parallel Cache Migrations

When multiple hard real-time tasks are migrated simultaneously, support for multiple cache migrations is required. Suppose all cache migrations were using RCM. By maintaining a multiple of B as a time difference between cache reads of any two RCM chains, we can support multiple cache migrations without contention (see Figure 3.7). We denote a cache migration by

Figure 3.7: Parallel Multiple RCMs



core pairs, where a core pair contains a source core ID and a destination core ID. For example, the first RCM in Figure 3.7 is denoted by the core pair (1,2), where 1 and 2 are the core IDs of source and destination cores, respectively. Also, multiple cache migrations can support a core to be a source as well as a target at the same time: In RCM, source cache is idle waiting for an acknowledgment once a cache line has been pushed. This idle time can be used for a write in case it is the target of another cache migration by placing the two transactions next to each other as shown in Figure 3.7. In fact, the only scenario when synchronization does not hold true is for inverted source,target pairs. For example, in PFair the tasks to be scheduled in the next time quanta are first identified. In order to find the tasks to be migrated, the allocated tasks may be visited in some order for allocating them to cores. If a task was executing on a core that has not been allotted to any other task yet, then it gets reserved for that task. In case it has already been taken by another task, then the task is migrated. Such a scheme will not lead migrations of core pairs (1,2) and (2,1). To obtain pairs like (1,2) and (2,1), one of the allotments has to precede the other. If (1,2) occurred first then core 1 has been allotted to another task. If such allotment was due to a task migrating from 2 to 1, then 2 has already been allotted. This contradicts our assumption that (1,2) occurred before (2,1). Same logic is applicable to non-existence of circular transactional paths, like (1,2),(2,3),(3,1). Parallel transactions have the ability to maximize the utilization of the bus bandwidth. However, it depends upon how many bus transactions can be performed between a push request and its acknowledgment, which is the limit to the number of transactions that can be supported in parallel. Numerically it is equal to $\lfloor D/B \rfloor$.

This scheme additionally requires synchronization between parallel cache migrations across

cores such that a cache migrations shown in Figure 3.7 do not conflict with each other. The key to this scheme are the initialization and synchronization of these transactions by the scheduler. The scheduler and cache controllers interact as follows:

1. The scheduler determines the tasks subject to migration and creates core pairs. Note that core pairs will have distinct targets but can have identical sources.

2. First, core pairs with the same source core need to be placed into separate parallel transactions, which we call buckets. Thus, each bucket will hold core pairs with distinct source cores. This allows each migration within a bucket to run in parallel. We use *count sort* to split the core pairs between buckets. Count sort makes two passes over the core pairs. During the first pass, it counts the number of times each source core id appears and stores the count value per source core. During second pass, the count value corresponding to a core pair's source core is used as the bucket ID and then decremented. This step would put cache migrations (1,2), (3,1) and (5,6) in a single bucket as these migrations can be performed in parallel as shown in Figure 3.7. However, this will put (3,7) and (5,8) in a separate bucket as they share sources 3 and 5 with cache migrations depicted in the earlier bucket. Thus, cache migrations within a bucket can run in parallel but each bucket is sequential with respect to another.

3. As can be seen in Figure 3.7, parallelism is obtained by utilizing each bus cycle for transactions. However, if there are two core pairs (1,2) and (3,1) in a bucket, where core 1 is a destination in the former and a source in the latter, it is important that these two cache migrations consume the bus cycles right after one another. For example, if the cache migrations (5,6) and (3,1) switch places in Figure 3.7, then there will be a conflict between the second cache line read for (1,2) and the first cache line write for (3,1). To avoid that, we use partial ordering.

   Algorithm 23 shows the partial ordering algorithm used. The algorithm requires an an input the number of cores, a list of cache migration core pairs $CPList$ and an Array of tuples CMD. Each entry of CMD has two members: The num value and a pointer to a core pair ptr. A core pair contains two core IDs in a member called $pair$. A core pair can point to another one using a $next$ pointer. Line 1 initializes each entry of CMD with 0 (num) and Null (ptr) values. The loop in lines 3-17, iterates over all core pairs in $CPList$. The nested loop iterates over both the core IDs for the $pair$ member of a selected core pair. Line 4 checks if there is any other core pair that has already been allocated to the core denoted by id. If there is no entry then the num value is incremented and the ptr to that of the corresponding CMD entry points to c, which is the selected core pair. Otherwise, the loop at line 8 iterates over each of the core IDs of the already seen core pair. One

of the core IDs is going to be the same as *id*. The other is going to be either higher or lower than *id*. In case it is higher, then *c* precedes the already seen entry. Otherwise, the already seen core pair precedes *c*. This is done by modifying the *next* pointer of the preceding core pair, which gives us a partial order. The *num* value corresponding to *id* is incremented and the respective *ptr* entry points to the preceding core pair. This partial ordering is accomplished by lines 9-17. Lines 18 initialize the *partOrder* list with no valid entries. The loop in lines 19-20 iterates over each CMD entry. If an entry's *num* value is found to be non-zero, then that entry points to a core pair that is at the head of a chain of core pairs. This chain may only have a single core pair entry or a list of core pairs. Line 21 appends this chain to *partOrder*. Line 22 decrements the *num* values of CMD entries corresponding to the core IDs that appears in the chain. The *num* values are decremented as many number of times as they appear in the chain, which is at most 2: as a source and as a destination. When a *num* value becomes zero, the corresponding *ptr* is changed to NULL. Finally, line 23 returns *partOrder*, which holds an ordered list of cache migration core pairs with partial ordering between core pairs that use a common core.

Figure 3.8 shows the steps taken by our algorithm to order a given set of core pairs with an example. Following is a description of these steps:

(a) shows an unordered list of core pairs: In order to sort them, we use an array of tuples, CMD, consisting of a numerical value and a pointer to a core pair. Here we assume 8 cores. CMD is indexed by core ID. The numerical value at an array index denotes the number of core pairs involving a particular core. Since each cache migration is bound to have a unique target core and a source core can occur only once within a bucket. The maximum number of core pairs a core might be involved within a bucket is 2: Once as a source and Once as a target. The numerical values are initialized to zero and all the pointers are initialized to NULL.

(b) selects core pair (1,3): This updates the first and the third items in the array. Their numerical values are incremented from 0 to 1. Both the array items point to the core pair (1,3).

(c) selects core pair (4,2): This updates the second and the fourth items in the array. Their numerical values are incremented from 0 to 1. Now both of them point to the core pair (4,2).

(d) selects core pair (6,5): Updates are made to the sixth and the fifth items similar to the prior two steps.

(e) selects the last the core pair (5,4): As we can see from the previous steps, the fifth and the fourth items already have an entry, each. However, both the entries point

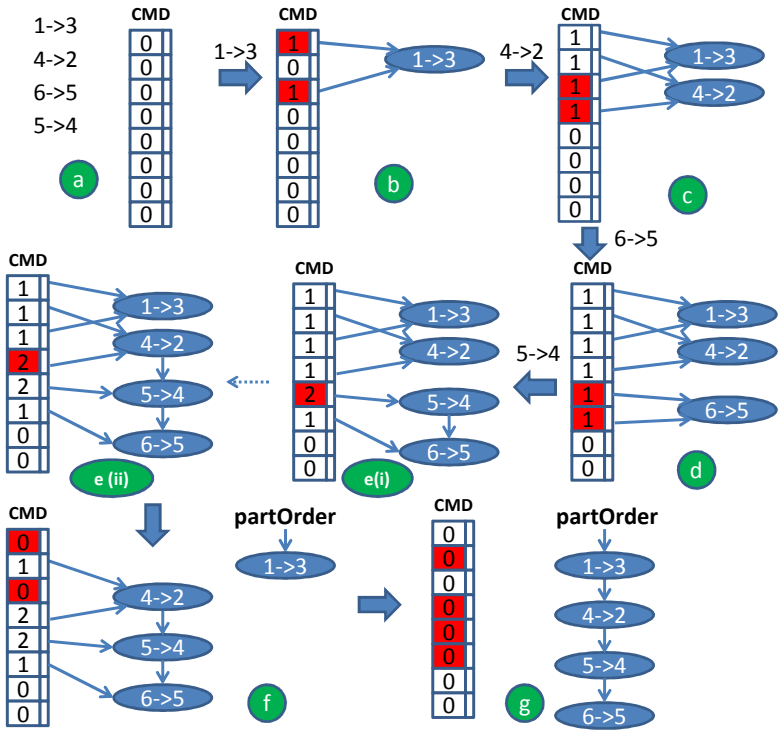
ALGORITHM 1: Partial Ordering of Parallel Cache Migrations

Input: $CPList$: List of cache migration core pairs, $numOfCores$: total number of cores,$CMD[numOfCores]$: Cache Migration Decriptor array

Output: $partOrder$: ordered list of cache migrations

```
1  initialize CMD ;
2  foreach CPList c do
3      foreach c.pair id do
4          if CMD[id].num ≠ 0 then
5              CMD[id].num := 1 ;
6              CMD[id].ptr := c ;
7              continue;
           end
8          foreach CMD[id].ptr.pair cid do
9              if id < cid then
10                 incrementCMD[id].num;
11                 c.next := CMD[id].ptr;
12                 CMD[id].ptr := c;
13                 break;
               end
14             if id > cid then
15                 incrementCMD[id].num;
16                 CMD[id].ptr.next := c;
17                 break;
               end
           end
       end
   end
18 initialize partOrder;
19 foreach numOfCores coreId do
20     if CMD[coreId] ≠ 0 then
21         Append partOrder with CMD[coreId].ptr;
22         Decrement num values for each ID that appears in list pointed by
           CMD[coreId].ptr;
       end
   end
23 return partOrder;
```

Figure 3.8: Partial Ordering of Core Pairs



to a different core pair. So, we first try to update the entries for the 5 (source) and then 4 (target).

i. Here, the numerical value of the fifth item is incremented from 1 to 2. The corresponding pointer which points to core pair (6,5) is dereferenced. To construct an order, we allow core pairs to point to the core pair next in order. In this example, when we encounter two core pairs using a core ID, we compare the their paired cores as they are bound to be dissimilar. The core pair whose paired core ID is smaller precedes the other core pair. Therefore, at the end of this sub-step, the fifth item is pointing to core pair (5,4) that precedes (6,5) since the paired core's ID 4 is smaller than ID 6.

ii. Here, the numerical value of the fourth item is incremented from 1 to 2. The pointer pointing to core pair (4,2) is dereferenced. The paired core ID 2 is smaller than ID 5, that of the current core pair. Therefore, (4,2) precedes (5,4) and the fourth item points to (4,2). At the end of this sub-step, all core pairs

have updated the array of tuples. All partial orders have been formed.

(f) begins forming the absolute order by traversing the array of tuples in ascending order of their core IDs. The item at index 1 points to core pair (1,3). The numerical values for the first and the third items are decremented as this core pair is removed and placed at the head of the ordered list.

(g) reads the second item in the array. This points to the core pair (4,2), which is at the head of *partOrder*, an ordered list of core pairs. All core pairs in the list are removed in that order and updated at the end of *partOrder*. While removing each of the core pairs, the corresponding items in the arrays are decremented. At the end of this step, all core pairs are removed. The process will terminate by reading 0 values of the rest of the entries suggesting that no other core pairs need to be ordered.

This ordering allows one to establish a synchronization among RCMs, like the one shown in Figure 3.7. Once ordered, we know that migration from 3 to 1 has to be started at a time instant such that the behavior shown in the figure can be replicated. This can be achieved by giving an offset value to each RCM. An offset indicates the number of cycles that a cache controller should wait to start an RCM relative to the start of the first RCM. For a static schedule, these offsets for each scheduling point can be computed offline and loaded into the Thread Control Blocks (TCB) of migrated tasks.

4. The core with an offset of zero packs its region registers within a data block. As we are using only 4 pairs of region registers, they fit within the size of 32 bytes (size of a cache block). This methodology is used because sources for two transactions in different buckets can be same. If we allow the scheduler to update the region registers, then the scheduler would need to activate after every bucket finishes. This can be avoided by an initial transfer from target to source. This requires additional 4 pairs of region registers within the push block that has to be sent to the source. But it does not change the parallelization of transactions because the transactions are already two-way communication. This adds a small overhead of $2 \times B + D$ cycles.

5. Each Snoop controller detects the very first message on the bus and records the current cycle time. This value is then added to the offsets. This allows all targets to determine when to issue the request for initialization of push requests.

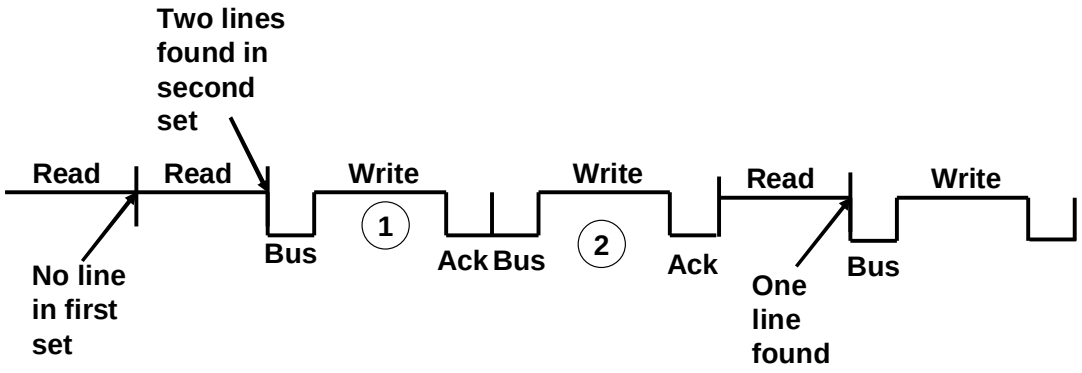
Note that all the offsets can be predetermined because the transaction time for each cache migration is bounded. Even with parallel transactions, the time at which a bucket of transaction finishes can be predicted accurately to compute the offsets for next bucket of transactions. Within a bucket, the offset values are such that the transactions are B cycles apart. If there is a single transaction in a bucket, it can use pipe-lined models like SCMP or CCMP. This

synchronization only requires an additional set of RRs, offset value registers and minor logic to extend access to these registers.

Until now, we have considered RCM-based approaches. RCM is useful when long sequential paths of code, global data arrays, or closely located groups of global variables are locked. However, the locking of sparse memory locations needs cache migration support as well. This motivates a hardware solution called the Set-Scan Cache Migration model.

3.4.4 Set-Scan Cache Migration (SSCM)

Figure 3.9: Set-Scan Cache Migration Operational Sequence

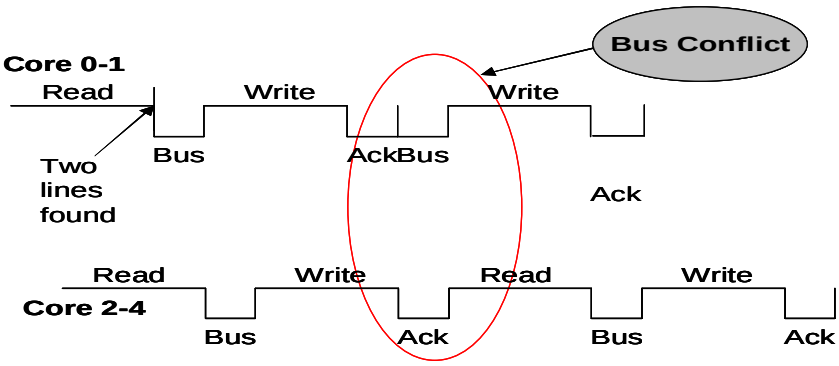


The push block in SSCM identifies the locked lines pertaining to the migrated task through hardware enhancements. Since the push block is unaware of the locked regions, it requires a Process Identifier (PID) information associated with the cache line to determine that the locked cache line belongs to the migrated task. Thus each cache line holds a PID tag to associate itself to a process. In a conventional cache access, the content of the whole set is read. If the searched tag matches to any entry then it is forwarded. However, SSCM can have multiple matches because multiple locked entries may belong to the same process within a set. Hence, SSCM reads the whole cache set, identifies the first matching line and forwards it to be placed on the SMP bus for migration. Instead of discarding the recurring entries of the set, they can be buffered. Thus multiple matching entries can be identified in very short time while the first matched entry is in transit. On receiving acknowledgment, the next matching entry can be transmitted immediately without any read delays. This is shown in Figure 3.9, where the push request marked 2 is placed on the bus as soon as the ACK for the push request marked 1 has arrived. This prevents the hardware mechanism from reading a set multiple times. When there are no matching entries in a set at all (as is the case with the first read in Figure 3.9) extra

reads may be introduced, each adding an extra read latency to the total migration cost. The mathematical analysis of this scheme introduces another parameter, namely the number of sets within the cache. The number of sets is computed by $\frac{size\ of\ the\ cache(S_c)}{Associativity(A)}$. In SSCM, each cache set is read once, this incurs an overhead of $\frac{S_c}{A} \times D$. In addition to that each cache line migration would result in two bus transactions and one cache access at the target giving a migration delay T_m of

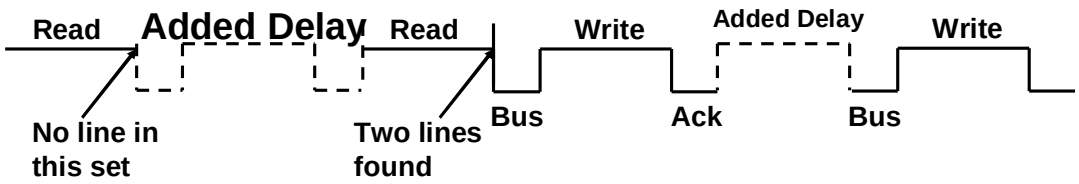
$T_m = \frac{S_c}{A} \times D + C_n \times (2 \times B + D)$.

