Push-Assisted Migration of Real-Time Tasks in Multi-Core Processors

Abhik Sarkar[1], Frank Mueller[1], Harini Ramaprasad[2], Sibin Mohan[3]

[1]North Carolina State University,[2]Southern Illinois University,[3]University of Illinois at Urbana Champaign

[1]asarkar@ncsu.edu,[1]mueller@cs.ncsu.edu,[2]harinir@siu.edu,[3]sibin@cs.uiuc.edu

Abstract

Multicores are becoming ubiquitous, not only in general-purpose but also embedded computing. This trend is a reflexion 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 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 while (b) increasing traffic on the network-on-chip (NoC) interconnect.

This paper puts forth a fundamentally new approach to increase the timing predictability of multicore architectures aimed at task migration in embedded environments. 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. Our scheme shows dramatically increased predictability in the presence of cross-core migration.

Experimental results for schedules demonstrate that our scheme enables real-time tasks to meet their deadlines in the presence of task migration. Our results illustrate that increases in execution time due to migration is reduced by our scheme to levels that may prevent deadline misses of real-time tasks that would otherwise occur. Our mechanism imposes an overhead at a fraction of the task's execution time, yet this overhead can be steered to fill idle slots in the schedule, *i.e.*, it does not contribute to the execution time of the migrated task. Overall, our novel migration scheme provides a unique mechanism capable of significantly increasing timing predictability in the wake of task migration.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design—real-time systems and embedded systems; D.4.1 [*Operating Systems*]: Process Management—scheduling; B.4.2 [*Memory Structures*]: Design Styles—cache memories

General Terms Design, Experimentation.

LCTES'09, June 19–20, 2009, Dublin, Ireland.

Keywords Real-Time Systems, Multi-Core Architectures, Timing Analysis, Task Migration.

1. Introduction

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 constrains, which has led designers to invest into chip multiprocessors (CMPs), a.k.a. multicores, to ensure that performance increases at past rates [1]. This trend already extends to embedded designs with heterogeneous multicores (*e.g.*, for cell phones) and also homogeneous multicores at a larger scale [2]. Multicores will become ubiquitous over the next years. Alongside, contemporary embedded systems are demanding a steadily increasing amount of processing power. This is being met by deploying multicore architectures because the performance potential of multicores make it feasible. However, on such platforms, prediction of timing behavior to ensure that deadlines of embedded tasks can be met is becoming increasingly difficult. While real-time multicore scheduling approaches help to assure deadlines at the theoretical level, 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 while (b) increasing traffic on the network-on-chip (NoC) interconnect.

This paper puts forth a fundamentally new approach to increase the timing predictability of multicore architectures aimed at task migration in embedded environments. A set of tasks, periodic or sporadic in invocation, interacts with the embedded environment through sensors and actuators such that tasks have to complete execution by predefined deadlines. Schedulability analysis provides the theoretical foundation to determine if such timing constraints can be met [32]. To meet deadlines, tasks should be predictable in their timing behavior so that timing analysis may derive tight upper bounds on their worst-case execution time (WCET) [8, 37, 46, 47]. In particular, so-called hard real-time embedded systems have *timing constraints that must be met or the system may malfunction.*

Scheduling Challenges and Task Migration: Central to the theme of multicore systems is the scheduling of tasks onto cores. Two approaches may be used for this purpose, namely partitioned and global scheduling. Under partitioned scheduling [19, 12], tasks are assigned to cores and are not allowed to migrate. While this approach imposes no migration overhead, it has several limitations. Optimal assignment of tasks to partitions is an NP-Hard problem, thus making any partitioned scheduling scheme inherently suboptimal. Furthermore, in situations of dynamic task admittance, the entire system has to be re-partitioned, thus making partitioned scheduling highly inflexible.

