Predictable Task Migration for Locked Caches in Multi-Core Systems *

Abhik Sarkar[1], Frank Mueller[1], Harini Ramaprasad[2]

[1]North Carolina State University,[2]Southern Illinois University

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

Abstract

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. However, cache locking poses a challenge for multi-core hard real-time systems since theoretically optimal scheduling techniques on multi-core architectures assume zero cost for task migration. Tasks with locked cache lines need to proactively migrate these lines before the next invocation of the task. Otherwise, cache locking on multi-core architectures becomes useless as predictability is compromised.

This paper proposes hardware-based push-assisted cache migration as a means to retain locks on cache lines across migrations. We extend 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 the scheduler decide which migration technique(s) to utilize to relocate 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 multi-cores is unprecedented to our knowledge.

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.

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

1. Introduction

Locking cache contents in uni-processor 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, real-time tasks that are usually of small cache footprints and low intra-task conflicts, are able to obtain a shorter worst-case execution time (WCET) by using locks.

Today, uni-processor designs have reached a clock frequency wall due to their fabrication process and power/leakage constraints, which has led processor vendors catering to both high-performance and embedded computing communities to design chip multi-processors [22]. This trend is evident in embedded heterogeneous multi-cores (*e.g.*, for cell phones) and also homogeneous multi-cores [2].

Acceptance of multi-cores in hard real-time systems has been slow due to several reasons. One of the early hurdles was to develop schedulability analysis for hard real-time tasks on multi-cores. By now, real-time schedulability theory for multi-cores has advanced and diversified to a multitude of policies for global and fair scheduling [6–9, 13, 25].

A fundamental premise of these policies is that tasks are migratable. 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 migrations of multiple tasks* that are prevalent in such scheduling mechanisms. Locked cache lines only add to this problem. Prior work on cache locking has been studied only in relation to uni-processors. Conventionally, the programmer incurs a cost of loading the locked cache lines prior to the execution of the task. Subsequently, execution of the task assumes cache hits for locked lines. This does not hold true in context of task migrations as they can occur during the execution of the task.

To ensure predictability, one could propose to pin hard real-time tasks onto cores and thus partition them on multi-cores. This approach generates sub-optimal schedules as optimal partitioning of tasks on multi-cores has been shown to be an NP-hard problem [13]. Additionally, dynamic task admittance would require re-partitioning that renders such a system inflexible.

Thus, we identify locked cache line mobility as a hindrance to task migration in real-time systems. Any solution to preserve predictability for locked cache lines has to be proactive in nature to guarantee that these lines are locked at the target core before a task resumes execution.

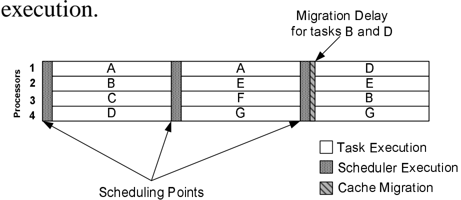

Figure 1. Cache Migration follows Pfair scheduling

This paper focuses on the migration of locked cache lines and providing a predictable task migration scheme applicable to theoretically optimal PFair scheduling algorithm [17]. Cache migrations can only be initiated by the scheduler after it has made migration decisions. All tasks are stalled until all cache migrations have completed. The resulting migration cost will add to the scheduling cost, as shown in Figure 1. The simplest way of performing proactive cache migration is by using a software model. A thread/task is spawned at the source core to unlock the lines. Thereafter, another thread spawned at the target core has to load and lock these lines. For multiple task migrations taking place between different cores, can execute locking/unlocking threads on those cores in parallel. However, such a scheme has following drawbacks:

* This work was supported in part by NSF grants CNS-0720496 and CNS-0905181.