Figure 3.10: Conflicts between RCM and SSCM Push Requests



So far, we discussed the parallelization of cache migration in terms of RCM. SSCM is a sequential cache migration scheme that could benefit from parallelization as well. In particular, a set of RCM migrations may issue when another set of migrations uses SSCM. A constraint of SSCM is that a push request cannot be overlapped with pushes of RCM. Figure 3.10 illustrates the cause: A set read may find zero or multiple lines to migrate. Thus we propose an aligned version of the SSCM called Slotted-SSCM.

Figure 3.11: Slotted-SSCM Operational Sequence



Slotted-SSCM

This migration model regulates the progress of cache set reads and issuance of push requests. Cache migration is now divided into slots. Each slot has a duration of $2 \times (B + D)$. When a set read yields only one push request, it takes one slot of time. In Slotted-SSCM, if no match is found, it takes one slot and if multiple matches are found then each line migrated accounts for one slot of time by adding delays. This is shown in Figure 3.11. Thus such a migration can now run in parallel with other chains of RCM cache migrations.

However, such a scheme complicates the estimation of migration delay. This is due to the delays added to migrations of cache lines that have been identified by a prior set read operation. This causes every cache line migration to take $(2 \times (B + D))$ cycles. In addition to that, each set that does not result in a cache migration will also cost $(2 \times (B + D))$ cycles. This could result in varying migration delays for a given number of locked cache lines to be migrated. For example, let us assume a 2-way associative cache that has only two cache sets and the user locks two cache lines. If the two cache lines map onto a single set, then it will lead to two cache set reads and two cache migrations. Since one of the cache set reads results in two cache migrations, that cache set read's delay consumes one cache migration delay. Thus, a total migration delay of $3 \times (B + D)$ is incurred. On the other hand, if the two locked lines are mapped onto separate cache sets, then each cache set read would result in a cache migration. This means that each cache set read delay will consume a cache migration delay. This leads to a migration cost of $2 \times (B + D)$. This shows that the migration delay is now dependent upon the mapping of the cache lines. The simple example shows that this delay can be deduced by adding the cache line migration delays and futile cache set read delays. The former is simply $C_n \times 2 \times (B + D)$. The latter is the $number\ of\ futile\ set\ reads \times 2 \times (B + D)$. The $number\ of\ futile\ set\ reads$ is the maximal when locked cache lines are locked to the fewest number of sets that they can map on to, which is $\lceil \frac{C_n}{A} \rceil$. In that case, the $number\ of\ futile\ sets\ read$ is as $\frac{S_c}{A} - \lceil \frac{C_n}{A} \rceil$. Thus, the worst-case cache migration delay experienced by Slotted-SSCM is

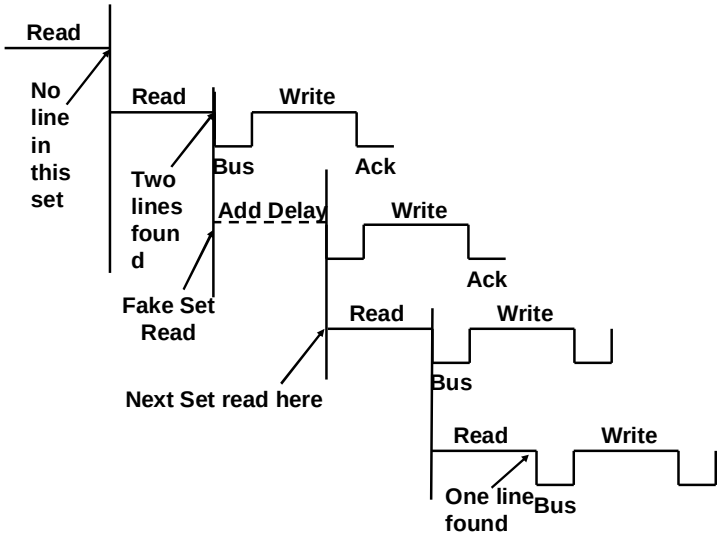
$T_m wc = (\frac{S_c}{A} - \lceil \frac{C_n}{A} \rceil + C_n) \times 2 \times (B + D)$.

Slotted-SSCM migration creates a deterministic behavior of the issuance of push requests. In analogy to the parallel execution of RCM chains under SCMP, we can develop a pipe-lined cache migration model for Slotted-SSCM.

Slotted-SSCM Pipe-lining

This migration model takes us back to pipe-lined migration, but for Slotted-SSCM instead of RCM. This is similar to SCMP in that the push block proceeds with the next set read without waiting for any acknowledgments of the previous transfer. A cache access delay of D cycles is associated with every migrated line. As for SCMP, $2 \times B + D$ cycles of the last push request do

Figure 3.12: Slotted-SSCM Pipe-lining Operational Sequence



not overlap with any other push traffic. However, as for Slotted-SSCM, when a cache set holds multiple locked cache lines for a migrated task, the first migrated line experiences a delay due to an actual set read while subsequent transactions add a fake cache access delay as shown in Figure 3.12. Also, as for Slotted-SSCM, this incurs a delay of D cycles for every futile set read. For example, the first cache set read in the figure results in no cache migration. Therefore, besides every cache line migrated, this also incurs the the delay for the *number of futile set reads*. As derived previously, the worst case *number of futile set reads* is $\frac{S_c}{A} - \lceil \frac{C_n}{A} \rceil$. Hence, the worst-case cache migration delay for Slotted-SSCM Pipe-lining is

$T_m wc = D \times (\frac{S_c}{A} - \lceil \frac{C_n}{A} \rceil + C_n) + 2 \times B + D.$

3.5 Simulation Platform

Our proposed solution requires micro-architectural modifications to a stable bus-based multicore CMP architecture.

The system architecture specifications are presented in Table 3.3. The base simulator environment has been enhanced that allows us to trigger task migration across cores. The purpose of this work is not to provide a scheduling mechanism but rather to evaluate the micro-architectural mechanisms based on migration costs due to single and multiple cache migrations. These mechanisms can be used with different schedulers. The recently proposed Predictable Execution Model (PREM) suggests that by isolating different phases within a task one can provide predictability [64]. We believe that for multicores, a cache migration phase can be

Table 3.3: Simulation Parameters

Component	Parameter
Processor Model	in-order
Cache Line Size	32B
L1 I-Cache Size/Associativity	2KB/4-way
L1 D-Cache Size/Associativity	2KB/4-way
L1 Access latency	1 cycle
Replacement Policy	LRU
L2 Cache Size/Associativity	8KB/8-way
L2 Access Latency	10 cycles
L2 Replacement Policy	LRU
Coherence Protocol	MESI
Network Configuration	Bus based
Processor To Processor Delay	2 cycles
External Memory Latency	100 cycles

added to the already existing memory and execution phases. Thereby, one can produce certain static schedules that allow the bus to be used for a single cache migration or co-locate multiple cache migrations with predictable migration costs obtained by our cache migration models. PFair also decides on all task migrations at the same time. This again allows cache migrations to occur simultaneously [15].

The cache model has been extended to allow us to support locks at different levels of caches. The migration models have been designed and implemented as part of the cache controller model. This allows the detection of locked cache lines, issuance of push requests from "source" core to "target" core and regulation of the rate of requests issued according to the migration models.

3.6 Evaluation

3.6.1 RCM v/s. SSCM

First, we compare the migration delays incurred by programmer-assisted implementations of RCM and complete hardware solutions for cache migration in SSCM as shown in Table 3.4. The first column shows the benchmarks used, the second column shows the number of cache lines that were locked, third column shows the migration delay experienced by RCM and fourth expresses the same in cycles for SSCM. The locked lines are statically locked prior to execution of the benchmark. The cache lines locked are composed of the instructions address space and the global variables and tables being used. As can be seen that for fft, jfdctint and crc, SSCM performed better than RCM. This is because SSCM identifies multiple cache entries in a set

through one cache access while RCM needs to perform as many cache reads as the number of cache lines are locked. This shows that the three benchmarks have their locked lines distributed over the entire cache. In case of bs, the migration delay experienced by SSCM is significantly larger than for RCM. This is because the number of cache lines locked for bs is much lower than the number of cache sets. Each futile cache set read adds to the migration delay. Looking back at the T_m derived for SSCM, $\frac{S_c}{A} \times D$ is much larger than $C_n \times (2 \times B + D)$.

Table 3.4: Migration Delays: RCM vs SSCM

Program	Number of locked cache lines	RCM [cycles]	SSCM [cycles]
fft	47	1128	978
jfdctint	36	864	824
bs	10	240	460
crc	41	912	894

3.6.2 Pipelined RCM techniques

In Section 3.4, we presented two similar pipelining schemes, CCMP and SCMP. Table 3.5 shows the potential of these schemes over serialized RCM scheme. The first column shows the benchmarks used. Second, third and fifth columns show migration delays incurred by RCM, CCMP and SCMP in cycles, respectively. Fourth and sixth columns show reduction in migration delay achieved by CCMP and SCMP over RCM in percent. These results correspond to the equations derived in Section 3.4.

Consider the differences between the schemes. SCMP performs well if the bus delay is less than or equal to half the cache access delay. Once the bus delay exceeds this threshold, CCMP becomes more efficient. This remains true till the bus delay is less than or equal to the cache access delay. However, pipelining becomes infeasible once the bus delay exceeds the cache access delay. This can be mitigated by adding delays to balance cache access and bus delays. This enables CCMP when the bus delay is marginally greater than the cache access delay. For higher bus delays, one may have to introduce multiple hops.

3.6.3 Slotted SSCM techniques

In Section 3.4, we presented the Slotted-SSCM and Slotted-SSCM Pipelining migration models. Table 3.6 shows the experimental and worst-case migration delays for the two schemes.

Table 3.5: Pipelined Cache Migration

Program	RCM [cycles]	CCMP [cycles]	CCMP Savings	SCMP [cycles]	SCMP Savings
fft	1128	576	48.9%	484	57.1%
jfdctint	864	442	48.8%	374	56.7%
bs	240	130	45.8%	114	52.5%
crc	912	446	48.9%	394	56.8%

The experimental results are obtained by triggering a migration of individual benchmarks from one core to another. The worst-case migration delays are computed by the T_mwc obtained in Section 3.4 for Slotted-SSCM and Slotted-SSCM Pipe-lining. The first column shows the benchmarks used. Second, third and fifth columns show the experimental migration delay in cycles for SSCM, Slotted-SSCM and Slotted-SSCM Pipelining, respectively. Fourth and sixth columns show the analytical worst-case migration delay for Slotted-SSCM and Slotted-SSCM Pipelining, respectively. It can be seen that migration delays are significantly reduced by pipelining. However, the worst case migration delay for both Slotted-SSCM and Slotted-SSCM Pipelining are considerably higher than their corresponding experimental results. This is because, in practice, locked cache lines are spread across the sets while the worst case occurs when the locked lines are located within the smallest number of sets that can hold those lines. Thus we recommend that experimental migration delays to be computed off-line if the distribution of locked cache lines is known. This can be inferred from cache design parameters and lock addresses. In order to compute the migration delay, one has to find the sets that the locked cache lines are mapped to.

Table 3.6: SSCM Variants

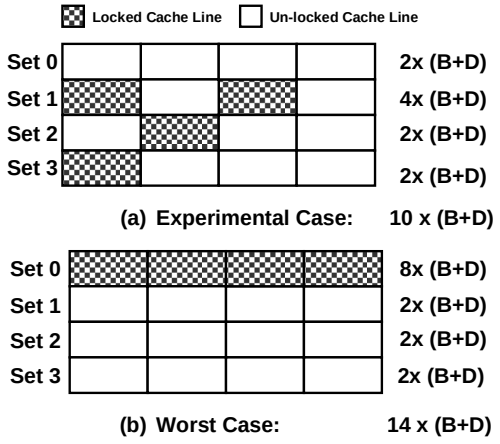
Program	SSCM [cycles]	Slotted SSCM [cycles]	Slotted SSCM (WC) [cycles]	Slotted SSCM Pipelining [cycles]	Slotted SSCM Pipelining (WC) [cycles]
fft	978	1128	1752	484	744
jfdctint	824	864	1512	374	644
bs	460	768	888	320	414
crc	894	1008	1608	434	684

For example, assume a task that has 4 locked cache lines in a 4-way associative cache with

4 cache sets. As explained in Section 3.4, the worst case scenario happens when the cache lines map into $\lceil \frac{C_n}{A} \rceil$ number of cache sets. For our scenario, this is $\lceil \frac{4}{4} \rceil = 1$. This case is represented in Figure 3.13(b). However, in practice, the locked cache lines are mapped as shown in Figure 3.13(a). In the figure, on the right side of each cache set, we show the number of cycles taken by Slotted-SSCM for that cache set. As explained in Section 3.4, each set without a locked cache line causes a delay of $2 \times (B + D)$. Once a line is found, this delay covers the first migrated line. Every additional line migrated from a set incurs a delay of $2 \times (B + D)$. In Figure 3.13(a), set 0 incurs $2 \times (B + D)$ cycles due to a futile cache set read. Set 1 has two locked cache lines. The cache set read covers the first cache line migrated but it incurs an additional migration delay due to the migration of the second locked cache line. Thus, set 1 takes $4 \times (B + D)$ cycles. Set 2 and set 3 hold one locked cache line. Thus, they incur only a cache set read delay of $2 \times B + D$. In practice, the migration delay is $10 \times (B + D)$. However, for the worst-case scenario shown in Figure 3.13(b), set 0 incurs one cache set read delay and three cache migration delays accounting for $8 \times (B + D)$ cycles. The rest of the cache sets incur a cache set read delay of $2 \times (B + D)$ cycles. This results in a worst case migration delay of $14 \times (B + D)$.

Since migration delays for these schemes are deducible off-line, we compare the experimental delays and observe that Slotted-SSCM Pipelining is able to reduce the migration delay over SSCM on an average by 46.7%.

Figure 3.13: Offline Computation of Migration Delays for Slotted-SSCM



3.6.4 Parallel vs. Pipelined Cache Migration

In this Chapter, we introduced pipelined schemes like CCMP, SCMP and Slotted-SSCM with pipelining. When multiple cache migrations have to be handled, pipelined cache migrations

for each task can be performed one after the other. However, system-level parallelism of cache migration can also be obtained by issuing multiple instances of RCM and Slotted-SSCM cache migrations as shown in Figure 3.7. Pipelined cache migration is useful for reducing migration delay for individual migrations while parallel migrations can utilize maximum bandwidth. For example, our experimental model with cache access delays of 10 and bus delays of 2 cycles supports 5 parallel cache migrations for an aggregate bandwidth utilization of 100%.

Table 3.7: Parallel vs Pipeline

List of tasks migrating	Parallel Migration Cost [cycles]	Pipelined Migration Cost [cycles]	Scheduler's Choice of Migration
1,2,3,4	1142	1366	Parallel
1,2,4	1142	1252	Parallel
2,4	926	768	Pipeline
1,3,4	1142	992	Pipeline

When multiple task migrations occur, the scheduler needs to compare the cost of running a sequence of pipelined migrations against that of parallel migration. To illustrate, we choose a scenario where fft(1), jfdctint(2), bs(3), and crc(4) are running on a multicore system. We assume that they all use RCM as the base cache migration scheme and all of them can migrate in parallel when selected. Table 3.7 depicts four combinations that exhibit the behavior of parallel and pipelined migrations. The first column shows the set of migrating tasks. The second and third columns show the migration cost in cycles for parallel migration and serialized SCMPs. The last column shows the choice that the scheduler makes. It can be deduced from Table 3.7 that when the number of cache migrations are large and all the RCM migration costs are comparable, parallel migration exhibits shorter migration cost (rows 1 and 2). Pipelined migration performs better when the number of migrations are small (row 3). Parallel migration cost is determined by the highest individual migration cost. Hence, it performs worse than pipelined migration when the variance for the costs of individual migrations is high (row 4).