To address these limitations, 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 [19, 28]. A whole class of fair scheduling algorithms have recently been developed and have been proved to be optimal for multicore scheduling [10, 35, 6, 7, 41, 9]. However, using global/fair scheduling algorithms in the context of real-time systems introduces the challenge of having to provide temporal guarantees for tasks. This has motivated us to study the impact of migration overheads on the predictability of tasks and also to present viable hardware/software solutions for the same.

Timing Predictability and Micro-Architectural Challenges: Multicores have been a hot 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 [1, 2] due to the frequency wall. Past research has focused on improving parallelization strategies [15, 48, 40, 45, 23] to increase average performance and to provide scalability on the memory path [14, 34, 26, 20, 44, 33], including capacity-oriented schemes that scavenge unused neighboring cache lines [21, 16, 17, 52]. 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. Real-time systems largely consist of multiple tasks, many of them 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 short processing actions of often control-theoretic nature and then engage in actuator actions. Jobs of such real-time task systems tend to be frequent and short in execution that has to complete by a job's deadline. Such short 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. In this context, contemporary multicore research focuses on benchmarks of much longer executions for an application. This diminishes the impact of timing dilation that are easily compensated by savings in subsequent executions and absence of a requirement to meet deadlines. The dilation of interest in this paper is due to migrating a task or an application between processor cores. Task migrations in contemporary architecture are followed by cache warm-ups on the migration target. Cache warm-ups are usually ignored or have insignificant impact while conducting high-performance design studies. 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 failure of the system.

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, static analysis can provide tight WCETs as processor behavior becomes quite predictable. However, as execution times are becoming increasingly tighter in deployments with high system utilization, 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. Our experimental results on a 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 56.6%.

We propose 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 [36, 51, 24]. 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. We propose a pure hardware scheme, called the whole cache migration (WCM), that restores the whole cache context of the task on to the target core. This reduces the time dilation that the task is subjected on the target core as a result of its migration. However, it 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 rest of this paper is organized as follows. Section 2 discusses the contributions of our work. Section 3 discusses the related work. Section 4 analyzes the problem. Section 5 describes the solution at the level of the real-time system scheduler. Section 6 describes in detail the design of the WCM and RCM schemes. Section 7 describes the architecture specifications of the simulation framework. Section 8 presents and analyzes the results obtained by implemented schemes. Finally, we draw conclusions in Section 9.

2. Contributions

This research work makes the following contributions toward the study of WCET for real-time tasks on multicore architectures:

1. This work establishes task migration as an important source of unpredictability in deriving tight WCET bounds of real-time tasks on emerging CMP architectures with large enough L2 caches. This opens a new area of investigation into real-time systems that focuses upon improving the predictability of tasks on CMPs in the wake of task migrations, which is central to multicore schedulability as discussed in Section 1.

2. It proposes a novel push-assisted micro-architectural scheme to migrate cache lines from one core to another. It interacts with the cache controller and its snoop capability to accomplish the cache migration.

3. It puts forth enhancements to the standard MESI coherence protocol so that cache lines in modified/exclusive states are migrated explicitly and in a targeted manner with their native state or an optional transitional state.

4. The work extends the ISA to enable scheduler-triggered cache migration. In contemporary multicore architecture, there currently exists no mechanism that notifies the core about the migration of a task.

5. It develops ISA extensions to support critical address range specification by the developer pertaining to the memory layout of a migrated task. This helps the proposed cache migrating hardware to limit its search specifically to targeted address ranges and thus reduces cache migration overhead considerably.

6. The study highlights novel research opportunities associated with task migration on multicores. These research opportunities are discussed in detail in Section 5.