LCTES'11, April 11–14, 2011, Chicago, Illinois, USA.
Copyright © 2011 ACM 978-1-4503-0555-6/11/04. . . $5.00

1. While performing multiple cache migrations using migration threads, each thread is oblivious of concurrently running cache migrations and unaware of the bus utilization. Unsynchronized issuance of demand requests from multiple target cores leads to unpredictable cache-to-cache request delays. *This makes parallel transactions unpredictable.*
2. Each demand request in a contemporary bus-based SMP will be snooped by all caches (ignoring coherence filters for now). This will induce multiple response actions on all cache controllers of the system. This not only affects the bus bandwidth but also causes useless cache accesses. Hence, *parallel transactions become highly unpredictable and inefficient.*
3. Also, in hard real-time systems, a software model executes the locking/unlocking thread on a simple in-order core. This means that a thread that loads and locks the cache lines on the target for a task can issue only one demand request at a time, i.e., the subsequent demand for a cache line can only be issued after the current demand request has completed. Thus, if there is a single task migration taking place in a system, a thread-based model is unable to utilize the complete bus bandwidth.

The closest work with regard to proactive cache migration is that by Sarkar et al. [24]. Their 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. The authors assume tasks are statically scheduled on every core, each core running its own scheduler instance. They present a single task migration scenario that triggers a task migration following dynamic task admission. A task that has ample slack time before its next invocation is chosen to migrate. The task's slack time is overlapped with proactive migration of cache lines from source to target cores. Thus, the overhead does not contribute to task execution when resuming on the target core, as shown in Figure 2. They also consider a Quality of Service (QoS)-based bus to avoid conflicts from other concurrently running tasks. However, their solution suffers from the following drawbacks:

1. It is not applicable to global schedulers (e.g., PFair [17]). Pfair schedules tasks on quantum basis. The scheduler is invoked periodically and globally synchronous across all cores. Even if a task needs to be admitted dynamically, this decision is made at a scheduling instance only. Thus, the decision of task migration happens just prior to the migration. *In such a scenario, the overhead cannot be overlapped with even earlier slack time.*
2. *Simultaneous migrations of multiple tasks has not been considered.*
3. Their migration costs consider migration of whole cache contexts. This results in loose bounds on migration costs for practical scenarios.

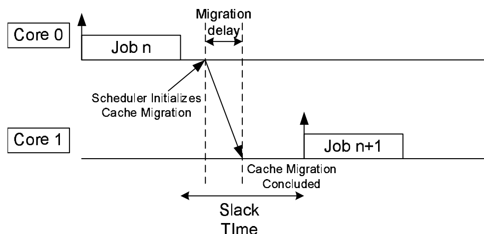

Figure 2. Scheduler Initiated Cache Migration Overlaps Slack Time

These inadequacies led us to exploit push-based micro-architectural cache migration. First, we enhance the current state of the art of push-assisted cache migration to improve the performance of individual cache migrations. Second, we develop a novel scheme that allows the scheduler to orchestrate multiple cache migrations in a seamlessly synchronized and parallel manner. We also derive a pre-calculated bound within which migrations complete.

Our work contrasts with [24] and technically contributes in the following ways:

1. This work is the first one to consider mobility requirements of locked cache lines for task migration. With such support, impractical theoretically optimal multi-core scheduling techniques thus become realistic in the context of hard real-time systems. Prior work is not directed towards locked cache lines.
2. Prior work provides 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 their identification of migratable cache lines to locked cache lines. We then expose the potential of reducing individual cache migration delays by pipelining 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.
3. They propose a hardware mechanism called Whole Cache Migration (WCM) for cache foot prints that are sparse with respect to memory address space. However, they deem it to be impractical as the overhead 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 (SSCM) that presents an efficient and practical solution to sparse locked cache footprints.
4. Their work considers single task migration. This assumption constrains multiple task migrations to be sequential, thereby under-utilizing the bus bandwidth. In this work, we propose a novel mechanism where the scheduler synchronizes multiple RCMs in order to support multiple cache migrations in parallel without any conflicts.
5. Although SSCM works efficiently for single task migration, it cannot be synchronized for parallel migration with RCMs. To handle this, we propose Slotted-SSCM. It allows multiple cache migrations based on RCM and Slotted-SSCM to progress in parallel without transaction 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.
7. In prior work, migration costs are assessed by assuming migration of the total cache foot-print of a task without cache contention. Thus, the overheads stated in [24] for RCM are too loose to be applicable to hard real-time systems. In contrast, our work considers inter-task contention and migrates locked caches lines. We present deterministic migration delay bounds for each scheme irrespective of whether cache locks are used to lock contiguous or sparse memory locations.
8. In Section 7, we provide insights into
   (a) applying our parallel migration mechanism to SMP architecture with TDMA-based bus systems and
   (b) applying our pipelined mechanisms to tile-based architectures with mesh interconnects.