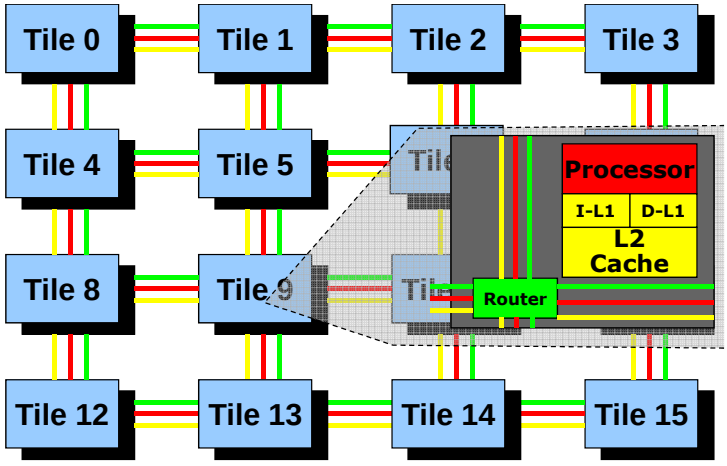
Chapter 4

Static Task Partitioning for Locked Caches in MultiCore Real-Time Systems

4.1 Introduction

The past two chapters have focused upon improving the predictability of cache migration cost that can be leveraged by global/semi-partitioning multi-processor scheduling algorithms. In the previous chapter, we discussed the potential of using locked caches in multicore real-time systems and the ability of such a hardware facility to make migrations predictable. In this chapter, we discuss the potential of cache locking for static partitioning algorithms. In general, cache locking techniques provide predictability to a task's cache access behavior. Cache locking can be realized at various granularities. Studies on uniprocessor cache locking have assumed the entire L1 cache to be locked [66, 67]. Another study on cache locking for shared caches has assumed locking of individual cache lines [88]. Locked caches on uniprocessors identify sets within a single cache way for a given task set to improve predictability and, indirectly, utilization/response time of tasks while ensuring schedulability on a single core. Most contemporary research aims at optimizing the analysis on aforementioned systems [25, 38]. These algorithms assume a shared cache space and use Integer Linear Programming algorithms in order to select cache lines that lower the response time of the tasks. Liu et al. have proposed locked caches for private L1 caches and partitioning for shared L2 caches of an SMP architecture [55]. They also focus upon finding the optimal response time of tasks on fixed numbers of cores. These algorithms may find reduced response time but they do not focus upon efficient distribution of the tasks on scalable CMPs like the 64 core TilePro64 [6]. E.g., let us assume that we have to partition 40 tasks on a 64 core CMP. Best response time can be achieved by allocating one

Figure 4.1: Tile-based Architecture



task per core. However, such a task distribution underutilizes the system in that a smaller set of cores could have sufficed.

In contrast, this work extends to scalable multicore architectures where tasks are statically partitioned. Our work focuses on distributing tasks over disjoint cores while considering their locked state. A real-time system developer may choose to lock a set of cache lines to tighten the WCET bound. This work uses these tightened WCET bounds to statically allocate tasks on a disjoint set of cores.

As stated earlier, research on cache contention has primarily considered shared caches. Due to the unscalability of such an architecture, the focus has been toward packing as many tasks on these limited/fixed number of cores. Such schemes become inapplicable to scalable multicores, such as shown in Figure 4.1. These architectures use private L1+L2 caches. Any task allocation algorithm on such architectures requires prior knowledge of each task's Worst Case Execution Time (WCET). However, the WCET of a task obtained by static cache analysis depends on cache analysis of all other tasks on a particular core. In this work, it is assumed that private L2 caches are large enough with high associativity (16-32 ways) to hold the data space and instructions of hard real-time tasks. This simplifies the analysis of L2 caches as any access to the L2 cache is a hit after a compulsory miss on warm-up. Thus, a tighter upper bound on the Worst Case Execution Time (WCET) can be established by modeling references resolved at the L2 level as hits after the warm-up phase of the first job execution in a periodic task system. Still, the access latency of L2 caches is an order of magnitude higher than that of L1 caches so that bounds on WCET are not as tight as they could be. To further tighten WCET bounds, cache locking of selected lines in L1 can be employed on scalable multicore platforms.

As stated before, prior literature on uniprocessor locking techniques focuses on filling a single

cache way while reducing the overall utilization of a core. Reduction of the system utilization can be achieved by placing all tasks with conflicting locked cache regions on different cores on scalable architectures. However, such a scheme would consume a large number of cores and result in under-utilization of computing resources. Also, multiple cache ways per L1 cache can be dedicated to locking. Hence, the objective of allocating tasks on scalable multicores has to be balanced between the following objectives:

1. Reduction of the number of cores; and
2. Reduction of the overall system utilization.

Static task partitioning has been considered as a viable scheduling option for real-time tasks on multiple cores. Such scheduling schemes aim at minimizing the number of cores for a set of tasks with given worst-case execution time (WCET). However, partitioning tasks with locked cache regions involves resolving the conflicts between locked regions of different tasks.

In this work, we split the problem into two scenarios.

1. Scenario A presents the problem with task sets of locked regions that can fit within a cache way. These task sets do not have any intra-task cache conflicts by design. It is further shown why naive solutions are inadequate. One of the most commonly used partitioning algorithm is the First Fit Decreasing (FFD) algorithm. First, we extend this algorithm with an approach called Naive Locked FFD (NFFD). Prior to allocation, NFFD decides to use cache locking for tasks that have prohibitively high utilization without locking. It avoids conflict analysis among locked regions by placing each locked task on a different core before allocating unlocked tasks using FFD. We call this algorithm cache-unaware as it avoids any form of analysis on locked cache regions. Then, we develop and evaluate two cache-aware partitioning algorithms: (1) Greedy First Fit Decreasing (GFFD), and (2) Colored First Fit Decreasing (CoFFD). GFFD tries to allocate tasks onto a minimum number of cores [18]. This scheme lacks prior information on the number of cores of a concrete processor but rather reasons abstractly about the minimum number of cores of a hypothetical processor design. CoFFD, a more sophisticated scheme, exhibits a novel approach based on graph coloring that delivers task partitioning. In contrast to GFFD, CoFFD initially assumes a given number of cores for an architecture. The algorithm then tries to allocate a given task set onto the fixed number of cores. In case of failure, the number of cores is incremented and the attempt to allocate tasks to cores is repeated. If the objective is to achieve minimum utilization, tasks should be allocated with all candidate regions locked as this lowers their WCET.
2. Scenario B looks into a more generic case: Tasks can have locked regions that cause intra-task conflicts and thus require multiple cache ways to avoid such conflicts. Also, static

analysis tools are capable of providing regional access frequencies. This allows us to lock and unlock parts of a task's address space. These changes render Scenario A solutions to be inadequate for task sets from Scenario B as explained in Section 4.5. We tackle this problem by splitting task partitioning into two phases: task selection and task allocation. task selection algorithms pick a task in some order and task allocation algorithms try to resolve regional conflicts at individual cores while allocating tasks onto them. We present two task allocation mechanisms, namely Regional First Fit Decreasing (RFFD) and Chaitin's Coloring (CC). We further present two task selection mechanisms, namely Monotonic (Mono) and Dynamic (Dyn). The task allocation algorithms are extrapolated from Scenario A solutions. Monotonic task selection is a commonly used FFD ordering, which delivers a monotonic order between tasks. However, our novel Dynamic scheme utilizes regional conflict information to order tasks while dynamically changing their order as tasks get allocated.

Table 4.1: Locking and Conflict Analysis for 32 Tasks

Number of Tasks	**Number of Cores Required**			
	FFD	**NFFD**	**GFFD**	**CoFFD**
High util.	Failed	32	22	20
Med. util.	31	31	21	20
Low. Util.	23	22	14	12

Table 4.1 depicts a comparison of the number of allocated cores for different **Scenario A task sets** of 32 tasks using FFD, NFFD, GFFD and CoFFD on an architecture that uses system parameters shown in Table 4.2. We consider two utilizations for each task: one with locking for all the regions specified by the developer (u_{locked}) and another without locking any of those regions ($u_{unlocked}$). A task is termed to be of high, medium and low utilization when ($0.55 > u_{locked} \geq 0.40$), ($0.40 > u_{locked} \geq 0.25$) and ($0.25 > u_{locked} \geq 0.10$), respectively. The first column depicts the number of tasks in the task sets. The remaining columns show the number of cores consumed by the task set under FFD, NFFD, GFFD and CoFFD, respectively. We observe that FFD fails to allocate high utilization task sets as $u_{unlocked}$ exceeds the utilization bound of 1 for such tasks. This is because it forces regions to be unlocked while the other policies allow locking. NFFD performs better than FFD for low utilization tasks as well. The table shows that the number of cores allocated by cache-aware schemes is significantly lower than the allocations performed by cache-unaware schemes. As the objective is to minimize the number of cores, the two algorithms are adapted to consider both u_{locked} and $u_{unlocked}$ during

allocation. The algorithms select one of these versions to avoid lock conflicts while ensuring that utilization constraints are met. We observe that CoFFD consistently results in allocating fewer cores than GFFD. Task sets composed of high utilization tasks allocate fewer cores under CoFFD with at most 3% higher system utilization than GFFD. For low utilization task sets, CoFFD allocates fewer cores and lowers system utilization by up to 40% over GFFD.

For task sets from Scenario B, the combination of Mono+CC outperforms Mono+RFFD for highly conflicted task sets. This shows the effectiveness of using the coloring mechanism at individual cores. In contrast, Mono+CC does not perform as consistently for low contention task sets. This necessitates the applicability of a global ordering scheme like Dyn to be used, which complements region level coloring at individual cores. Our results show that Dyn+CC consistently performs better than Mono+RFFD. For high contention task sets, Dyn+CC achieves up to a 22% reduction in number of cores allocated, i.e., it allocates 21 cores as opposed to 27 cores (Mono+RFFD). Even for low contention task sets, it is able to achieve a reduction of up to 17%. Since Dyn+CC deals with a more generic problem set, it is also applicable to task sets from Scenario A. While comparing the CoFFD with Dyn+CC, we observe that CoFFD performs better that Dyn+CC for High Utilization tasks. That is because unlocking a region or a task does not allow multiple tasks to be scheduled together. Thus, the global ordering achieved by CoFFD is better. However, CoFFD is unable to unlock entire tasks for medium utilization tasks either while the unlocking at regional granularity is feasible and thus Dyn+CC performs better. These observations suggest that CoFFD and Dyn+CC perform far better than naïve implementations of FFD-based task partitioning on distributed core CMPs.

Summary of contributions: This research makes the following contributions in the context of hard real-time systems with cache locking:

1. This work is the first to employ locked caches on massive multicore architectures for hard real-time systems.

2. We implement two task partitioning algorithms for Scenario A type task sets: GFFD and CoFFD. These algorithms resolve the conflicts at task level by selectively locking or unlocking tasks.

3. Our novel CoFFD algorithm (i) derives task allocations for a given number of cores resulting in a feasible schedule, (ii) enhances a coloring algorithm to deliver balanced allocation and (iii) reduces the number of cores relative to Greedy First Fit Decreasing (GFFD).

4. For Scenario B, we propose a novel mechanism to resolve conflicts at the granularity of regions. We propose a Dynamic (Dyn) ordering mechanism that adapts to the changes in the regional conflict graph induced by the allocation of tasks to cores. Dynamic order-

ing consistently allows the allocation of tasks on fewer cores with both Task Allocation algorithms. Dyn+CC proves to be the best among all the combinations of Task Selection and Task Allocation Schemes.

4.2 Related Work

In the past decade, there has been considerable research promoting locked caches in the context of multitasking real-time systems. Static and dynamic cache locking algorithms for instruction caches have been proposed to improve system utilization in [66, 65]. Several methods have been developed to lock program data that is hard to analyze statically [93]. Further techniques have been developed for cache locking that provide performance comparable to that obtained with scratchpad allocation [67]. Recently, cache locking has also been proposed for multicore systems that use shared L2 caches [88]. Liu et al. propose cache locking for private L1 caches while using cache partitioning for L2 caches [55]. Their focus has been upon reducing the task utilization while partitioning a task set on all the processor cores in the system. This is applicable to unscalable shared cache architectures. In contrast, our work focuses upon optimizing allocation of computational resources. These related efforts show that cache locking is a viable solution in future real-time system designs for multicores. Guan et al. propose L2 cache partitioning using cache coloring for soft real-time systems [38]. Paolieri et al. have proposed hardware cache partitioning mechanisms for multicore real-time systems with shared L2 caches [63]. However, the focus of both of these cache partitioning schemes is to pack as many real-time tasks as possible on an unscalable multicore architecture.

4.3 System Design

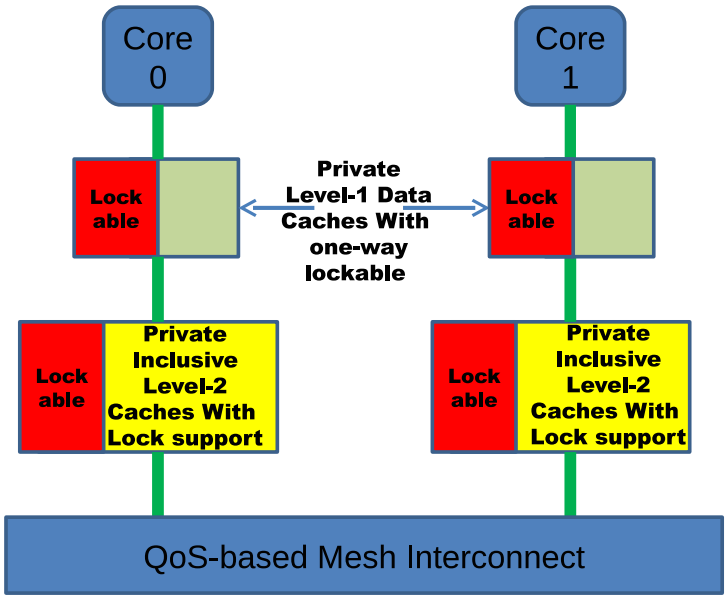
In this section, we describe our system architecture and assumptions to WCET analysis for this study. The objective of this work is to best utilize a private cache architecture. This corresponds to the current trend in potentially mesh or tile-based multicore designs. Tile-based architectures consist of a large number tile processors (cores). Each tile consists of an in-order processor, a private L1, a private L2 cache and a router (see Figure 4.1). Each tile acts as a node on a mesh interconnect. Recent work has added Quality-of-Service (QoS) policies to mesh-interconnects [61]. We have identified these trends as the driving force for the simplification of our system. We assume an architecture that has private caches and has a QoS-based interconnect. We assume that the first level of cache allows a certain number of ways of the associative cache to be locked as shown in Figure 4.2. We also assume that the L2 caches are large enough with high associativity so that the address space of allocated hard real-time tasks on a core fit within the L2 cache. Thus, we assume that the off-chip references occur only while accessing sensory

data, which accounts for a very small fraction of the total references. Also, these systems can have inclusive or non-inclusive L2 caches. With inclusive caches, the locked regions in L1 need to be locked in L2 as well. Our algorithms are applicable to a system considering both data and instruction caches. However, for the simplicity of analysis we assume that instruction references for hard real-time tasks are all hits at the first level of cache. We also assume that loads to the lines that have not been locked in the L1 cache bypass the L1 cache (as in a previous research work [41]). This allows cores with lower core utilization to co-schedule non-real-time tasks along with hard real-time tasks without affecting the deterministic behavior of the latter. Such hybrid execution of application tasks has been considered in recent research [62]. Here, we analyze two scenarios

1. A hard real-time task can only lock one cache line per set. All the locked regions of a task can fit within a direct mapped L1 cache. So, for a 8KB L1 cache with an associativity of two, a hard real-time task can lock up to 4KB of cache content. We call this Scenario A.

2. A hard real-time task can lock multiple cache lines per set. Here, conflicting locked regions are able to occupy multiple cache ways. So, for a 8KB L1 cache with an associativity of two, a hard real-time task can lock all of the 8KB cache space or any subset at cache line granularity. We call this Scenario B.

Figure 4.2: A Lock-based Architecture



We assume that all hard real-time tasks are periodic. Each task's deadline is the same as its period, i.e., an invocation of a task's job has to finish before its next invocation. We further assume that the system runs a scheduler per core. Each of these schedulers independently schedules the tasks allocated to this core. We assume them to utilize Earliest Deadline First (EDF) scheduling. EDF optimally schedules tasks for uniprocessor, i.e., the utilization bound for each core is defined by the following equation: $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$, where C_i and P_i are the WCET and the period of the i^{th} task, respectively. Deadlines are assumed to be the same as the periods.

For the algorithms, each task needs to provide the following information: $<list_{locked-sets}$, $WCET_{locked}$, $WCET_{unlocked}>$. $list_{locked-sets}$ is the list of sets where the programmer intends to lock a cache line for the task. $WCET_{locked}$ and $WCET_{unlocked}$ are the WCETs of a task when all the lines of $list_{locked-sets}$ are locked and unlocked, respectively. $WCET_{locked}$ does not include the overhead of loading the contents of a task because it is a one-time cost incurred at system start-up.

We also assume that the real-time tasks are pairwise independent. Hence, these tasks do not cause any coherence traffic on the interconnect.

4.4 Task Partition Algorithms: Scenario A

4.4.1 Cache-Unaware Schemes

Static task partitioning algorithms for multicore architectures have been widely studied. Most of these approaches consistently aim at minimizing the number of cores utilized [18]. They use bin-packing schemes considering a single utilization value per task. These algorithms for distributed systems are cache unaware. In the following section, we present two cache-unaware schemes, namely FFD and NFFD.