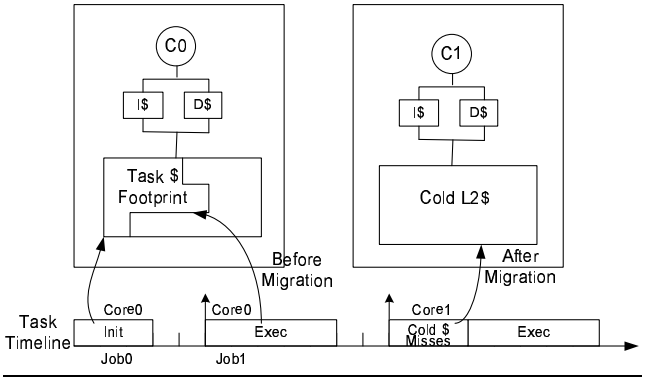

Figure 1. Task Execution amid Task Migration

3. Related Work

Yan and Zhang have recently proposed techniques to calculate the WCET of tasks in real-time multicore systems [50, 49, 51], and other approaches develop multi-level WCET cache analysis as well [36, 24]. 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 [18]. 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 [30, 31]. 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 [5, 13]. 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 [22], but they do not target real-time systems and do not address task migration issues.

Eisler *et al.* [21] 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.* [11, 4] 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 [2]. Their focus is on task replication and re-creation across different memory spaces while our work focuses on task migration within part shared, part private memory spaces.

4. Problem Analysis

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

Experimental Architecture: Our experimental model is an SMP based architecture. This choice was made so as to exhibit similar properties to the contemporary NUCA-based architecture minus the interconnects [27]. 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 1MB L2 cache. The standard MESI coherence protocol is used to maintain coherence across L2 Caches. SESC [39], an event driven simulator is used to design this architecture.

Experimental Model: Figure 1 shows the experiment methodology of this work. Real-time applications are usually composed of tasks that are invoked periodically. 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 the tasks fit within the local cache, then through static timing analysis one can use the L2 cache hit latency instead of the memory latency as the upper bound for each load. This approach gives a much tighter bound for the WCET of a task. However, in a multicore environment, this does not hold anymore as subsequent jobs of the task might be scheduled on a different core as shown in Figure 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.

Benchmark Name	Dataset Size (kbytes)	Before Migration (cycles)	After Migration (cyles)	WCET Dilation (%)
bs	1024	1530	2396	56.6
crc	1	7000	8332	19
cnt	308	2014500	2309385	14.6
stats	1010	15201732	16176250	6.4
srt	2	4003360	4007100	0.1
matmult	2.6	955420	963720	0.9

Table 1. Task migration & dilation in WCET

Matrix Dimension	Before Migration (cycles)	After Migration (cyles)	WCET Dilation (%)
15x15	189800	192700	1.53
30x30	1457400	1468670	0.77
60x60	11449750	11485155	0.31

Table 2. Matrix multiplication WCET v/s matrix size

Array Elements	Before Migration (cycles)	After Migration (cyles)	WCET Dilation (%)
50	161160	162400	0.77
100	645652	647000	0.21
250	4007302	4010504	0.08

Table 3. Bubble Sort WCET v/s Number of Elements

Impact of Migration on WCET: Experiments were performed over a subset of the WCET benchmarks from Malardalen [3] as shown in Table 4. Each of the benchmarks were executed as described in Experimental Model. The results of the experiments as

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	$O(n^3)$

Table 4. WCET Benchmarks

listed in Table 1. 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 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 lower than or equal to $O(n)$ 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 good 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 1. 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.

We conducted experiments to obtain the execution cycles for different size of data sets for Matrix Multiplication and Sort. Tables 2 and 3 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