2. Related Work

In the past decade, there has been considerable research on cache line locks in the context of multi-tasking real-time systems. Static and dynamic cache locking algorithms for instruction caches have been proposed to improve system utilization in [18,19]. Data cache locking techniques that pin data when cache behavior is hard to analyze statically have been proposed [28]. Past work presented techniques for cache locking that provides comparable performance to scratchpad allocation [20]. Recently, cache locking has also been proposed for multi-core systems that use shared L2 caches [27]. Of course, cache locking can also be used in conjunction with private L2 caches. Multi-cores certainly make cache locking even more attractive in terms of real-time predictability.

Choffnes *et al.* propose migration policies for multi-core fair-share scheduling [12]. Their technique strives to minimize migration costs while ensuring fairness among the tasks by maintaining balanced scheduling queues as new tasks are activated. The work is in the context of soft real-time systems while ours focuses on hard real-time. Calandrino *et al.* propose scheduling techniques that account for co-schedulability of tasks with respect to cache behavior [5, 11]. 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. Li *et al.* discuss migration policies that facilitate efficient operating system scheduling in asymmetric multi-core architectures [16]. Their work focuses on fault-and-migrate techniques to handle resource-related faults in heterogeneous cores and does not consider real-time constraints. Eisler *et al.* [14] develop a cache capacity increasing scheme for multi-cores 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 and retains the locks in caches across migration.

Acquaviva *et al.* [4, 10] 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 multi-cores [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. Hardy *et al.* have recently proposed static cache analysis techniques to quantify cache-related migration delay cost on multi-cores by estimating re-use of cache lines that cause cache misses [15]. Our methodology focuses upon eliminating migration delay to support cache line locking in multi-cores and providing support for deterministic migration delay for locked cache lines.

3. Problem Analysis

Era of Multi-core architectures: The multi-tasking requirement of real-time systems and the higher processing needs have motivated embedded system vendors to advocate multi-core processors. These designs leverage multi-processing ability instead of only instruction-level parallelism. Thus, multi-cores have simpler cores but complex cache architectures. Symmetric multi-processors (SMPs) have already become popular among consumer electronics. Figure 3 exhibits the two SMP designs that are widely deployed. Both architectures have three levels of caches except that in Figure 3(a) L3 cache is shared among processors while, in Figure 3(b), sharing begins at L2. Thus, the latter design suffers from contention at L2, which results in high contention as L2 is much smaller than L3. Therefore, the former design seems more suitable for real-time systems that execute multiple independent tasks.

Also, tile processors from Intel and Tilera with on-chip message passing capabilities have recently been advocated [2,22]. Thus, for predictable behavior we consider the design shown in Figure 3(a) for our study.