First Fit Decreasing (FFD)

FFD is a commonly used algorithm for allocating tasks on distributed cores. This implementation assumes that the tasks are unlocked, i.e., we consider all tasks with a utilizations of $u_{unlocked}$ using $WCET_{unlocked}$. This algorithm takes task (i), already allocated set of cores N_{procs} and a flag that decides whether task to be allocated in a locked state or unlocked state if it adds a new core to N_{procs}. The FFD algorithm picks tasks in decreasing order of their $u_{unlocked}$ and allocates them using Algorithm 9. Line 1 sorts the cores in N_{proc} in decreasing order of core utilization. Lines 3-8 iterate over the cores until the task is allocated or until all cores have been considered and task could not be allocated. A task is allocated to a core if a core's utilization does not exceed 1 (utilization bound for EDF). If a task could not be allocated

to any core in N_{procs}, lines 9-13 add a new core to N_{procs} and the task is allocated to it in an unlocked state.

ALGORITHM 2: FFD Task Allocation (baseFFD)

```
   Input: i : task, N_procs : processors, isLock: boolean
   Output: N_procs number of processors
 1 N_procs.sort(decreasing utilization) ;
 2 foreach N_procs j do
 3 |   if Success = false then
 4 |   |   if i.u_unlocked ≤ 1 - j.u then
 5 |   |   |   allocate task i to core j;
 6 |   |   |   j.u = j.u + i.u_unlocked;
 7 |   |   |   Success := true ;
 8 |   |   |   break;
   |   |   end
   |   end
   end
 9 if Success = false then
10 |   allocate New_proc;
11 |   N_procs := N_procs ∪ New_proc;
12 |   allocate task i to New_proc;
13 |   if isLock = true then
   |   |   New_proc.u := i.u_locked;
   |   end
   |   else
   |   |   New_proc.u := i.u_unlocked;
   |   end
   end
```

Naive Locked FFD (NFFD)

We extend FFD with a simple approach of using locked caches. Tasks are defined to be locked or unlocked prior to their allocation. Thus, all the tasks have a single WCET before allocation, which is $WCET_{locked}$ for a locked task or $WCET_{unlocked}$ otherwise. Bin packing has difficulties to co-locate multiple tasks with high utilization. Any task whose utilization is greater than a certain threshold is deemed to be locked. Each of these locked tasks is allocated to a separate core as the algorithm is cache-unaware. The algorithm proceeds to allocate the set of unlocked tasks with an initial value of N_{procs}, the number of cores assigned to locked tasks.

4.4.2 Cache-Aware Task Partitioning

We next present two cache-aware mechanisms. Initially, our algorithms consider two values, $WCET_{locked}$ and $WCET_{unlocked}$.The $list_{locked-sets}$ item is used to deduce a conflict matrix M_{conf} for locked tasks. A conflict among the locked sets indicates the existence of common locked cache set(s). Each empty entry in $M_{conf}(i, j)$ signifies the absence of conflicts between tasks i and j while every filled entry signifies existence of a conflict.

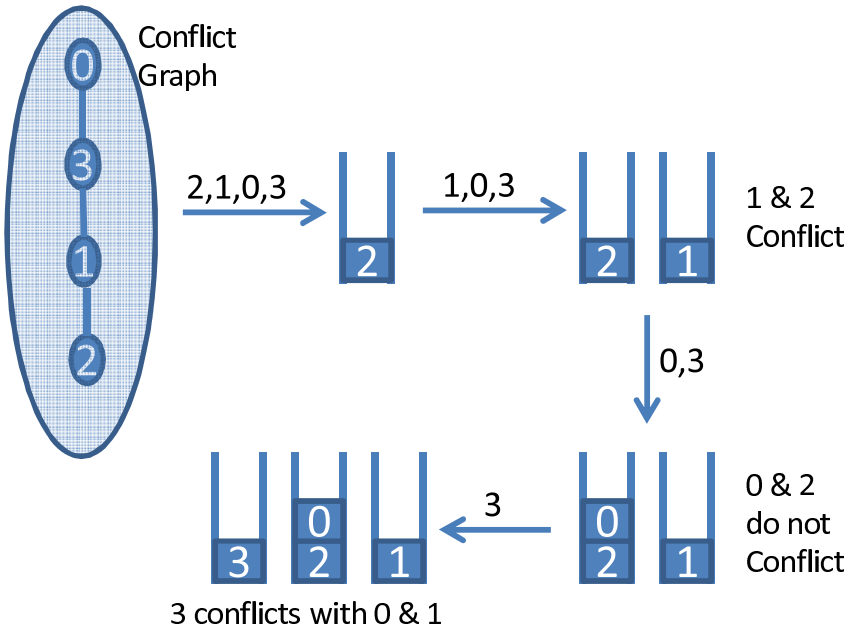
ALGORITHM 3: Greedy First Fit Decreasing Heuristic (GFFD)

```
   Input: M : Set of Tasks, Assoc : Number of locked ways per cache, M_conf : conflict
          Matrix
   Output: N_procs number of processors
 1 N_procs := 1 ;
 2 M.sort(decreasing u_locked);
 3 while M is not empty do
 4 |   Success := false ;
 5 |   N_procs.sort(decreasing utilization) ;
 6 |   i := M.front;
 7 |   foreach N_procs j do
 8 |   |   if k:=IsAllocatable(j,i,Assoc,M_conf) ≠ −1 then
 9 |   |   |   if i.u_locked ≤ 1 - j.u then
10 |   |   |   |   allocate task i to core j in kth way;
11 |   |   |   |   j.u = j.u + i.u_locked;
12 |   |   |   |   Success := true ;
13 |   |   |   |   break ;
   |   |   |   end
   |   |   end
   |   end
14 |   if Success = false then
15 |   |   N_procs := baseFFD(i, N_procs,true);
   |   end
   end
```

Greedy First Fit Decreasing (GFFD)

We first illustrate GFFD by example using a conflict graph in Figure 4.3. An undirected conflict graph of four nodes/vertices is depicted in Figure 4.3. A conflict graph in the context of task partitioning is a graph $G = (V; E)$, where every vertex/node $v \in V$ corresponds uniquely to a task and an $edge(i; j) \in E$ indicates that tasks i and j are in conflict and cannot be allocated

Figure 4.3: Greedy First Fit Decreasing in Operation



onto the same core. The objective is to map nodes into buckets while keeping the number of buckets low. The FFD algorithm arranges nodes in traversal order via heuristics before allocating them. In this example, the algorithm establishes an allocation order of nodes 2, 1, 0 and 3. At each step, the node in question checks if it can be placed within any of the existing buckets. A node can be allocated to a bucket if the bucket does not contain any node that conflicts with it. In the example, node 0 gets allocated to a bucket that contains node 2, which does not conflict with 0. In case all buckets conflict, a new bucket is created, e.g., during the allocation of nodes 1 and 3.

We developed a modified version of the FFD algorithm. We call this Greedy First Fit Decreasing (GFFD). Algorithm 3 presents the details of the algorithm. This algorithm takes a task set and the number of locked ways per cache as an input. The idea is to incrementally add cores to the schedule starting with an initial number of cores, N_{procs}, of 1. Lines 3-13 proceed to allocate tasks in FFD fashion using u_{locked}. Line 8 uses a procedure IsAllocatable() that returns the cache way that is still unassigned to any locked lines of tasks that conflict with any locked lines of task i. In case a valid cache way is found and the allocation of the task with the locked region passes the schedulability test, the task is allocated to the core. If, however, all the lockable cache ways of the core's L1 are in conflict or the schedulability test fails, the algorithm tries to allocate the task to another core until it runs out of cores in N_{procs}. If the task remains unallocated, line 15 uses Algorithm 9 to allocate the task. The value of *true* for

the third parameter to $baseFFD$ forces the task to be allocated in locked state when a new core is added to N_{procs}.

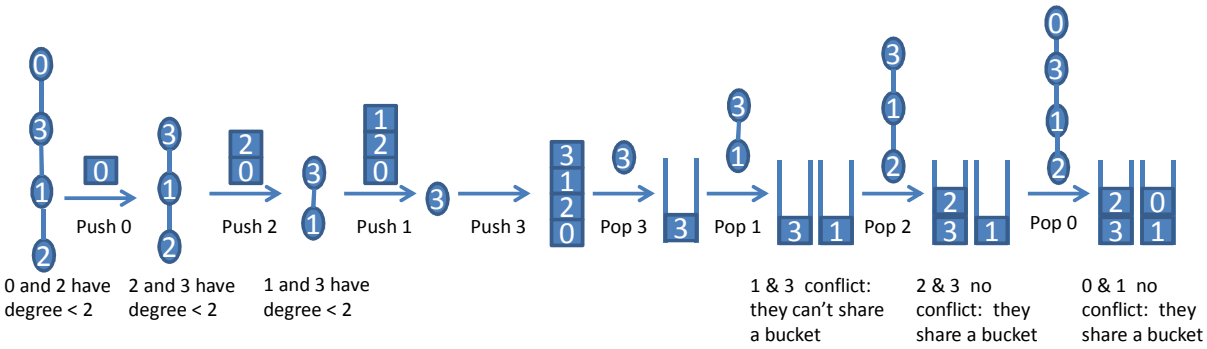
GFFD identifies task conflicts only after a task has been committed for allocation, even though a conflict matrix is already present. The algorithm does not have a prior notion of the number of cores available within the system. Furthermore, the order in which tasks are assigned to cores is still based on task utilization. We can do better. When tasks contend for cache regions, analysis of the cache conflict graph yields superior, conflict-guided allocations. Such analysis considers tasks in a conflict-conscious order that ensures they can co-exist with each other for a given number of cores. To this end, we adapted a graph coloring approach by Chaitin [21, 20] that is widely used in register allocation, which is based on the following theorem:

Chaitin's Theorem 1 *Let G be a graph and v ∈ V(G) such that deg(v) < k, where deg(v) denotes the number of edges of vertex v. A graph G is k-colorable if and only if G - v is k-colorable.*

This theorem provides the basis for graph decomposition by repeatedly deleting vertices with degree less than k until either the graph is empty or only vertices with degree greater than or equal to k are left. In the latter case, the graph cannot be colored. However, by removing a task from a conflict graph using some heuristic, a new coloring attempt can be made for the remaining of the graph. Figure 4.4 shows how Chaitin's theorem can be used in practice. In this example, the conflict graph is the same as in the FFD example in Figure 4.3. This new example shows how Chaitin's approach allocates the set of nodes to two buckets/colors. At first, the algorithm fills up a stack removing one node at a time. A node is a viable candidate for being pushed onto the stack if and only if the degree is less than 2. When a node is removed, it reduces the degree of its neighbor in the remainder of the graph. Since all nodes can be pushed onto the stack, the graph is two-colorable (cf. Chaitin's theorem). During the following steps, nodes are popped off the stack and associated with a color/bucket. In our example, Chaitin's algorithm successfully allocates nodes to two buckets. In contrast, three buckets were required by the FFD algorithm.

Algorithm 14 shows the task coloring mechanism, which is responsible for finding non-conflicting tasks that can be grouped together in a given number of colors. The number of colors is equal to the number of locked cache ways that can be filled within a given number of cores. Lines 4-13 fill up two data-structures, $colorStack$ and $spilledList$. Every iteration of this loop finds a task that can be placed on either of these stacks. Line 5 searches through the list of unallocated tasks and finds the task with lowest degree. A task with minimum degree is pushed onto $colorStack$ if and only if its degree is less than $NumOfColors$. Otherwise, the algorithm finds a task using a heuristic that focuses on minimizing a metric. For example in

Figure 4.4: Chaitin's Coloring in Operation with 2 Colors



Algorithm 14 it minimizes the metric $u_{locked}/degree$ at line 10. The objective of this heuristic is to decrease the conflict degrees of as many tasks as possible and, at the same time, to pick a task that causes the minimum increase in the system utilization while remaining unlocked ($u_{unlocked}$). This task is then added to the *spilledList*. While removing the tasks from M, we decrease the conflict *degree* of neighbors.

Colored First Fit Decreasing (CoFFD)

Once all tasks have been distributed among either of the stacks, lines 13-27 put the tasks in *colorStack* into different colorLists. Assigning a task from *colorStack* to a *colorList* is equivalent to allocating the task to a core as each color corresponds to a lockable cache way. The *colorList*s are associated with cores in a round robin manner, i.e., if the number of lockable cache ways per task is equal to two and the number of cores is three, then there are a total of six *colorList*s. The first, second and third *colorList*s are associated with the first cache way on cores one, two and three, respectively. The fourth, fifth and sixth *colorList*s are associated with the second cache way on cores one, two and three. Lines 15-16 pop a task from the *colorStack* and re-populate the conflict edges in the graph with the tasks that have already been colored. The algorithm then loops through all the colors until it finds a color that has not been allocated to any of its neighbors in the graph. Line 20 picks the core associated with that color. For a task to be assigned a color, the task has to pass the EDF schedulability test.

Furthermore, the current utilization of the core has to be less than *aveCoreUtil*, where *aveCoreUtil* has been computed at line 14. These conditions prevent *colorList*s from becoming unbalanced. Chaitin's algorithm in its purest form is

- unaware of the tasks in the *spilledList* and
- unable to deliver a balanced *colorList*.

ALGORITHM 4: Task Coloring Algorithm

Input: M : Set of Tasks, $NumOfColors$: Number of Cores × Number of locked ways per cache, M_{conf} : conflict Matrix

Output: $colorList$, $spilledList$, $rejectedTaskList$

```
1  colorStack := empty;
2  spilledList := empty;
3  colorList := empty;
4  while M is not empty do
5  |  t := lowest degree task by linear search of M ;
6  |  if t.degree < NumOfColors then
7  |  |  push t onto colorStack ;
8  |  |  remove t from M and M_conf ;
   |  end
9  |  else
10 |  |  t := task with minimum (u_unlocked/degree) ;
11 |  |  push t onto spilledList ;
12 |  |  remove t from M and M_conf ;
   |  end
   end
13 aveCoreUtil = colorStack.u / NumOfColors;
14 while colorStack is not empty do
15 |  t := Pop colorStack ;
16 |  repopulate M_conf ;
17 |  curColor:=0;
18 |  for curColor = 0 → NumOfColors − 1 do
19 |  |  if None of the neighbors has this color then
20 |  |  |  curCore := curColor  mod  number Of Cores ;
21 |  |  |  if curCore.u < aveCoreUtil and curCore.u + t.u ≤ 1 then
22 |  |  |  |  t.color := curColor ;
23 |  |  |  |  colorList[curColor] := t ;
24 |  |  |  |  Add t.u to curCore.u ;
25 |  |  |  |  break ;
   |  |  |  end
   |  |  end
   |  end
26 |  if t.color is not a valid Color then
27 |  |  push t onto rejectedTaskList ;
   |  end
   end
```

E.g., if none of the tasks are conflicting then all tasks can be given the same color. Conditions at line 21 allow the tasks to be evenly distributed across cores. If either of the conditions fail, then the algorithm moves on to the next color until all the colors have been tried. If a task cannot be assigned a valid color, it is moved to $rejectedTaskList$.

ALGORITHM 5: Colored First Fit Decreasing (CoFFD)— Uncolored Lists

```
   Input: rejectedTaskList, Assoc : Number of locked ways per cache, M_conf : conflict
          Matrix,N_procs : number of cores
 1 rejectedTaskList.sort(decreasing u_locked);
 2 foreach rejectedTaskList i do
 3 |   N_procs.sort(decreasing u);
 4 |   Success = false;
 5 |   foreach N_procs j do
 6 |   |   foreach Assoc k do
 7 |   |   |   if IsAllocatable(j,i,Assoc,M_conf) ≠ −1 then
 8 |   |   |   |   allocate task i to core j in kth associativity;
 9 |   |   |   |   j.u = j.u + i.u_locked;
10 |   |   |   |   Success = true;
11 |   |   |   |   goto ;
   |   |   |   end
   |   |   end
   |   end
12 |   if Success==false then
13 |   |   put task i on spilledList ;
   |   end
   end
14 spilledList.sort(decreasing u_unlocked);
15 foreach SpilledList i do
16 |   if N_procs ≠ baseFFD(i,N_procs,false) then
17 |   |   return Failed Allocation;
   |   end
   end
18 return Successful Allocation;
```

The task coloring stage outputs partially filled cores and a list of tasks in $rejectedTaskList$ and $spilledStack$. These are subsequently used by the second part of the allocation shown in Algorithm 18. Algorithm 18 first tries to allocate tasks from the $rejectedTaskList$. It sorts the tasks of $rejectedTaskList$ in decreasing order of their u_{locked}. Each iteration of the loop starting at line 2 then picks a task in order and tries to allocate it in FFD fashion on N_{procs}. If a

task cannot be allocated to a core, it is moved to the *spilledList*. Once the *rejectedTaskList* is empty, all the tasks in *spilledList* are allocated using $baseFFD$. If all the tasks in *spilledList* are allocated, the task set is deemed to be schedulable on a given number of N_{procs} cores. Otherwise, N_{procs} is incremented by the caller of CoFFD. This process repeats until a schedule has been found.