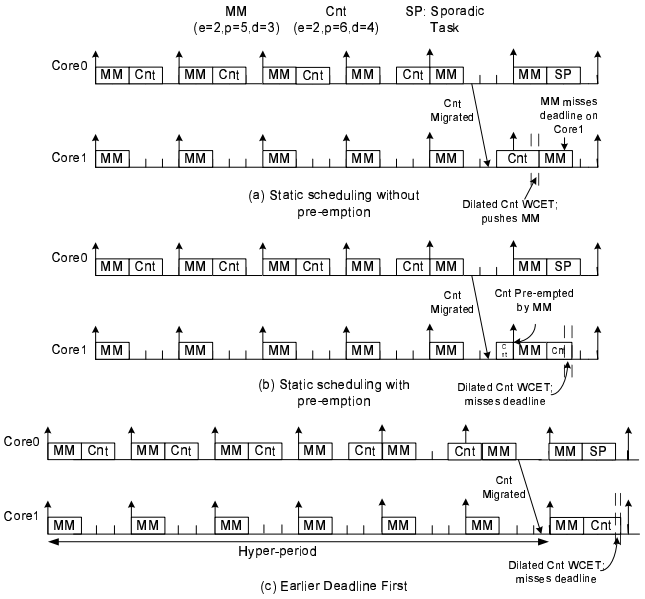

Figure 2. Task migration and scheduling anomalies

Figure 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(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 actuator. An interrupt from which may necessitate the 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 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(a).

One might argue that if the system was preemptive then matmult would have preempted the migrated task and met its deadline. However, Figure 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. The failure of static scheduling to counter the impact of task migration motivated us to 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(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.

5. Proposed Solution

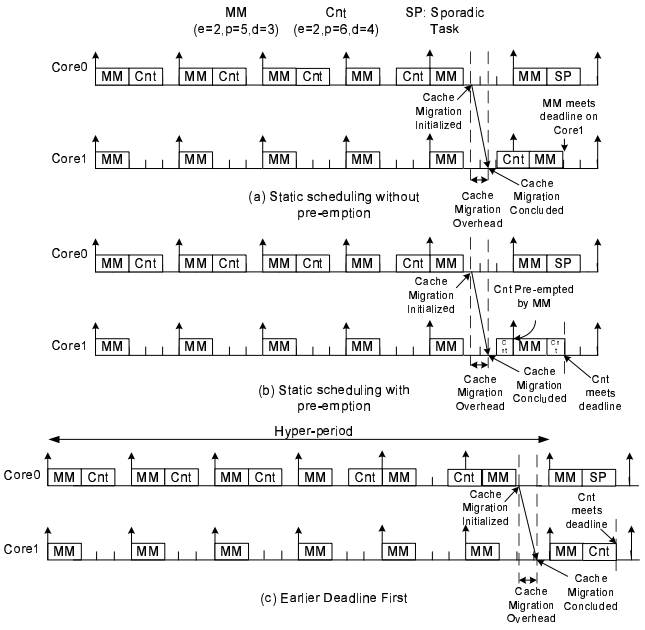

Figure 3. Task Migration Coupled with Cache Migration

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. This task migration scenario was exemplified in Figure 2. We propose a micro-architectural design enabling migration of L2 cache lines from one core to another. This migration is initiated by the scheduler.

As discussed earlier, the architecture of contemporary multicore designs is completely ignorant of task migration decisions by the scheduler. A simple experimental scenario was constructed (see Figure 2). It illustrates this inability inherent to task migration leading to a deadline failure of scheduled tasks. Our proposed solution is a micro-architectural design allowing the scheduler to propagate the information of task migration to the affected cores. The same task set is modeled with our proposed solution shown in Figure 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 our 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). In all three experiments, our novel approach of cache migration allowed tasks to meet their deadlines. This emphasizes the potential of our micro-architectural enhancement to multicore design to bound the execution time of real-time tasks under task migration.

Since this work has the potential to spark a new direction in research, our focus here is to establish the problem and present a first approach to tackle it. The feasibility of the approach will be discussed in following sections. The following are the assumptions that underlie our experimental model. Presently, we consider a bus-based system as is common in many multicores instead of network-on-chip interconnects with multi-hop routing. This allows us to gauge the potential of cache migration in terms of the cache footprint of tasks and their execution complexity. A study of task migration involving scalable multicore interconnects is beyond the scope of this paper. To isolate the effect of migration in the experimental model, we currently constrain our schedule to one with a single task migration, *i.e.*, simultaneous migrations of tasks are beyond the scope. Our experimental model requires that cache migration take place during idle slots for both source and target cores (or any other bus activity). If migration were to take place during tasks' execution on any of the cores, push-assisted cache migration accesses might contend with cache accesses due to active tasks. Multi-rail interconnects could mitigate this problem by providing a separate migration network-on-chip, which is also beyond the scope of this paper. Furthermore, this work does not focus on inter-task cache conflicts as such conflicts are orthogonal to our work and have been studied elsewhere [29, 42, 43, 38]. Overall, our aim is to isolate tasks from non-migration inflicted cache conflicts while executing multiple tasks on a core. This allows us to isolate the inherent overhead due to migration without diluting them with other side effects.