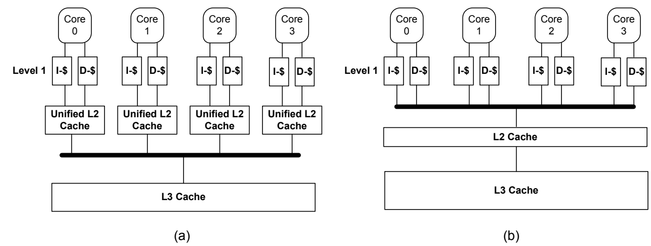

Figure 3. Symmetric Multi-processors

Task Migration on Multi-core Platforms: Scheduling of real-time tasks on multiple cores has been widely studied. Researchers have contributed by proposing global and fair scheduling algorithms that assume task migrations are allowed. However, allowing task migrations makes it necessary to consider the cost of migration. When a task migrates from one core to another, it starts executing on the new core with a cold cache as shown in Figure 4. This incurs a Cache-Related Migration Delay (CRMD). Hardy et al. determined the upper bound on CRMD for shared instruction caches [15]. However, their scheme is limited to the estimation of CRMD and does not consider mechanisms to reduce migration overheads. Sarkar et al. propose push-based cache migration schemes for migrating the entire cache context of a single task [24]. However, their solution of overlapping migration overhead with slack time within a schedule is not applicable to theoretically optimal PFair scheduling, as discussed in Section 1. Furthermore, the immobility of locked caches in multi-core real-time systems is not addressed in literature. In contrast, we specifically focus on the importance of locked caches in hard 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. Such systems are employed in highly sensitive environments where a missed deadlines may have catastrophic consequences. A real-time system is usually composed of short hard real-time tasks sharing common resources with other soft real-time tasks and non real-time tasks. In such mixed criticality 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, Hard Real-time tasks can fit within the cache with low or no intra-task contention. None the less, inter-task cache contention with other non real-time tasks hampers the cache behavior predictability of hard real-time tasks. Thus, cache locks are necessary to improve the predictability of cache behavior. As a side effect, it may reduce execution time of hard real-time tasks assuming that they are small since intra-task cache contention may be low if not eliminated.

Cache locks have long been studied for uni-processor 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. Dynamic cache locking is more suitable for soft real-time tasks that have high intra-task cache contention.

Impact of Cache locks on multi-tasking systems: We conducted experiments on a uni-processor model with and without

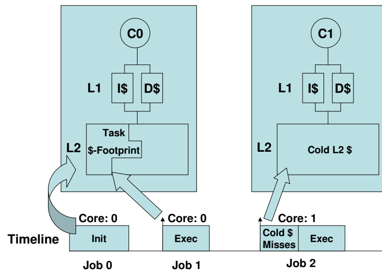

Figure 4. Impact of Task Migration on Execution Time

Table 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)

cache locks for hard real-time tasks to substantiate the impact of cache locking for hard real-time systems. Table 1 shows a subset of Malardalen WCET Benchmarks [3] used in our experiments as hard real-time benchmarks. Each of these benchmarks were run individually along with a non real-time Cnt benchmark.

Assumptions of this study: We chose SESC [23], a light weight event driven simulator, to design our experiments. Our processor model consisted of an in-order core with multi-level inclusive caches. 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. Our experimental cache hierarchy has 2KB 4-way associative L1 data and instruction caches, and a 8KB 8-way associative unified L2 cache, with same cache line size of 32 bytes. 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. 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 However, lack of such a WCET bounding tool neither prevents developers from using cache locks in current hard real-time systems, nor does it change the problem addressed in this paper. Thus, development of such an analysis tool is orthogonal and out of the scope for this paper. 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 refactored as stand-alone hard real-time tasks.

Table 2 shows the impact of cache locking when each of the benchmarks is run in contention 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 vs. 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 these lines are locked. Notice that locks show a tighter bound and lower observed WCET due to reduced intra-task cache contention. Such behavior is typical for hard real-time applications.

Table 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 %

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 [18, 19, 21, 26, 28].

Task Migration and Locked Caches on Multi-core Platforms: Developers choose to use locks on memory addresses with the assumption that when a task executes on a core, the core's cache holds all locked lines. However, task migrations are scheduler-triggered events that a developer has a limited control over. Thus, developers may pin a process onto a core in a multi-core environment. This precludes the use of efficient real-time scheduling techniques that rely on task migration. If tasks with locked lines were allowed to migrate, locked lines would still remain in the source core (abandoned by the task) while the task resumes execution on a new (target) core. This violates the predictability assumptions of the locking mechanism per se. As discussed in Section 1, using threads to unlock (at the source) and lock (at the target) have significant drawbacks. Hence, multiple cache migrations have to be serialized to ensure predictable cache migration. On top of that, each cache line is migrated sequentially due to in-order cores. Thus, there is no scope for reducing migration costs while maintaining predictability.