Figure 4.5: Task Coloring in Operation

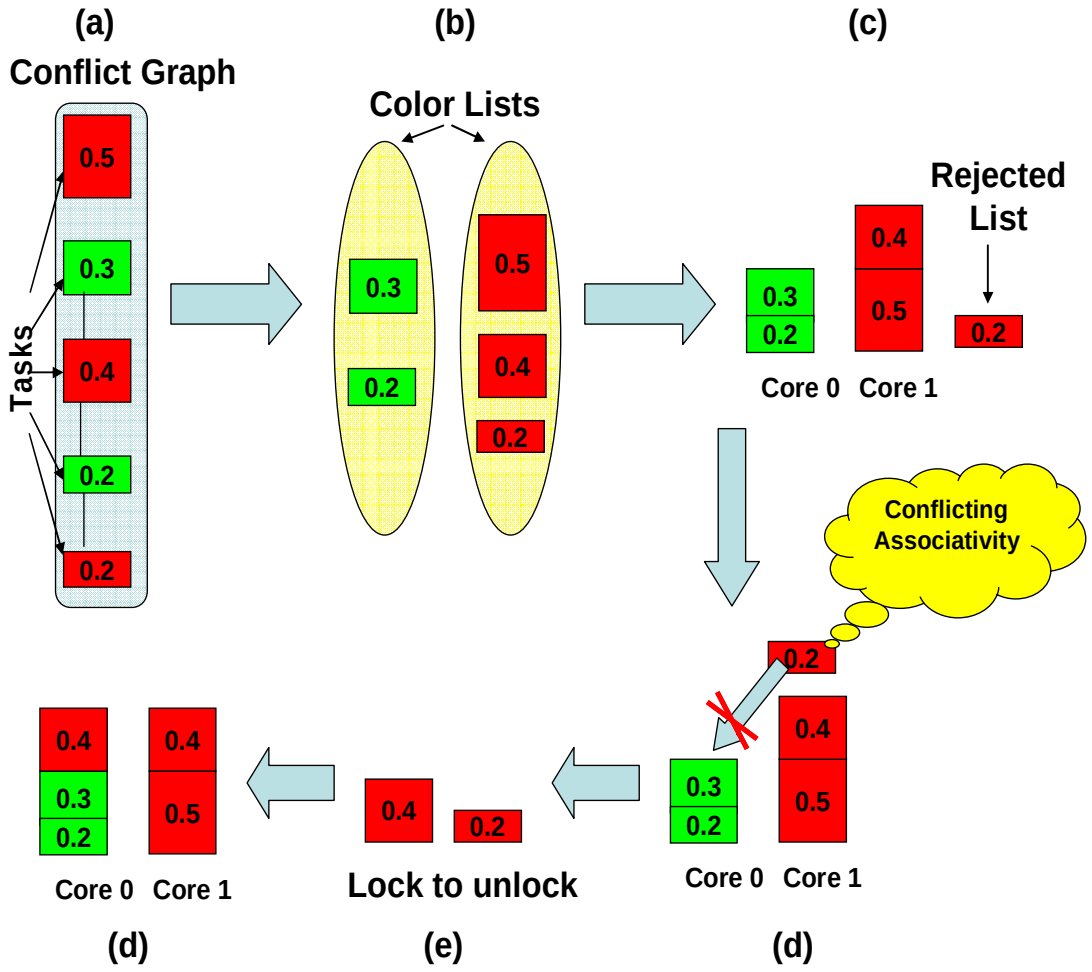


Figure 4.5 depicts a step-by-step working example:

(a) Tasks are grouped in a conflict graph. Our example has five tasks with u_{locked} utilizations of 0.5, 0.3, 0.4, 0.2 and 0.2. Each task conflicts with its neighboring task. Therefore, tasks form a chain of conflicts in the graph.

(b) Our graph coloring algorithm is applied to split the tasks in $ColorLists$. The task set is split into two colors alternating between adjacent tasks in the same $colorList$.

(c) We assume a multicore system with single-way locking in the L1 cache. Since the aggregate utilization is 1.6, N_{procs} is initialized with the ceiling of system utilization, which is 2. The tasks in each $colorList$ are sorted in decreasing order of u_{locked}. The cores are filled in a round-robin fashion. The green $colorList$ fits within core zero. Tasks in the red $colorList$ are allocated to core one. Tasks with higher utilization (0.5 and 0.4) are allocated to core one while the task with utilization 0.2 is moved to the $rejectedTaskList$ as it exceeds the utilization bound of 1.

(d) The algorithm now tries to allocate the task from $rejectedTaskList$ to core zero. It fails due to task conflicts with an already allocated task and due to the availability of only one cache way for locking.

(e) At this stage, the task is moved to the $spilledList$. The task's utilization is increased to $u_{unlocked}$. This changes its utilization from 0.2 to 0.4.

(f) The task is allocated on core 0 with this inflated utilization because such allocation does not violate the utilization bound on core 0.

Applicability to task sets from Scenario B: Algorithms discussed till now assume that the locked cache regions of a task fit within a single cache way. Tasksets from Scenario B have intra-task conflicts within locked regions. The following are two techniques to apply the NFFD, GFFD and CoFFD policies to task sets from Scenario B:

1. Lock only one region out of all the conflicting regions: This could significantly increase the $WCET_{locked}$, which may render a task unschedulable.

2. Retain all the locked regions of a task that can fit within the whole cache while treating all the conflicting regions as one region that spans from the least indexed set to the maximum indexed set: Each core's cache can be treated as a direct-mapped cache, where a region is assumed to be spread across all the sets. This effectively partitions the cache horizontally. This solution does not have a notion of cache associativity and leads to inefficient task allocations.

As stated above, both techniques are highly inefficient. This motivates us to develop algorithms specifically for Scenario B. We present them next.

4.5 Task Partition Algorithms: Scenario B

The algorithms presented in this section assume tasks to have intra-task conflicts and assume that a task can use multiple cache ways as described in Section 4.3. So far we have assumed that conflicting tasks can only share a resource either by locking all specified regions or keeping all of them unlocked. This is useful when locked regions should remain transparent to the programmer. We can improve on our results if programmers can accurately estimate the upper bound on the number of references to each locked cache line (e.g., based on upper loop bounds). This can be achieved through static analysis of tasks i.e., by specifying the number of references (N_{refs}) for each locked cache region in $list_{locked\_set}$. We can then compute the reference frequency, R_f, of a locked region for task t as

$R_f = \frac{N_{refs}}{Period_t}$.

This becomes the basis for resolving conflicts at a finer granularity. However, the prior solutions do not provide us enough flexibility as conflicts are resolved at task level. The task partitioning algorithm for Scenario B is being split into two phases: task allocation and task selection.

4.5.1 Task Allocation

The task allocation mechanism resolves conflicts between regions of task t and the tasks that have already been allocated onto the core. We use two heuristics for resolving regional conflicts, namely:

1. Regional First Fit Decreasing (RFFD): RFFD allocates a list of regions to a given cache associativity. The regions are sorted in decreasing order of their R_f. Regions are picked for allocation in that order. The algorithm then attempts to allocate a region to every cache way until it finds one where it does not conflict with any allocated region. If the regional allocation fails then it is considered unlocked. Once all the regions have been considered for allocation, a dilated WCET is computed based upon the unallocated regions. A task with dilated WCET is deemed to be allocatable if the utilization bound of 1 is not exceeded. Otherwise the task allocation fails on that core. Please note that when a new task is being considered for allocation, all the regions of the previously allocated tasks are reallocated along with the regions of the new task.

2. Chaitin's Coloring algorithm (CC): Chaitin's algorithm is used in its original form For region allocation using conflict graph for regions as depicted in Figure 4.4. The number of colors is the cache associativity. All the regions that have not been colored are unallocated regions. A task is deemed to be allocated if the utilization bound of 1 is not exceeded. Otherwise task allocation fails on that core. Similar to RFFD, allocation of regions is

not permanent. Regions are reallocated whenever another task is being considered for allocation on the core.

4.5.2 Task Selection

As for the task Selection, we use the following two heuristics:

1. Monotonic Order (Mono): This uses task selection procedure of similar to GFFD, where tasks are picked in the decreasing order of $WCET_{locked}$. Then, each task is considered for allocation on every core until a core is found where it can be allocated along with the tasks that have already been allocated. Otherwise, a new core is added to the list of cores, which is then considered during allocation of current and subsequent tasks.

2. Dynamic Order (Dyn): Monotonic ordering using $Util_{locked}$ and $Util_{unlocked}$ is oblivious of conflicts. Here, we present a scheme that orders tasks according to their $Util_{probabilistic}$ and dynamically adjusts to changes in the regional conflict graph caused by allocation of tasks to cores. In order to obtain $Util_{probabilistic}$, we first generate a directed graph of conflicting regions. A conflicting region X is considered to be replaced by another region Y if Y's R_f is greater than that of X. The edge in the conflict graph runs from Y to X. So this adds an incoming edge to region X and an outgoing edge from region Y. If a majority of conflicting regions replace a region then it is considered to be unlocked for the purpose of calculating $Util_{probabilistic}$ at task selection stage. $Util_{probabilistic}$ is obtained by the following equation:

   $Util_{locked} + (\sum_{i=0}^{unlocked\ regions} R_{f_i}) \times latency_{lower\ cache}$.

   The task with the highest $Util_{probabilistic}$ is picked for allocation. Inspired from Chaitin's coloring algorithm, every task allocation can be used to dynamically change the state of the graph. We observe that with our greedy allocation process, when a task allocation fails, the remaining tasks get allocated to the newly created core more often than to previously existing ones. Hence, the directed graph should contain the conflicts pertaining to the unallocated regions and the regions allocated to the newly created core, only. Algorithm 25 shows the dynamic process. The $Util_{probabilistic}$ of tasks is initialized based upon the initial graph containing all the regions. As stated earlier, the task selector picks the task t that has the largest $Util_{probabilistic}$ through a linear search over the list of tasks. Our algorithm is then called to allocate t, which changes the state of the graph as explained below.

   The algorithm begins by declaring $UremTasksFromGraphs$, which holds those tasks that have been allocated but have not been removed from the M_{conf}. The loop starting at line 2 iterates till all tasks have been allocated to a core in N_{procs}. On each iteration, one task is allocated. Line 3 picks a task t that has the highest $Util_{probabilistic}$ using a

linear search over the list of tasks M. Lines 4 through 8 try to allocate task t to any one of the N_{procs} cores. If $TaskAllocation(t)$ succeeds, then the task is allocated to core i. Task t is added to $UremTasksFromGraphs$, which is a list of allocated tasks that have not been removed from the directed graph M_{conf}. It then goes back to line 2 to start allocating another task. In case $TaskAllocation(t)$ fails to allocate the task to any of the N_{procs} cores, lines 9 and 10 add a *new* core to N_{procs} and allocate task t to the *new* core. At this point, $UremTasksFromGraph$ contains all the allocated tasks since the last addition of a new core. The loop starting at line 11 iterates over each of those tasks using an iterator rt while removing them from $UremTasksFromGraph$. The nested loop beginning at line 12 populates rr with every iteration from the list of regions owned by rt. In line 13, one region is selected at a time, denoted as cr, which conflicts with rr. For clarification, rr and cr are regions removed from and resident within the directed graph, respectively. Lines 15 through 16 remove these edges by decrementing $NConfs$ values for the corresponding regions. If the edge is an incoming edge for cr, then line 18 decrements the incoming edge count. The ratio $\frac{numberofIncomingedges}{numberofedges}$ gives an approximate indication of whether a locked region will remain locked or not when allocated. Lines 14 and 19 compute this ratio for cr before and after removal of an edge as $priorProbability$ and $currentProbability$, respectively. If this ratio is > 0.5, then for the purpose of calculating $Util_{probabilistic}$ it is assumed to be unlocked. Thus, if this ratio changes from being > 0.5 to ≤ 0.5, then the increase in utilization due to unlocking is added to $Util_{probabilistic}$ to the task that owns cr. However if the ratio changes from being ≤ 0.5 to > 0.5, then $Util_{probabilistic}$ is being reduced. Lines 20 through 23 perform these changes accordingly. Line 24 adds this task t to $UremTasksFromGraphs$. At this point, $UremTasksFromGraphs$ holds only one task. The control returns back to line 2 for allocation of another task where a task is selected based upon these changes in $Util_{probabilistic}$. Once all the tasks have been allocated, N_{procs} is returned.

Solutions for Scenario A can be used in several ways. If tasks can meet their deadlines only under locking with $WCET_{locked}$, then these algorithms will allocate them with $WCET_{locked}$. If $WCET_{locked}$ and $WCET_{unlocked}$ are provided, then both fully locked and fully unlocked scenarios can be assessed by the algorithms. Dealing with execution times at coarser levels seems more attractive to the developers. This allows them to select lockable lines with rough estimates of the access patterns.

Solutions for Scenario B allow us to tackle a more generic case where a task may lock multiple locked regions that may require multiple cache ways to accommodate the locked address space for a task. These algorithms also utilize memory access frequencies that static analysis tools, such as [70], are capable of deducing. This allows us to partition tasks while resolving conflicts among regions within a core. These solutions are also applicable to Scenario A and we present

ALGORITHM 6: Task Selection: Probabilistic Utilization

Input: M : Set of Tasks, $Assoc$: Number of locked ways per cache, t: allocated task , M_{conf} : Directed Conflict Graph of Regions, $InEdges$:Incoming Edges per Region, $NConfs$: Number of Conflicts per Region,

Output: N_{procs}: Number of processors

```
1  declare UremTaskFromGraphs: Unremoved Tasks from M_conf;
2  while M is not empty do
3      t = linear search to pick task with highest U_prob ;
4      foreach N_procs i do
5          if TaskAllocation(t)isSuccessful then
6              allocate t → i;
7              UremTaskFromGraph := UremTaskFromGraph ∪ t;
8              goto 2;
           end
       end
9      N_procs:=N_procs ∪ new;
10     t → new;
11     foreach UremTaskFromGraph rt do
12         foreach rt.lockedRegions rr do
13             foreach M_conf cr do
14                 priorProbability := InEdges[cr]/NConfs[cr];
15                 NConfs[cr]:=NConfs[cr] - 1;
16                 NConfs[rr]:=NConfs[rr] - 1;
17                 if M_conf[ct][rt]==-1 then
18                     InEdges[cr] := InEdges[cr] - 1;
                   end
19                 currentProbability := InEdges[cr]/NConfs[cr];
20                 if priorProbability ≤ 0.5 ∧ currentProbability > 0.5 then
21                     cr.task.U_prob = cr.task.U_prob + cr.U_unlocked − cr.U_locked;
                   end
22                 if priorProbability > 0.5 ∧ currentProbability ≤ 0.5 then
23                     cr.task.U_prob = cr.task.U_prob − cr.U_unlocked + cr.U_locked;
                   end
               end
           end
       end
24     UremTaskFromGraph:= UremTaskFromGraph ∪ t;
   end
25 return N_procs;
```

our experimental observation in the following sections.

4.6 Algorithmic Complexity

Bin packing is known to be NP-hard. Any known optimal solution is exponential in complexity. Besides experimental evaluations, it is important to assess the complexity of sub-optimal, heuristic approaches to assess their scalable in terms of number of tasks and cores. In the following, the algorithmic complexity of GFFD and CoFFD are assessed.

For the purpose of complexity analysis, let the number of tasks be X and the number of cores be Y. Let t be the task to be allocated next.

GFFD: The outer loop in algorithm 3 iterates over all tasks. The inner loop from 8-13 iterates over all cores. The function $IsAllocatable$ iterates over the task to task conflict set, M_{conf}, bounded by the number of tasks, to detect if t conflicts with any of the tasks allocated to a core, i.e., $IsAllocatable$ has an algorithmic complexity of $O(X)$. Thus, the combined algorithmic complexity of GFFD is $O(YX^2)$).

CoFFD: CoFFD consists of algorithms 14 and 18. The former algorithm colors the tasks while allocating them to cores. It has two loops. The first loop between lines 4 and 12 iterates over all tasks. The nested computations of linear search at line 5, reduction of number of conflicts for tasks conflicting with t at line 8 and 12, and linear search at line 10 are bounded bounded by the number of tasks, i.e., they have an algorithmic complexity of $O(X)$ for a combined complexity of $O(X^2)$ for the first loop. The second loop between lines 14 and 27 iterates over all tasks while pushing them onto a stack. The nested loop within iterates over the set of colors, which is bounded by the number of cores, Y. The nested conditional at line 19 iterates over the set of neighboring nodes in the repopulated graph whose cardinality is bounded by the number of tasks, X. This implies an algorithmic complexity of $O(YX^2)$ for the second loop, which dominates the complexity of the first loop, i.e., is the overall algorithmic complexity of algorithm 14. The algorithmic complexity of algorithm 18, sequentially invoked next, is $O(YX^2)$ following the same argument as for GFFD since their algorithmic structure are equivalent in terms of loop iterators, i.e., the rejected task list is bounded by the number of cores. The loop iterating over the spilled list is bounded by the number of tasks but its complexity is dominated by the previous loop. Thus, the algorithmic complexity of CoFFD is $O(YX^2)$. If this algorithm fails to allocate tasks to a given number of cores, the algorithm is repeated with an incremented number of cores. This increases the complexity of CoFFD to $O(Y^2X^2)$.