6. 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.

6.1 The 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.

6.2 A Novel Push Model

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.

6.2.1 The Push Block

The push block has the responsibility to identify migratable lines. Any valid cache line cannot be considered as a migratable line. 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 transactions.
2. The target will experience cache pollution because of the displacement of the valid lines of some other task running on it.
3. On reinvocation of dormant 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.

6.2.2 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.

6.2.3 Regional Cache Migration (RCM)

RCM allows programmers to define the region of the expected cache footprint. 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. This is accomplished by software support 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. The real-time operating system (RTOS) saves the region register values as part of the context of the task. 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.

6.2.4 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.

7. Simulation Platform

Our proposed solution requires micro-architectural modifications to a stable bus-based multicore CMP architecture. We chose SESC [39], 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 [52] 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.

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	1MB/4-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

Table 5. Simulation Parameters

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 [25] 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 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.

8. Evaluation

Benchmark	Tight WCET "Warmed Cache" [cycles]	Execution Time "WCM" [cycles]	Deviation from tight WCET (%)
bs	1530	1848	20.78
crc	7000	7590	8.43
cnt	2014500	2015161	0.03
stats	15201732	15202426	0.004
srt	4003360	4004450	0.03
matmult	955420	957521	0.22

Table 6. Cache Migration & Potential to Bound WCET

Cache Migration & Potential to Bound WCET: We performed an evaluation on the simulation platform discussed in the previous section. Table 6 shows the potential of the push-assisted cache migration scheme in making migrated task perform close to their tighter WCET bounds. The whole cache migration scheme provides the upper bound for the entire cache context of the migrated task. This hides most latencies so that migration remains nearly unnoticed by the affected task. However, our current implementation of the whole cache migration scheme pushes the lines from one L2 to another but does not push it up to the L1 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. While this deviation from the tighter WCET provided by the warmed-up cache is not significant for cnt, stats, srt and matmult, it is considerably high for bs and crc. This is because the execution cycles of both of these tasks are very small. Benchmark bs has an algorithmic complexity of O(log n), due to which, in spite of having a large memory footprint, its execution time is very small. Crc has a small execution time because it computes over a very small data set even though it has an algorithmic complexity of $O(n)$. Hence, we can deduce that L1 cache misses can not be ignored 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 of matmult and bubblesort.

Benchmark	Execution Time "WCM" [cycles]	Migration Overhead [cycles]	Overhead vs. Execution Time (%)
bs	1848	1703923	92203.6
crc	7590	44521	586.6
cnt	2015161	542693	26.9
stats	15202426	1691015	11.1
srt	4004450	49559	1.2
matmult	957521	55947	5.8

Table 7. WCM & Cache Migration Overhead

WCM & Cache Migration 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.

Table 7 shows the amount of overhead that WCM incurs. The third column shows overhead in the absolute number of cycles. The fourth column shows the same as a percentage of the entire execution time. These percentages are quite large for bs and crc because the execution time for these two benchmarks are pretty small. BS has a large cache footprint. Thus, the amount of data to be migrated is large as well. This cannot be avoided unless we have prior knowledge about the access pattern for the next invocation of the task. The benchmark crc has a small cache footprint, but WCM has the drawback of scanning through all cache lines irrespective of whether the line is migratable or not. Furthermore, the size of the cache may have an impact because it directly correlates to the number of cache lines that WCM scans. Thus, these extra cycles impose a large overhead, especially for tasks with small cache footprint, thereby causing the execution time to grow linearly with the data set size. However, apart from bs and crc, the remaining benchmarks experience migration overhead of less than 30% of the targeted execution cycles.