4. Proposed Solution

4.1 Assumptions on Multi-Core Real-Time System

In this paper, we assume an SMP bus-based system (see in Figure 3(a)). We further assume a PFair scheduling algorithm that synchronously schedules tasks onto cores after every time quantum

at scheduling slots (see Figure 1). Cache migrations immediately follow scheduling decisions. Tasks are stalled during scheduling slots and resume execution only after completion of scheduling and cache migration. This prevents all tasks from (a) accessing locked data lines during cache migration and (b) generating any bus traffic during cache migration.

We assumed absence of task non-migration bus traffic during push-based migrations to avoid the complexities of bus conflicts when concurrent tasks execute. In Section 7, we discuss the applicability of our mechanisms to systems with bus conflicts while preserving the mathematical bounds derived in the rest of this section. We assume that the target core chosen by the scheduler can accommodate the locked cache lines to be migrated. A study of finding whether there is sufficient space in the target cache is orthogonal to our work as scheduler interaction is required irrespective of the migration type (thread-based pull or hardware-based push). Our objective is to demonstrate that we can guarantee an efficient and predictable cache migration delay once a hard real-time task has been selected for migration.

4.2 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.

4.2.1 Push Model: An Overview

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. Sarkar et al. introduced two such migration schemes: Whole Cache Migration (WCM) and Regional Cache Migration (RCM) [24]. 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, the prior 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 Region 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 sequential addresses 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. See [24] for further details on the hardware complexity of the push block, and integration of push requests with coherence.

Figure 5 depicts diagrammatic representation of two consecutive push transactions using RCM. Each transaction 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 for the previous transfer. This serializes each transaction. If we 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, then the migration delay T_m incurred by a single task migration can be represented as

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

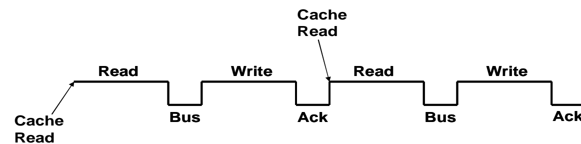

Figure 5. Regional Cache Migration Operational Sequence

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 Pipelining.

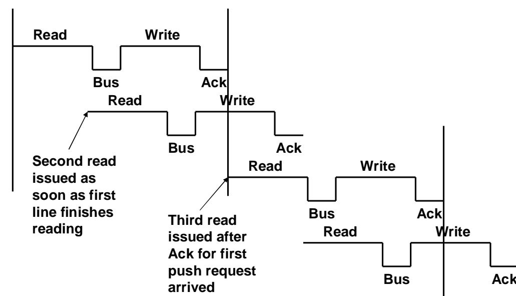

Figure 6. Controlled Cache Migration Pipelining Operational Sequence

4.2.2 Controlled Cache Migration Pipelining (CCMP)

Subsequent cache reads can be serviced while a request has been placed on the bus. 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 6. One unique observation of such pipelining is that bus transactions of push requests and acknowledgments do not interfere with each other. The migration delay for such a scheme is

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

where V=0 when Cn is odd, V=D when C_n is even.

Figure 6 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. Cache-to-cache transfers will continue to be much faster than cache accesses due to advanced interconnect technologies such as HyperTransport (HT) [1]. HT operates at frequencies as high as 3.2GHz, which is much higher than any clock frequency of a known embedded processor. Latencies within embedded network-on-chip interconnects as low as a single cycle have been reported, which confirm this. These observations lead us to develop our next scheme called Streamed Cache Migration Pipelining.