Mono+RFFD: A monotonic task selection that tasks be sortd. Heap sorting X number of tasks would have a complexity of $O(XlogX)$. Then each task will be picked for allocation on each available core leading to the complexity of monotonic task selection as O(YX). Now, if the

number of regions is R and the cache associativity is A, then RFFD's algorithmic complexity is $O(AR^2)$. That is because the complexity of RFFD can be computed in the same ways as that of GFFD. Instead of allocation of tasks on a given number of cores, regions are allocated over a given number of cache associativity. As the associativity is a constant, RFFD's complexity can be stated as $O(R^2)$. Since RFFD allocation is nested within Mono selection, the overall complexity of Mono+RFFD is $O(YXR^2)$. With the number of cores being much less than the number of tasks and regions, one can approximate this complexity as $O(XR^2)$.

Dyn+CC: In order to determine the algorithmic complexity of this algorithm, we refer to Algorithm 25. The outermost loop at line 2 iterates over all tasks introducing a complexity of $O(X)$. The combined complexity with the linear search at line 3 is $O(X^2)$. The nested loop at line 4 iterates over each core with Chaitin's coloring based region allocation. The complexity of Chaitin-based region allocation can be determined the same way it was determined for Chaitin-based task allocation. Instead of tasks, here we are allocating regions to a given number of cache ways. The complexity of Chaitin-based task allocation is $O(X^2)$. Similarly, it is $O(R^2)$ for region allocation. Combining the complexity of the two outer loops leads to an algorithmic complexity of $O(YXR^2)$. This complexity can be approximated as $O(XR^2)$ as the number of cores is much less than the number of regions or tasks. Now we determine the complexity of the removal of the regions from the conflict graph. Line 11 iterates over a set of tasks in $UremTasksFromGraphs$. One might deduce that this may deliver a complexity of $O(X^2)$ when combined with the loop at line 2. However, this loop gets executed only X number of times. That is because each task is removed from the graph only once. If a task is allocated in the loop at line 4 then the loop at line 11 is not executed. Thus, the combined complexity of iterations at line 2 and 11 is $O(X)$. When combined with the complexity introduced by the nested loops at line 12 and 13, the overall complexity of region removal becomes $O(XR^2)$. This matches algorithmic complexity for Chaitin-based region allocation. Hence, the overall algorithmic complexity of Dyn+CC is $O(XR^2)$.

The algorithmic complexity of Dyn+CC and Mono+RFFD is the same and is dominated by the number of regions. CoFFD has a higher complexity than GFFD. The additional term is the number of cores, which is not the dominant component.

4.7 Task set Generation

Due to the unavailability of a full-blown real-time application for massive multicore architectures, we decided to utilize synthetic task sets in our experiments. This allows us to vary task set parameters like utilization of tasks, size of the locked regions, number of tasks, and number of regions per task, which in turn tests corner cases of our algorithms. For scenario B, as it allows larger numbers of locked regions, we also focus upon generation of denser conflict graphs.

We assume that static analysis tools, such as [70] deliver the $WCET_{locked}$, $WCET_{unlocked}$ and R_f, which is beyond the scope of this work.

Here, we explain the procedure of generating benchmarks for Scenario B, as it is less restrictive in its constraints. Next the steps used in automating the pseudo-random benchmark generation.

1. We use Task Graph For Free (TGFF) [31] to generate a conflict graph with a given number of regions (200) and with a randomly chosen number of conflicts per region, which is a randomly generated proportion of the total number of regions. (For high conflicts, the upper bound is 0.9 and lower bound varies from 0.1-0.5; for low conflicts, the lower bound is 0.1 and the upper bound varies from 0.2-0.5).

2. Randomly generate a number of accesses for each region within a given range of accesses (50-200).

3. Generate tasks and randomly pick a number of regions(2-8) for each until all the regions have been allocated to a task. The last task generated may have a lower number of regions than the lower bound of two regions. The lockable associativity of the L1 cache is assumed to be four.

4. The total number of references to locked regions were derived by aggregating the number of references incurred within the locked regions of the task. Since the programmer will be locking the regions in L1 (highest utilization benefit), we assume that these locked lines consume 90% of the total data loads. Out of the remaining 20%, we assume 18% are hits in the L2 cache and 2% are references to sensory data that goes off chip. We randomly choose 6-9 instructions per load. This lets us infer the number of instruction fetches that incur L1 cache hits (see Section 4.3). These assumptions allow us to derive a $WCET_{locked}$ for a task. The processor cores are assumed to feature an in-order 3-stage pipeline, with each instruction taking one cycle to execute, except branch instructions which incur a penalty of three cycles. The L1, L2 and Memory access latencies have been assumed to be 1, 10 and 100 cycles, respectively.

5. To derive the $WCET_{unlocked}$, we assume unlocked regions to hit in L2 cache. If two locked regions are accessed by two different paths, then the increase in WCET is due to just one region (the one that dominates the references), not both. Thus, we randomly select tasks to accommodate such behavior. This also results in varied increases in execution time between $WCET_{locked}$ and $WCET_{unlocked}$ across tasks.

6. Next, each task i is assigned a period to group them into the following categories of utilization: medium-high utilization ($0.55 > U_{locked_i} > 0.30$), and medium-low utilization

$(0.30 > U_{locked_i} > 0.1)$).

7. We assume that the tasks do not have any inter-task dependencies.

8. We assume task utilizations to be equal to a task's density. In other words, a task's deadline is equal to its period.

The generation of benchmarks for Scenario A differs from Scenario B in the following aspects:

1. First, a given number of tasks are generated. Then, each of them is given a number of randomly generated locked cache regions instead of generating a conflict graph. These locked cache regions are generated such that there is no intra-task conflict. In order to generate memory regions, we assume the cache architecture shown in Table 4.2. The table also displays the characteristics of the tasks and locked regions generated. The lockable associativity of cache, in Scenario A is lower than that of scenario B.

2. In Scenario A, we unlock a task while in Scenario B, we unlock a region. In order to observe the performance of task sets that have only high utilizations, we partition the utilization range into: high $0.55 > U_{locked} > 0.40$, medium $0.4 > U_{locked} > 0.25$, low $0.1 > U_{locked} > 0.25$.

Table 4.2: System Parameters

Parameter	**Value**
Cache Line Size	32B
L1 Cache Size/Associativity	8KB/2-way
Lockable associativity	1/2
locked regions per task	1 - 4
Sets locked by a task	8-114 out of 128
Size of locked regions	8-57 sets
Max. size of task sets	42
total tasks generated	126
Min. locked regions by a task	1

4.8 Evaluation

This section firstly presents the improvement of cache-aware schemes over cache-unaware schemes for task sets from Scenario A. Then it compares the performance of GFFD and CoFFD for task

sets of Scenario A. This is followed by evaluation with different combinations of task selection and task allocation (Mono+RFFD, Mono+CC, Dyn+RFFD, Dyn+CC) on task sets of Scenario B. Then we compare the performance of the best solutions from both the scenarios when applied upon task sets of Scenario A. This assesses the applicability of these algorithms.

4.8.1 Scenario A

We present our experimental results for a system that supports single locked cache ways. Such a scheme is also applicable when considering horizontal cache partitioning, where all the lockable ways in each set are dedicated to a task.

Table 4.3: Allocated Cores for Cache-aware & Cache-unaware Schemes

Number	**High Util.**		**Med. Util.**		**low Util.**	
of Tasks	**Unaware**	**Aware**	**Unaware**	**Aware**	**Unaware**	**Aware**
4	4	**3**	4	**2**	3	**2**
8	8	**5**	8	**4**	5	**3**
12	12	**8**	12	**5**	8	**4**
16	16	**10**	16	**8**	12	**6**
20	20	**13**	20	**11**	16	**8**
24	24	**15**	23	**15**	19	**10**
28	28	**19**	27	**19**	21	**11**
32	32	**20**	31	**21**	22	**12**
36	36	**21**	35	**22**	23	**15**
42	42	**25**	41	**24**	24	**17**

Cache-unaware vs. Cache-aware: First, we compare the cache-aware schemes (FFD, NFFD) against cache-unaware ones (GFFD, CoFFD). Table 4.3 shows the best allocations produced by schemes within the two categories, i.e., NFFD (cache-unaware) and CoFFD (cache-aware). On average, the number of cores used by cache-aware schemes is 40% less than that of contemporary allocation schemes applicable for distributed core mechanisms. We also observe that the contemporary FFD fails to allocate high utilization task sets. It performs worse than NFFD for low utilization task sets as shown earlier in Table 4.1.

Allocations while retaining locked state: Table 4.4 depicts the results of our algorithms when tasks are allocated in locked state, i.e., with an execution time of $WCET_{locked}$. The first column shows the number of tasks in the task set. The second and third columns show the number of cores allocated by GFFD and CoFFD, respectively, when a task set is composed of high utilization tasks only. The fourth and fifth columns represent the same for medium utilization tasks, and the sixth and seventh columns for lower utilization tasks. Lower core

Table 4.4: Allocated Cores for CoFFD & GFFD: All Tasks Locked

Number	**High Util.**		**Med. Util.**		**low Util.**	
of Tasks	**GFFD**	**CoFFD**	**GFFD**	**CoFFD**	**GFFD**	**CoFFD**
4	**3**	**3**	3	**2**	3	**2**
8	6	**5**	5	**4**	**4**	**4**
12	9	**8**	6	**5**	**5**	**5**
16	11	**10**	9	**8**	**8**	**8**
20	**13**	**13**	12	**11**	12	**11**
24	16	**15**	16	**15**	16	**15**
28	20	**19**	20	**19**	20	**19**
32	22	**20**	22	**21**	22	**21**
36	24	**21**	24	**22**	23	**22**
42	27	**25**	25	**24**	24	**23**

allocations are depicted in bold font. In all cases, CoFFD results in fewer cores allocated than GFFD, especially as the number of tasks increases. As more tasks are added to the system, the conflict graph becomes denser. CoFFD avoids conflicts strategically due to its coloring scheme while the greedy scheme results in a less conflict-conscious allocation.

Table 4.5: CoFFD vs GFFD: Selected Tasks Unlocked

Number of Tasks	GFFD	CoFFD	GFFD Util.	CoFFD Util.	Util. decreased by CoFFD
4	**2**	**2**	1.48	0.88	**40.54** %
8	**3**	**3**	2.05	2.027	**0.88** %
12	5	**4**	3.77	3.06	**18.83** %
16	7	**6**	5.07	4.13	**18.54** %
20	9	**8**	7.33	5.86	**19.64** %
24	11	**10**	8.6	7.04	**18.13** %
28	12	**11**	10.2	8.65	**15.19** %
32	14	**12**	11.57	9.7	**16.16** %
36	15	**15**	12.67	10.27	**18.94** %
42	17	**17**	14.04	11.87	**20.37** %

Allocations with all or none: This experiment allows allocation of tasks either with locking of all regions or while leaving all of them unlocked. After a locked allocation with $WCET_{locked}$ is attempted, algorithms can fall back to an unlocked allocation with $WCET_{unlocked}$ for a given task in case conflicts have prevented the allocation on a given core. Table 4.5 depicts the results with best results in bold face. The first column shows the number of tasks in the task set. The second and the third columns show the number of cores allocated by GFFD and

CoFFD, respectively. Sets with higher/medium utilization tasks result in similar allocations. This is because it is difficult for the higher utilization tasks to be allocated under the inflated execution budget of $WCET_{unlocked}$. However, tasks with lower utilizations can be allocate tasks with $WCET_{unlocked}$. The fourth and the fifth columns depict the system utilization delivered under the allocations of the algorithms. The last column shows the decrease in system utilization achieved by CoFFD over GFFD. The results indicate that CoFFD beats GFFD not only in terms of allocating fewer cores but also in improving system utilization by over 18% for task sets with large numbers of tasks. This is because GFFD inflates the execution budget of tasks that cannot be allocated to cores under locking. In addition, conflict analysis prior to allocation allows the algorithm to apply heuristics to reduce the number of tasks that remain unlocked. The results of CoFFD are due to combined heuristics for selecting spilled tasks. Heuristic 1 selects the task with the least $\frac{WCET_{unlocked}}{degreeofConflicts^2}$ value, which emphasizes the task's degree. This prevents the number of cores to be increased when non-conflict placements are still feasible. Algorithmically, CoFFD avoids spills of tasks onto the stack (see Algorithm 18). Heuristic 2 selects the task with the least $WCET_{unlocked}$ value. Of the two heuristics, CoFFD selects the one that results in the allocation of fewer cores. For example, most task sets in Table 4.5 resulted in the allocation of fewer cores under heuristic 1, but the last task set would have resulted in the allocation of 18 cores whereas heuristic 2 reduced this allocation to 17.

4.8.2 Scenario B

Next we present results of our algorithms that statically partition tasks for task sets of Scenario B. For scenario B, we generated a large set of benchmarks that can be varied with regard to utilization (medium-high, medium-low and mixed) and density of conflicts (high and low). A total of 1200 experiments were conducted. Each conflict ratio and utilization range, 10 benchmarks were created with different randomization seed values.

Mono+RFFD vs. Mono+CC: Table 4.6 shows the impact of of CC over RFFD on high conflict task sets. The first column shows the conflict ratio range of the high conflict benchmarks. Due to the large set of results, we only present the Best and Worst cases for CC over RFFD along with the average number of cores used per benchmark. The second column indicates conflict ratio ranges. The third, fourth and fifth columns show the results associated with benchmarks of high-medium, low-medium and mixed utilization ranges. Each of these columns have two sub-columns that depict the results for Mono+RFFD and Mono+CC for each utilization range and case. The results show that Mono+CC allocates the task sets to fewer or

Table 4.6: Mono+RFFD vs. Mono+CC for High Conflict: Number of cores allocated

Conflict Ratio Ratio	Performance Category	high-medium Util. Range		low-medium Util. Range		mixed Util. Range	
		Mono+RFFD	Mono+CC	Mono+RFFD	Mono+CC	Mono+RFFD	Mono+CC
0.1-0.9	Best	24	**22**	14	**12**	19	**17**
	Average	22.4	**21.3**	12.9	**12**	18.4	**17.1**
	Worst	**22**	**22**	**12**	**12**	**18**	**18**
0.2-0.9	Best	25	**22**	14	**12**	17	**15**
	Average	22.6	**21.2**	13	**12.1**	18.1	**17**
	Worst	**21**	**21**	**13**	**13**	19	**18**
0.3-0.9	Best	27	**23**	14	**12**	18	**15**
	Average	23.7	**22**	13.6	**12.5**	19	**17.6**
	Worst	21	**20**	13	**12**	**19**	**19**
0.4-0.9	Best	23	**21**	15	**13**	20	**17**
	Average	24.1	**22.8**	13.5	**12.6**	19	**17.5**
	Worst	**21**	**21**	**12**	**12**	**17**	**17**
0.5-0.9	Best	27	**23**	15	**13**	20	**17**
	Average	25	**23.1**	14	**13.2**	19.3	**18.1**
	Worst	**23**	**23**	**13**	**13**	**19**	**19**

Table 4.7: Mono+RFFD vs. Mono+CC for low Conflict: Number of cores allocated

Conflict Ratio Range	Performance Category	high-medium Util. Range		low-medium Util. Range		mixed Util. Range	
		Mono+RFFD	Mono+CC	Mono+RFFD	Mono+CC	Mono+RFFD	Mono+CC
0.1-0.2	Best	21	**20**	10	**9**	15	**14**
	Average	20.2	**20.1**	9.8	**9.7**	14.1	**13.9**
	Worst	same	same	same	same	**13**	**13**
0.1-0.3	Best	**20**	21	12	**11**	16	**15**
	Average	**20.3**	20.4	10.8	**10.6**	15.3	**15.1**
	Worst	same	same	**10**	11	**16**	17
0.1-0.4	Best	22	**21**	12	**10**	15	**14**
	Average	**20.1**	**20.1**	11.2	**10.3**	15.2	**14.9**
	Worst	**18**	19	**11**	**11**	**14**	15

a similar number of cores compared to Mono+RFFD for all the high-conflict benchmarks. This is primarily because the conflicts are high enough such that resolving them locally at the level of a core proves to be useful. Table 4.7 compares the performance of the CC over RFFD on low conflict task sets. The layout of the table is same as that of Table 4.6. However, we observe here that the worst cases force Mono+CC to map the tasks onto more cores than required by Mono+RFFD. This highlights the limits of a locally efficient coloring mechanism as it does not perform better at a global scale. This is because even though the coloring scheme does a better job at packing a given set of tasks within a core, sometimes failure to allocate some tasks within a core could pave the path for a better fit of subsequent tasks. This is the basis for our Dynamic algorithm that is sensitive to conflicts and provides a better global ordering during task selection phase.