Practical Issues with WCM: WCM allows the complete footprint of the tasks to be migrated. However, its practicality is limited by the number of extra cache line reads that it has to perform since it does not have knowledge about the migrated task's cache footprint. Each migrated line incurs the latency overhead of a push request while every extra cache line read incurs a read latency. Thus, the overhead becomes a function of the cache footprint and the cache size. Contemporary L2 cache sizes are in the order of MBs, which is potentially orders of magnitude larger than the cache footprint of the real time tasks. Thus, the migration overhead becomes pro-

portional to the size of the L2 cache instead of the task's cache footprint. WCM also imposes the requirement of storing Task IDs associated with each cache line and extra comparison logic. This increases the die area, power requirements and hardware complexity, thereby affecting the access time of L2 Caches.

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 6.2.3.

Benchmark	No Cache Migration (%)	WCM (%)	RCM (Data Only) (%)	RCM (Data + Instr) (%)
bs	56.6	20.8	-	-
crc	19	8.4	14.9	8.9
cnt	14	0.03	0.06	0.05
stats	6.4	0.004	0.008	0.006
srt	0.09	0.027	0.045	0.028
matmult	0.86	0.22	0.24	0.23

Table 8. Migration Paradigms and Percent Additional Execution over WCET

Table 8 compares the execution time of the benchmarks with different cache migration schemes. Each value in the table is the percentage of additional cycles over WCET required by a migrated job to finish under the respective cache migration scheme. The first column exhibits the problem of dilation in execution time of migrated tasks. This might cause the migrated tasks or other tasks in the same task set to miss their deadlines (Section 4). The second column shows the smallest observed overhead incurred if the whole cache footprint is migrated. The third and the fourth columns are variants of the Regional Cache Migration (RCM). Scheme RCM (Data Only) reduces the number of additional execution cycles to less than a percent for most of the benchmarks, except for crc. This is because the data set size of crc is very small and its algorithmic complexity is $O(n)$. Hence, the cache misses caused by instructions become dominant. This underlines the significance of the fourth column showing that, if the developer is able to specify both data and instructions clearly, it can match the effect of WCM. WCM results for the bs benchmark show a significant reduction in execution time (from 56.6% to 20.8%) but the increase in execution time is still quite high. The identification of critical memory regions for bs is challenging as this benchmark does not access all the records on subsequent invocation. Hence, user-configured RCM regions would be identical to the whole data set, which is quite large. Thus, RCM would not be effective for this particular benchmark, and those experiments are therefore omitted.

Benchmark	WCM (%)	RCM (Data Only) (%)	RCM (Data+Instr) (%)
bs	92203	-	-
crc	586.57	18.23	32.07
cnt	26.93	25.28	25.29
stats	11.12	11.11	11.114
srt	1.23	0.08	0.09
matmult	5.84	1.38	1.41

Table 9. Cache Migration Overhead cycles over Target Execution cycles [percent]

Overall, the feasibility of any cache migration scheme 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 is as stated in Table 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 in Table 9. The table depicts the ratio of the overhead to the WCET bound in percent. The first, second and third columns show those ratios for WCM, RCM with data regions migrated, and RCM with regions for both data and instruction migrated, respectively. We observe that the overhead does not vary much for cnt (26%) and stats (11%) across the various schemes. This is due to data set sizes for cnt and stats being large. Thus, migration of those lines dominates the overhead as it constitutes the cache footprint. Also, RCM with data migration is able to obtain task execution times comparable to WCM, as seen in Table 8. However, crc has a small cache footprint. Therefore, the migration overhead reduces from 586% for WCM to 18% for RCM with data regions migrated and 32% for data+instruction region migration. However, the performance of RCM with both 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.