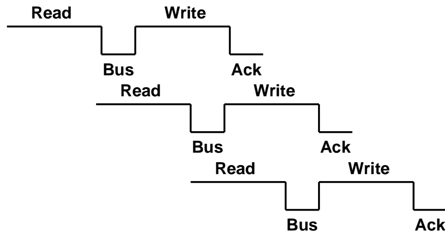

Figure 7. Streamed Cache Migration Pipelining Operational Sequence

4.2.3 Streamed Cache Migration Pipelining (SCMP)

This migration model allows a cache read to start immediately after the previous read without waiting. This scheme is extremely efficient when the bus delay is shorter than half of the cache read access time. As shown in Figure 7, none of the push requests conflict, effectively resulting in streaming behavior of cache migration.

Thus, the migration delay for SCMP is
$T_m = C_n \times D + (2 \times B + D)$.

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

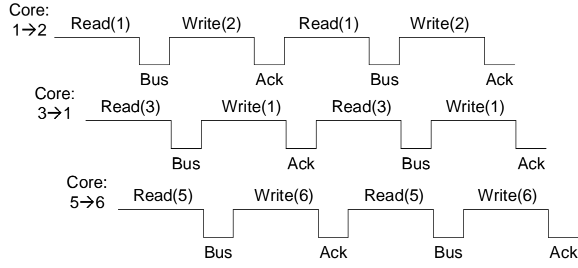

Figure 8. Parallel Multiple RCMs

4.2.4 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 time difference that is a multiple of D between cache reads of any two RCM chains, we can support multiple cache migrations without contention (see Figure 8). 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 8. In fact, the only scenario when synchronization does not hold true is for inverted source,target pairs. But PFair will never cause two simultaneous migrations of core pairs (1,2) and (2,1). In PFair scheduling, a task is first picked to execute. If it 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. 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).

There is a limit to the number of transactions that can be supported in parallel, which is equal to $\lfloor D/B \rfloor$. Parallel transactions have the ability to maximize the utilization of the bus bandwidth. Such a scheme additionally requires synchronization across cores. The key to this kind of a scheme is the method used by the scheduler to initiate and synchronize these transactions. 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. A variation of the count sort algorithm divides these migrations into buckets. Each migration within a bucket can run in parallel. Each bucket with respect to another is sequential. The system can support maximum bucket size of $\lfloor D/B \rfloor$ transactions that can run in parallel. Buckets with a large number of parallel transactions are split. The complexity of this procedure is $O(number_of_cores)$. Partial ordering is then obtained among transactions within each bucket to place core pairs like (1,2) and (3,1) next to each other. The complexity of this is O($bucket_size$).
3. The scheduler stores offsets into the TCB of migrated tasks. These offsets indicate cycle times when a particular transaction starts relative to the start of cache migrations. The scheduler then loads the contexts onto the target cores.
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 pipelined 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.

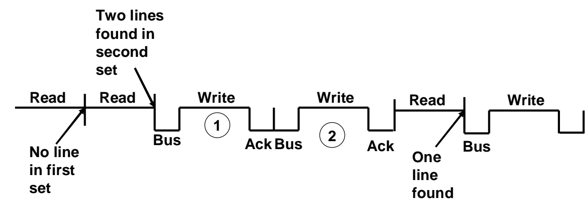

Figure 9. Set-Scan Cache Migration Operational Sequence

4.2.5 Set-Scan Cache Migration (SSCM)

The push block in SSCM identifies 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 with a process. During a normal cache access, the contents of an entire are read. If the searched tag matches any entry, it is forwarded to the bus. However, multiple locked entries within a set may belong to the same process. So, efficiency of migration may be increased by buffering the rest of the entries in the set instead of throwing them away. Thus, multiple matching entries can be identified in very short time while the first matched entry is in transit. On receiving an acknowledgment, the next matching entry can be transmitted immediately without any read delays. This is shown in Figure 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 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. It is calculated from the size of cache (S_c) and the associativity (A). The migration delay T_m is

$T_m = S_c/A \times D + C_n \times (2 \times B + D)$

This is deduced from the behavior of SSCM, which reads each set once. Each migration takes two bus transactions and one cache access at the target.