Table 4.8: Average allocations performed by Scenario B algorithms: Number of cores allocated

Conflict Ratio Range	high-medium Util				low-medium Util				mixed Util			
	Mono+ RFFD	Mono+ CC	Dyn+ RFFD	Dyn+ CC	Mono+ RFFD	Mono+ CC	Dyn+ RFFD	Dyn+ CC	Mono+ RFFD	Mono+ CC	Dyn+ RFFD	Dyn+ CC
0.1-0.9	22.4	21.3	21.1	**20.3**	12.9	12	12.4	**11.6**	18.4	17.1	16.6	**16.1**
0.2-0.9	22.6	21.2	21	**20.4**	13	12.1	12.3	**11.7**	18.1	17	16.4	**15.9**
0.3-0.9	23.7	22	22.2	**21**	13.6	12.5	12.9	**12.2**	19	17.6	17.5	**16.7**
0.4-0.9	24.1	22.8	22.5	**21.3**	13.5	12.6	12.8	**12.3**	19	17.5	17.3	**16.8**
0.5-0.9	25	23.1	24.1	**22.1**	14	13.2	13.2	**12.4**	19.3	18.1	17.9	**17.1**
0.1-0.2	20.2	20.1	19.8	**19.7**	9.8	9.7	9.7	**9.4**	14.1	13.9	13.4	**13.4**
0.1-0.3	20.3	20.4	20.1	**20**	10.8	10.6	10.5	**10.3**	15.3	15.1	14.3	**14.1**
0.1-0.4	20.1	20.1	19.9	**19.7**	11.2	10.3	10.6	**10.2**	15.2	14.9	14.4	**14.2**

Table 4.9: Best and Average improvement by Dyn+CC over Mono+RFFD

Conflict Ratio Type	Improvement Case	high-medium Util	low-medium Util	mixed util
High	Best	22% 21 vs 27	17% 10 vs 12	19% 17 vs 21
	Average	10.49%	10.03%	11.81%
Low	Best	5.26% 18 vs 19	17% 10 vs 12	13% 13 vs 15
	Average	1.78%	5.72%	6.21%

Scenario B algorithms performance: Table 4.8 shows the average performance of the various algorithmic combinations of the task selection and task allocation algorithms for all conflict ratio ranges (low and high) on all utilization ranges (high-medium, low-medium and mixed). The first column shows the conflict ratios. The second, third and fourth columns show the results on benchmarks with high-medium, low-medium and mixed utilization ranges, respectively. Each of those columns have 4 sub-columns all task selection and task allocation combinations (in the following order: Mono+RFFD, Mono+CC, Dyn+RFFD, Dyn+CC). Due to the large number of results, we again resort to presenting the average number of cores allocated per benchmark for given conflict and utilization ranges. The highlighted numbers are again the best allocations. The highlighted results clearly show that Dyn+CC produces the best allocations compared to any other combination. Also, Dyn+RFFD consistently outperforms Mono+RFFD. Dyn+RFFD performs better than Mono+CC for the cases where the latter performed worse than Mono+RFFD. This shows the effectiveness of the Dynamic task selection mechanism. However, one should note that Dyn+RFFD does not always outperform Mono+CC, even though Dyn+CC is the best performing algorithm overall. Table 4.9 shows the Improvement achieved by Dyn+CC over the base case of Mono+RFFD. The first column

shows the two conflict ratio range types. The second column identifies the best and average case improvements in each of the cases. The third, fourth and fifth columns depict the improvement in terms of percentages, while the best case also shows the allocated core numbers (Dyn+CC vs. Mono+RFFD). Dyn+CC shows distinct high improvement percentage benefits for high conflict task sets (up to 6 cores). Among all the utilization ranges, the mixed utilization range produces the highest average improvement. This shows that the Dyn+CC algorithm not only works well with corner cases but it also performs best with systems that have a variety of utilizations and conflict densities.

Table 4.10: CoFFD vs Dyn+CC: task vs region unlocking

Number	**High Util.**		**Med. Util.**		**low Util.**	
of Tasks	**CoFFD**	**Dyn+CC**	**CoFFD**	**Dyn+CC**	**CoFFD**	**Dyn+CC**
16	**10**	**10**	**8**	**8**	**6**	**6**
20	**13**	14	11	**10**	**8**	**8**
24	**15**	17	15	**12**	**10**	**10**
28	**19**	20	19	**14**	**11**	**11**
32	**20**	22	21	**15**	**12**	13
36	**21**	25	22	**18**	**15**	**15**
42	**25**	27	24	**20**	**17**	**17**

CoFFD vs Dyn+CC: Dyn+CC has been the most useful combinations for task set of Scenario B. Thus, it becomes imperative to gauge its effectiveness relative to benchmarks of Scenario A and compare its performance against CoFFD. Table 4.10 depicts such results with higher conflict density task sets. CoFFD performs better that Dyn+CC for High Utilization tasks. That is because unlocking a region or a task does not allow multiple tasks to be scheduled together. Thus, the global ordering achieved by CoFFD is better. However, CoFFD is unable to unlock tasks of medium utilization either where the unlocking at regional granularity is feasible. Thus, Dyn+CC performs better. With low utilization task sets, CoFFD is able to unlock tasks and still allocate other tasks along with it. This makes both CoFFD and Dyn+CC perform well. Nonetheless, CoFFD benefits from a superior global ordering, which allows it to find a better allocation with our 32-task benchmark as shown in the table.

Chapter 5

Conclusion and Future Work

In this chapter, we present the conclusion drawn from the current state of our research work and the scope of the future work that may be pursued.

5.1 Conclusions

The work presented in this document first identifies task migration as a key contributor to unpredictability in determining WCET bounds of real-time tasks on multicore architectures. With larger L2 caches and increasing numbers of processing units, WCET bounds have the potential to become tighter in the future. Static timing analyzers can capitalize on large L2 caches in that, after the initial warm-up of the cache, execution of real-time tasks will become predictable. This work has shown that, in the wake of task migrations, dilation in execution time due to cache warm-up will become significant enough to occasionally prevent real-time tasks from meeting deadlines. It is thus imperative to develop real-time systems capable of tightly bounding — if not eliminating — the impact of task migration. Simulation results on a subset of WCET benchmarks experience a dilation in execution time ranging from of 6% to 56.6% for tasks whose algorithmic complexity does not exceed $O(n)$. Tasks with higher complexity show a significant dilation for small data set sizes.

We consolidate the idea of proactive cache migration as a means to diminish the dilation introduced by the target warm-up overhead using a software technique called PTM. PTM launches a low priority prefetch task at the target core to prefetch the cache lines that belong to the address ranges specified by the programmer. The software approach shows that it can prevent the dilation in execution time but may result in high and potentially unbounded migration delay requirements. Hence, we propose two schemes of push-assisted cache-to-cache migration in multicores. (1) WCM, a hardware scheme replicating the cache context of the task onto the target L2 cache, reduces dilation in execution time to less than a percent for the

majority of simulated tasks, except for those with the smallest data set sizes or an algorithmic complexity lower than $O(n)$. WCM eliminates any execution overheads of PTM has but still maintains a high overhead due to a complete scan of the cache. (2) RCM uses the address range specification capabilities of PTM with hardware-based pushing of cache lines of WCM. This allows RCM to achieve performance benefits similar to WCM while making the cache migration overhead proportional to the cache footprint of the task. (3) This overhead is further reduced by BCM, which allows multiple push requests to be processed simultaneously. Through these contributions, migration overhead for most of the tasks is reduced to less than 20% of the task's execution time. This demonstrates that cache migration is a feasible solution for preserving execution times after task migration close to those tight WCET bounds otherwise only valid in the absence of migration.

We further enhance the MESI coherence protocol to significantly reduce or even eliminate the number of write misses due to task migration. This eliminates extra bandwidth requirements due to cache migration except for residuals of tasks with large data sets.

The work further promotes multicores in hard real-time systems under cache locking. In hard real-time systems, cache locking increases the predictability of worst-case execution time potentially resulting in higher utilization. On multicore platforms, optimal scheduling assumes task migration as a fundamental premise. This work discusses support required under cache locking for proactive lock and cache content migration. We develop a wide range of cache migration models that provide deterministic migration delay.

We exploit pipelining for Regional Cache Migration (RCM) through Controlled Cache Migration Pipelining (CCMP) and Streamed Cache Migration Pipelining (SCMP) that reduce the migration cost over RCM by 48% and 56%, respectively. We expose system-wide parallelism for cache migration through a novel hardware synchronization mechanism. This allows multiple cache migrations to overlap and maximize system bus utilization. We also present a hardware mechanism called Set-Scan Cache Migration (SSCM) to migrate sparse cache locks that cannot be specified by large memory regions within Region Registers. Slotted-SSCM, an extension to SSCM, allows cache migrations to progress in parallel with RCM-based cache migrations. Slotted-SSCM also lends itself to pipelining that leads to Slotted-SSCM Pipelining. Slotted-SSCM Pipelining delivers a reduction in migration cost over SSCM by 46.7%. Individually, Slotted-SSCM may seem to have high overhead with large caches due to extra set reads. This cost can be mitigated if Region Registers (otherwise recommended for RCM) are used to specify a group of contiguous sets that contains locked lines. This is based on the observation that locks that seem sparse in large memory space may fit within a small set of cache sets. This hybrid design of RCM and Slotted-SSCM has the potential to significantly reduce the overhead of extra cache set reads.

The cache migration schemes mentioned in this work provide the scheduler with opportuni-

ties to deliver deterministic and efficient cache migrations. Single cache migrations should make use of pipelined mechanisms. SCMP and Slotted-SSCM Pipelining deliver the best results. In case of multiple cache migrations, the scheduler can choose between parallel migration and pipelined migration based on the knowledge of individual migration costs.

The work presented in this document finally studies static partitioning of real-time tasks with locked caches on distributed cache systems. Contemporary static scheduling schemes may not use locked caches. However, this renders certain high utilization tasks unschedulable as their unlocked WCET is prohibitively high. A simplistic solution would be to allowing locking of such tasks and placing locked tasks onto different cores. We call this Naive locked FFD (NFFD) as it locks certain tasks with high utilizations and is unaware of caches.

This work proposes two cache-aware algorithms for Scenario A type task sets. These algorithms allocate tasks in a multicore environment where tasks are allowed to lock cache lines in a specified subset of cache ways in each core's private L1 cache. The first algorithm, GFFD, is an enhanced version of the First Fit Decreasing (FFD) algorithm. The second, CoFFD, is based on a graph coloring method. CoFFD reduces the number of core requirements from 25% to 60% compared to NFFD with an average reduction of 40%. CoFFD consistently performs better than GFFD as it lowers both the number of cores and the overall system utilization.

Further, we present algorithms for task sets of Scenario B. Tasks within these task sets can have intra-task cache conflicts and allocate these conflicting locked regions into different cache ways. These algorithms resolve conflicts among locked regions by locking and unlocking regions instead of locking or unlocking entire tasks as was the case in Scenario A. Here, task partitioning is split into task selection and task allocation phases. We use two task allocation mechanisms, namely (a) Regional FFD and (b) Chaitin's coloring. We propose two task selection algorithms, namely (a) Monotone and (b) Dynamic. The combination of Mono+CC outperforms Mono+RFFD for highly conflicted task sets. This shows the effectiveness of using the coloring mechanism at individual cores. In contrast, Mono+CC does not consistently perform best for low contention task sets. This necessitates a global ordering scheme like Dyn, which complements core-level coloring. Our results show that Dyn+CC consistently performs better than Mono+RFFD. For high contention task sets, Dyn+CC achieves up to a 22% reduction in the number of cores allocated, i.e., it allocates 21 cores as opposed to 27 for Mono+RFFD. Even for low contention task sets, Dyn+CC is able to achieve a reduction of up to 17%. Since Dyn+CC deals with a more generic problem set, it is also applicable to task sets of Scenario A. While comparing CoFFD against Dyn+CC, we observe that CoFFD performs better than Dyn+CC for high utilization tasks. That is because unlocking a region or a task does not allow multiple tasks to be scheduled together. Thus, the global ordering achieved by CoFFD is superior. However, CoFFD is unable to unlock medium utilization tasks. In contrast, unlocking at regional granularity is feasible and thus Dyn+CC performs better. These observations suggest

that CoFFD and Dyn+CC perform far better than contemporary FFD-based task partitioning on distributed core CMPs. Overall, this work is unique in considering the challenges of future multicore architectures for real-time systems. This work also provides key insights into task partitioning with locked caches for architectures with private caches and presents the ways in which this line of research may be extended.

Overall, this dissertation promotes deployment of locked caches and hardware support for cache migration on scalable multi-processors that can enable more predictable and efficient multiprocessor real-time scheduling.

5.2 Future Work

There are several additional directions in which this work can be extended.

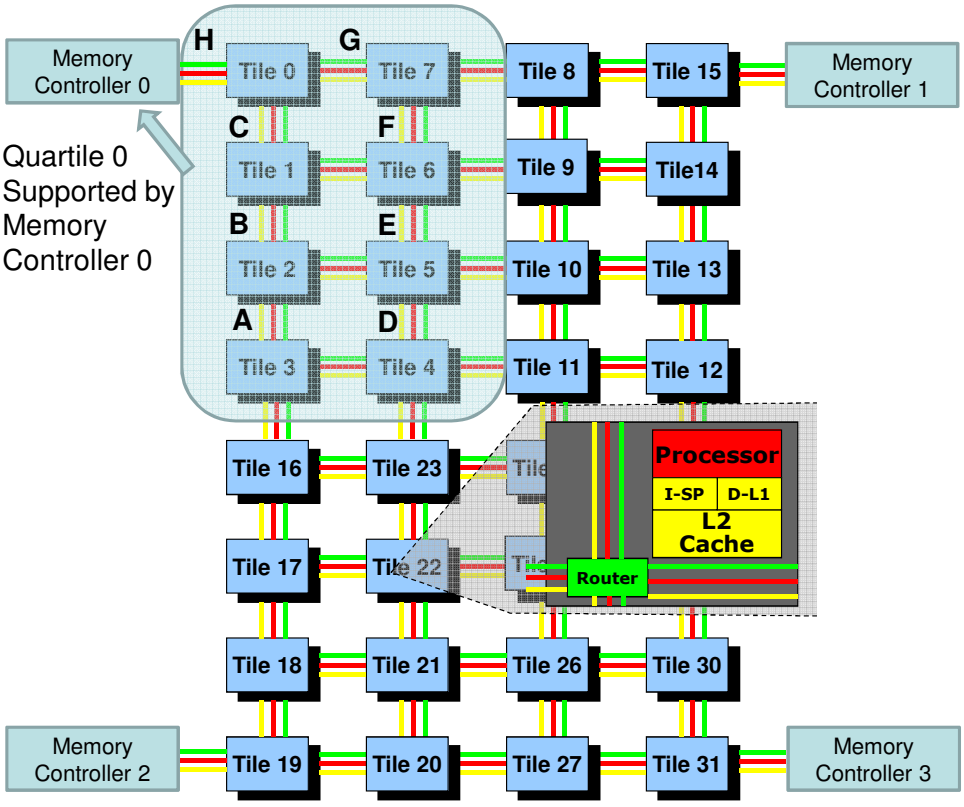
5.2.1 Holistic Task Partitioning on Scalable Multicore Real-Time Systems

The algorithms of Chapter 4 are applicable to any multicore system that has two levels in the memory hierarchy. We resolved conflicts among locked regions in private L1 cache while assuming that the private L2 cache has sufficient associativity and size to hold all the locked regions of all the tasks mapped to its corresponding core. As our solution only supports two levels, we consider partitioning the tasks onto cores based on conflict resolution at the L2 cache level and references to unlocked regions to be issued to the memory.

Figure 5.1 shows a 32 core tile-based architecture. There are four on-chip memory controllers that interface with four separate memory chips. Consider the mesh interconnect in the highlighted part of Figure 5.1. This highlighted portion depicts a quartile. Memory traffic from/to each quartile is statically routed to/from a designated memory controller along the mesh and arbitrated using a time-division-multiplexed (TDM) approach, as described below.