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 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. Results reported for cache migration schemes were taken using the RCM as the migration model except for bs due to the ineffectiveness of RCM for this benchmark (see discussion above). For the other benchmarks, the execution time of a migrated task is closely resembling the WCET, and overhead (not charged during task execution but rather in idle slots) represents only a fraction of the total execution time.

It can be deduced that those tasks accessing their complete data set will have minor or no difference in the total accesses

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	32753	11	1	32753	11	1
crc	0	35	5	47	6	5	47	6	4
cnt	0	9809	1	9807	2	1	9807	2	1
stats	0	32589	7	32493	166	7	32493	166	7
srt	0	79	64	70	9	64	70	9	1
matmult	0	182	86	260	8	86	260	8	1

Table 10. Cache Migration and Bandwidth Overhead[number of requests]

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 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 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 paper, we focus on it as part of future work.

9. Conclusion

This paper 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 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 paper 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 $O(n)$. Even tasks with higher complexity show a significant dilation for small data set sizes.

This paper proposes two schemes of push-assisted cache-to-cache migration in multicores as a means to diminish the dilation introduced by the target warm-up overhead. Our first scheme, a hardware scheme replicating 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 complexity lower than $O(n)$. This shows that L1 cache misses also pose a significant risk in their dilation of execution times under task migration. Our second scheme extends software support by a hardware scheme so that developers may specify address ranges associated with the task. Through these contributions, migration overhead for all the tasks is reduced to less than 33% of the task's execution time. This makes software-assisted cache migration a feasible solution for preserving execution times after task migration close to those tight WCET bounds otherwise known only 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.

Overall, we have presented a practically feasible push-assisted cache migration approach to tackle the unpredictability in WCET caused by task migration that has no precedence.

Acknowledgements

This work was supported in part by NSF grants CCR-0237570, EEC-0812121, and U.S. Army Research Office (ARO) grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI). We thank James Tuck for his help with SESC Simulator. We also thank Niket Choudhary for discussions on contemporary processor design specifications.

References

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

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

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

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

[5] 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, Apr. 2006.

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

[7] 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.

[8] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, Dec. 1994.

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

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

[11] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. 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.

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

[13] 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.

[14] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *International Symposium on High Performance Computer Architecture*, pages 340–351, 2005.

[15] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2000.

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

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

[18] 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.

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

[20] N. Eisley, L.-S. Peh, and L. Shang. In-network cache coherence. In *International Symposium on Microarchitecture*, pages 321–332, 2006.

[21] N. Eisley, L.-S. Peh, and L. 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.

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

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

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

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

[26] 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, Nov. 2008.

[27] S. W. Kim, M. Voss, B. Kuhn, H.-C. Hoppe, and W. 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, Apr. 2002.

[28] 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.

[29] C. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *IEEE Real-Time Systems Symposium*, pages 700–713, Dec. 1996.

[30] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *In ACM/IEEE conference on Supercomputing*, pages 1–11, Nov. 2007.

[31] T. Li, P. Brett, B. Hohlt, R. Knauerhase, S. 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.

[32] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

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

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

[35] 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, Dec. 1999.

[36] 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.

[37] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

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

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

[40] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. 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.

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

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

[43] J. Staschulat, S. Schliecker, and R. 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.

[44] K. Strauss, X. Shen, and J. Torrellas. Uncorq: Unconstrained snoop request delivery in embedded-ring multiprocessors. In *International Symposium on Microarchitecture*, pages 327–342, 2007.

[45] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196, 2002.

[46] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, Nov. 2001.

[47] R. Wilhelm, J. Engblohm, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, Apr. 2008.

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

[49] 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*, Dec. 2007.

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

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

[52] M. Zhang and K. Asanovic. Victim migration: Dynamically adapting between private and shared cmp caches. TR 2005-064, MIT CSAIL, 2005.