We assume a dimension-order Y-X routing, i.e., a request from a core to a memory controller first traverses interconnect vertically until it cannot go any further, and then travels horizontally. For example, the routing paths of requests from tile 3 and 4 are $A \rightarrow B \rightarrow C \rightarrow H$ and $D \rightarrow E \rightarrow F \rightarrow G \rightarrow H$. The traffic from the from the memory controller to the cores uses a separate channel and uses X-Y routing. The bandwidth allocated to memory traffic from a given core along a given route is proportional to the number of hops from the core to the target of that link. For example, the bandwidth along link C is divided among tiles 3, 2 and 1 in the ratio 3:2:1 since traffic from tile 3 crosses three hops to get to the target of link C, tile 2 crosses two hops and tile 1 crosses one hop. If we assume that each hop takes one cycle, the NoC latencies for tiles 1, 2 and 3 across link C are 2, 3 and 6 cycles, respectively. Similarly, NoC latencies for tiles 2 and 3 across link B are 2 and 3 cycles and that for tile 3 across link A is 1 cycle. Hence, the total NoC latency for traffic from tile 1, 2 and 3 to reach tile 0 is 3, 5

Figure 5.1: Memory Traffic Routing of a Quartile



and 6 cycles, respectively. Similarly, traffic from tiles 4, 5, 6 and 7 takes 10, 9, 7 and 4 cycles, respectively, to reach tile 0. Since, the router at tile 0 receives traffic from all the 8 cores in the quartile, each request may have to wait for a maximum of 8 cycles before being received by the memory controller. For safety, we assume that the NoC latency of every memory request is the maximum of all these latencies, namely 18 cycles. This is the number of cycles needed in each direction by the traffic involving tile 4 and the memory controller. In order to study the impact of memory latency on our algorithms, we assume two cases:

1. Fully-pipelined memory: This is the ideal scenario where all memory references take the minimum number of cycles needed to access memory. As per TilePro64 specification, each memory access takes 70 cycles [36]. This delivers the following latency for an access to an unlocked cache line: $70(memory) + 2 \times 18\ (controller\ to fro\ tile) + 2(bypassed\ processor\ to fro\ router) = 108 cycles$

2. TDM-based memory: This is the worst case scenario which does not parallelize or pipeline memory requests. For the architecture depicted in Figure 5.1, it adds another 490 cycles to

the access latency derived for fully-pipelined memory. That is because a memory access from one tile has to multiplex with requests from for seven other tiles (7×70). This delivers an access latency of 598 cycles to an unlocked cache line.

To evaluate our mechanism on an L2 cache, we use our worst (Mono+RFFD) and best(Dyn+CC) algorithms described in Chapter 4. we conducted some experiments on a few synthetic benchmarks of Scenario B. We use the TGFF-based procedure to generate benchmarks, as described in the previous chapter. We generate 25 benchmarks (medium-low utilization and high conflict ratios) for both the fully-pipelined and the TDM-based latency cases. As explained in Chapter 4, medium-low utilization has ($0.30 > U_{locked_i} > 0.1$)). Also, recollect that the conflict ratio is the ratio between the number of conflicts a region has versus the total number of regions. Since the conflicts are randomly generated using TGFF, high conflict ratios have an upper bound of 0.9 and the lower bound varies from 0.1-0.5 [1]. We assume that the instructions of all tasks fit within the L1 cache. For each benchmark, TGFF generates a conflict graph of 200 regions that are distributed among tasks. The number of regions per task is between 4 and 8.

Table 5.1: Average core allocation: Sensitivity to Memory Latency

Conflict Ratio Range	Fully-Pipelined		TDM-based	
	Mono+ RFFD	Dyn+ CC	Mono+ RFFD	Dyn+ CC
0.1-0.9	17.2	**14.6**	30.2	**27.8**
0.2-0.9	17.6	**14.8**	30.8	**28**
0.3-0.9	17.8	**13.8**	31.2	**28**
0.4-0.9	18.2	**15.2**	31.2	**27.6**
0.5-0.9	18.4	**14.8**	31.8	**27.6**

Table 5.1 shows the preliminary results obtained on these synthetic benchmarks with a lockable associativity of 8 for an L2 cache. The first column shows the conflict ratios. The second and third columns correspond to fully-pipelined and TDM-based latencies, respectively. Each of those columns have 2 sub-columns that show the average allocations obtained by the Mono+RFFD and the Dyn+CC algorithms, respectively. These averages are obtained across 5 benchmarks, each corresponding to a group of conflict ratio ranges and latencies. The results show that Dyn+CC delivers better core requirements than Mono+RFFD. However, due to the high latency in TDM-based memory access, it is almost impossible to unlock any of the regions.

[1] To read more on utilization ranges and conflict ratios, please refer page 93

Thus, the improvements obtained by Dyn+CC are solely due to the coloring localized within each core.

The aforementioned system assumptions and the results obtained point us to research opportunities that will complement task partitioning studied in this dissertation

1. As shown in Figure 5.1, the memory controllers are dedicated to serve a particular quartile. This would require memory partitioning that allows all the tasks mapped to a particular quartile to access only one memory controller. Strategies to partition memory need to be accommodated along with cache-based task partitioning to devise a holistic approach.

2. The results obtained also show that by reducing the upper bound of memory latency, a significant improvement can be obtained in allocations. This is another area that would require certain hardware schemes to either parallelize the memory accesses or pipeline them. These also need to be studied in combination with the memory partitioning strategies.

3. A more immediate extension to our work would be to support multi-level caches. This would require integrating conflict graphs for multiple levels of a memory hierarchy.

5.2.2 Thermal Analysis of Multicore Real-Time Systems

This topic is motivated by exhibiting the use of task migration for improving the reliability of multicore real-time systems. Heavy and frequent access of architectural resources by certain real-time tasks may cause temperature of those resources to rise to high levels. Partitioned scheduling, which is heavily being used in real-time systems, may be susceptible to these effects because of a lack of task migration. We intend to analyze how task migration coupled with cache migration can help in bringing down the temperatures while preventing the dilation in the execution time of the task.

REFERENCES

[1] Hypertransport technology. http://www.amd.com/us/products/technologies/hypertransport-technology.

[2] Intel pentium d architecture. http://en.wikipedia.org/wiki/Pentium-D.

[3] Intel quickpath interconnect. http://www.intel.com/intelpress/sum_qpi.htm.

[4] Tegra 3 architecture. http://www.anandtech.com/show/5072/nvidias-tegra-3-launched-architecture-revealed.

[5] Tera-scale research prototype: Connecting 80 simple sores on a single test chip. ftp://download.intel.com/research/platform/terascale/terascaleresearchprototypebackgrounder.pdf.

[6] Tilera processor family. http://www.tilera.com/products/processors.php.

[7] Wcet project benchmarks, 2007. http://www.mrtc.mdh.se/projects/-wcetbenchmarks.html.

[8] Andrea Acquaviva, Andrea Alimonda, Salvatore Carta, and Michele Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP J. Embedded Syst.*, 2008(2):1–15, 2008.

[9] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 179–190, April 2006.

[10] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Euromicro Conference on Real-Time Systems*, pages 35–43, June 2000.

[11] J. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *Euromicro Conference on Real-Time Systems*, pages 76–85, June 2001.

[12] James H. Anderson, Vasile Bud, and UmaMaheswari C. Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ECRTS '05, pages 199–208, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Patricia Balbastre, Ismael Ripoll, and Alfons Crespo. Minimum deadline calculation for periodic real-time tasks in dynamic priority systems. *IEEE Trans. Comput.*, 57(1):96–109, January 2008.

[14] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *IEEE Real-Time Systems Symposium*, pages 119–128, 2007.

[15] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[16] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. Asr: Adaptive selective replication for cmp caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 443–454, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Proceedings of the conference on Design, automation and test in Europe*, pages 15–20, 2006.

[18] A. Burchard, J. Liebeherr, Y. Oh, and S.H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. on Computers*, 44(12):1429–1442, 1995.

[19] J. Calandrino and J. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Euromicro Conference on Real-Time Systems*, pages 209–308, July 2008.

[20] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 98–105, 1982.

[21] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[22] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *International Symposium on High Performance Computer Architecture*, pages 340–351, 2005.

[23] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2000.

[24] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *International Symposium on Computer Architecture*, pages 264–276, 2006.

[25] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, pages 6:1–6:10, New York, NY, USA, 2010. ACM.

[26] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. In *International Symposium on Computer Architecture*, pages 357–368, 2005.

[27] D. Choffnes, M. Astley, and M. J. Ward. Migration policies for multi-core fair-share scheduling. *ACM SIGOPS Operating Systems Review*, 42:92–93, 2008.

[28] Gille Damien. Study of different cache line replacement algorithms in embedded systems. Master's thesis, KTH Royal Institute of Technology, March 2007.

[29] Michael L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

[30] S.K. Dhall and C.L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[31] Robert P. Dick, David L. Rhodes, and Wayne Wolf. Tgff: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, CODES/CASHE '98, pages 97–101, Washington, DC, USA, 1998. IEEE Computer Society.

[32] F. Dorin, P. Meumeu Yomsi, J. Goossens, and P. Richard. Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms. In *Real-Time and Network Systems*, pages 207–216, November 2010.

[33] Noel Eisley, Li-Shiuan Peh, and Li Shang. In-network cache coherence. In *International Symposium on Microarchitecture*, pages 321–332, 2006.

[34] Noel Eisley, Li-Shiuan Peh, and Li Shang. Leveraging on-chip networks for data cache migration in chip multiprocessors. In *International conference on Parallel architectures and compilation techniques*, pages 197–207, 2008.

[35] A. Fedorova, M. Seltzer, and M.D. Smith. Cache-fair thread scheduling for multicore processors. Technical Report TR-17-06, Harvard University, October 2006.

[36] Mondello Filippo. Architectures for multimedia systems. http://home.dei.polimi.it/silvano/FilePDF/ARC-MULTIMEDIA/Presentation_Tilera_-Tile64.pdf.

[37] Shelby Funk and Sanjoy Baruah. Restricted edf migration on uniform multiprocessors. *Technique Et Science Informatiques 24 (8)*, pages 917–938, 2005.

[38] 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.

[39] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *International Symposium on Microarchitecture*, pages 343–354, 2005.

[40] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program analysis. *Journal of Instruction Level Parallel*, Sep 2005.

[41] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 68–77, Washington, DC, USA, 2009. IEEE Computer Society.

[42] Damien Hardy and Isabelle Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of Real-Time Systems Symposium*, pages 456–466, 2008.

[43] John L. Hennessy and David A. Patterson. *Computer architecture - a quantitative approach, 3rd Edition*. Morgan Kaufmann, 2003.

[44] Dong ik Oh and T.P. Baker. Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-Time Systems*, 15:183–192, 1998.

[45] Ravi Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *Proceedings of international conference on Supercomputing*, pages 257–266, 2004.

[46] N. Jerger, M. Lipasti, and L. Peh. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. In *International Symposium on Microarchitecture*, pages 35–46, November 2008.

[47] D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Trans. Softw. Eng.*, 19(9):920–934, September 1993.

[48] Seon Wook Kim, Michael Voss, Bob Kuhn, Hans-Christian Hoppe, and Wolfgang Nagel. Vgv: Supporting performance analysis of object-oriented parallel applications. In *Proc. of IPDPS'2002 (HIPS'2002): Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 108–115, April 2002.

[49] T. Kim, H. Shin, and N. Chang. Deadline assignment to reduce output jitter of real-time tasks. In *Proc. 16th IFAC Workshop Distributed Computer Control Systems*, 2000.

[50] S. Lauzac, R. Melhem, and D. Mosse. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *Euromicro Workshop on Real-Time Systems*, pages 188–195, 1998.

[51] T. Li, P. Brett, B. Hohlt, R. Knauerhase, S.D. McElderry, and S. Hahn. Operating system support for shared-isa asymmetric multi-core architectures. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 19–26, June 2008.

[52] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *In ACM/IEEE conference on Supercomputing*, pages 1–11, November 2007.

[53] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *MICRO*, pages 226–237, 1996.

[54] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[55] Tiantian Liu, Yingchao Zhao, Minming Li, and Chun Jason Xue. Task assignment with cache partitioning and locking for wcet minimization on mpsoc. In *Proceedings of the 2010 39th International Conference on Parallel Processing*, ICPP '10, pages 573–582, Washington, DC, USA, 2010. IEEE Computer Society.

[56] Michael R. Marty, Jesse D. Bingham, Mark D. Hill, Alan J. Hu, Milo M. K. Martin, and David A. Wood. Improving multiple-cmp systems using token coherence. In *International Symposium on High Performance Computer Architecture*, pages 328–339, 2005.

[57] Michael R. Marty and Mark D. Hill. Virtual hierarchies to support server consolidation. In *International Symposium on Computer Architecture*, pages 46–56, 2007.

[58] Wen mei W. Hwu and Yale N. Patt. Checkpoint repair for high-performance out-of-order execution machines. *IEEE Trans. Computers*, 36(12):1496–1514, 1987.

[59] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *IEEE Real-Time Systems Symposium*, pages 294–303, December 1999.

[60] F. Mueller. Timing predictions for multi-level caches. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36, June 1997.

[61] Jin Ouyang and Yuan Xie. Loft: A high performance network-on-chip providing quality-of-service support. *Microarchitecture, IEEE/ACM International Symposium on*, 0:409–420, 2010.

[62] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 57–68, New York, NY, USA, 2009. ACM.

[63] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Robert I. Davis, and Mateo Valero. Ia3: An interference aware allocation algorithm for multicore hard real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 280–290, 2011.

[64] Rodolfo Pellizzoni. *Predictable and monitored execution for cots-based real-time embedded systems.* PhD thesis, Champaign, IL, USA, 2010. AAI3452102.

[65] Isabelle Puaut. Wcet-centric software-controlled instruction caches for hard real-time systems. In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 217–226, Washington, DC, USA, 2006. IEEE Computer Society.

[66] Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *In IEEE Real-Time Systems Symposium*, pages 114–123, 2002.

[67] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[68] H. Ramaprasad. *Analytical Bounding Data Cache Behavior for Real-Time Systems.* PhD thesis, Dept. of CS, North Carolina State University, July 2008.

[69] H. Ramaprasad and F. Mueller. Bounding worst-case response time for tasks with non-preemptive regions. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 58–67, April 2008.

[70] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemptions. *Transactions on Embedded Computing Systems*, March 2008 (accepted).

[71] J. Rawat. Static analysis of cache analysis for real-time programming. Master's thesis, Iowa State University, 1995.

[72] J. Renau, B. Fragela, J. Tuck, W. Liu, L. Ceze, S. Sarangi, P. Sack, and and P. Montesinos K. Strauss. Sesc simulator. http://sesc.sourceforge.net, Jan. 2005.

[73] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 2008.

[74] Abhik Sarkar, Frank Mueller, and Harini Ramaprasad. Predictable task migration for locked caches in multi-core systems. In *LCTES*, pages 131–140, 2011.

[75] Abhik Sarkar, Frank Mueller, and Harini Ramaprasad. Static task partitioning for locked caches in multi-core real-time systems. In *submitted to CASES*, 2012.

[76] Abhik Sarkar, Frank Mueller, and Harini Ramaprasad. Static task partitioning for locked caches in multi-core real-time systems. *to be submitted to TECS*, 2012.

[77] Abhik Sarkar, Frank Mueller, Harini Ramaprasad, and Sibin Mohan. Push-assisted migration of real-time tasks in multi-core processors. In *LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 80–89, New York, NY, USA, 2009. ACM.

[78] Y. Sazeides and J.E. Smith. The predictability of data values. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 248 –258, dec 1997.

[79] Rathijit Sen and Y. N. Srikant. Wcet estimation for executables in the presence of data caches. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 203–212, New York, NY, USA, 2007. ACM.

[80] M. Shekhar, A. Sarkar, H. Ramaprasad, and F. Mueller. Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems. In *Euromicro Conference on Real-Time Systems*, 2012.

[81] Abhishek Singh. *Co-scheduling Real-time Tasks and Non Real-time Tasks Using Empirical Probability Distribution ofExecution Time Requirements*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 2008.

[82] E. Smith and Arden Hills Minnesota. A study of branch prediction strategies. In *In Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, 1981.

[83] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *ISCA*, pages 36–44, 1985.

[84] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *ACM Symposium on Theory of Computing*, pages 189–198, May 2002.

[85] Jan Staschulat and Rolf Ernst. Multiple process execution in cache related preemption delay analysis. In *In international conference on Embedded software*, pages 278–286, 2004.

[86] Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Euromicro Conference on Real-Time Systems*, pages 41–48, 2005.

[87] Karin Strauss, Xiaowei Shen, and Josep Torrellas. Uncorq: Unconstrained snoop request delivery in embedded-ring multiprocessors. In *International Symposium on Microarchitecture*, pages 327–342, 2007.

[88] Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*, pages 300–303, New York, NY, USA, 2008. ACM.

[89] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *REAL-TIME SYSTEMS*, 18:157–179, 1999.

[90] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196, 2002.

[91] Rafael Ubal, Julio Sahuquillo, Salvador Petit, H. Hassan, and Pedro Lopez. Leakage Current Reduction in Data Caches on Embedded Systems. In *Intelligent Pervasive Computing, 2007. IPC. The 2007 International Conference on*, pages 45 –50, oct. 2007.

[92] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, December 2003.

[93] Xavier Vera, Bjrn Lisper, and Jingling Xue. Data caches in multitasking hard real-time systems. In *In IEEE Real-Time Systems Symposium*, pages 154–165, 2003.

[94] www.arm.com. *ARM Processor Specifications.*

[95] www.openmp.org. *Official OpenMP Specification*, May 2005.

[96] J. Yan and W. Zhang. Time-predictable l2 caches for real-time multi-core processors. In *Work in Progress session of IEEE Real-Time Systems Symposium*, December 2007.

[97] J. Yan and W. Zhang. Wcet analysis of multi-core processors. In *Work in Progress session of IEEE Real-Time Systems Symposium*, December 2007.

[98] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 80–89, April 2008.

[99] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, MICRO 24, pages 51–61, New York, NY, USA, 1991. ACM.

[100] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266. ACM, 1993.

[101] Michael Zhang and Krste Asanovic. Victim migration: Dynamically adapting between private and shared cmp caches. TR 2005-064, MIT CSAIL, 2005.