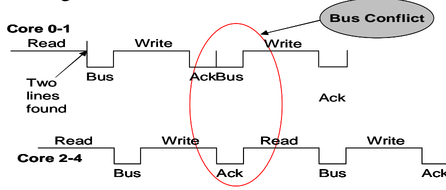

Figure 10. Conflicts between RCM and SSCM Push Requests

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 10 illustrates the reason for this: a set read may find zero or multiple lines to migrate. Thus, we propose an aligned version of the SSCM called Slotted-SSCM.

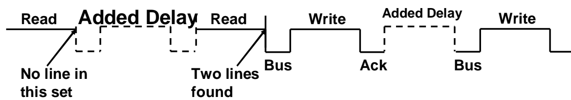

Figure 11. Slotted-SSCM Operational Sequence

4.2.6 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. This is shown in Figure 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 delay of fake set reads added to migrations of cache lines that have been identified by a prior set read operation. The worst-case scenario occurs when locked cache lines are locked in $\lceil Cn/A \rceil$ cache sets while the best-case cache migration delay is obtained when each cache set has one locked cache line.

Thus, the worst-case cache migration delay experienced by Slotted-SSCM is

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

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.

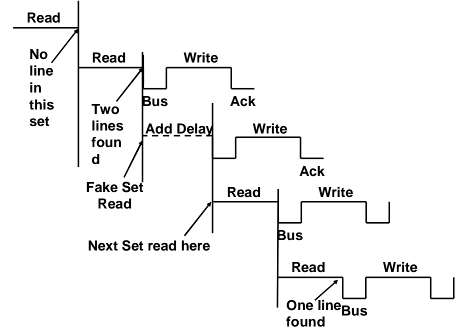

Figure 12. Slotted-SSCM Pipelining Operational Sequence

4.2.7 Slotted-SSCM Pipelining

This migration model takes us back to pipelined 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 is associated with every migrated line. 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 12. This migration model exhibits a complexity in analytical terms. The worst-case scenario is the same as for Slotted SSCM. Hence, the worst-case cache migration delay for Slotted-SSCM Pipelining is

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

4.2.8 Proactive Pull Model

The proactive pull model is a migration model where demand requests are issued by the target core before the migrated task resumes execution on the target. We did not consider a hardware model for this because it is impossible to create an SSCM-like migration model.

5. Simulation Platform

Table 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
Bus Width	256 bits
External Memory Latency	100 cycles

Our proposed solution requires micro-architectural modifications to a stable bus-based multi-core CMP architecture. We chose SESC [23], 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 multi-core CMP architecture where each core has private L1 and L2 caches.

The system architecture specifications are presented in Table 3. The base simulator environment has been enhanced with a scheduler that triggers task migration across cores. 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.

6. Evaluation

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 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. 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 sets in the cache.

Table 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

Pipelined RCM techniques: In Section 4.2, we presented two similar pipelining schemes, CCMP and SCMP. Table 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 4.2.

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. A detailed analysis of such systems is out of scope of this paper and subject to future work.

Slotted SSCM techniques: In Section 4.2, we presented the Slotted-SSCM and Slotted-SSCM Pipelining migration models.

Table 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%

Table 6 shows the experimental and worst-case migration delays for the two schemes. 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 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

The computation for Slotted-SSCM has been shown in Figure 13. Note that a set with no locked cache line causes a delay of 2x(B+D), but once a line is found this delay covers the first migrated line. Every additional line migrated from a set incurs a delay of 2x(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%.

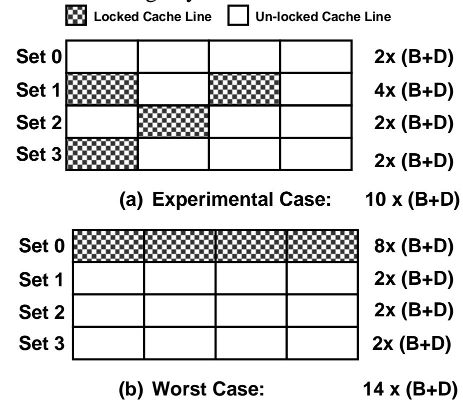

Figure 13. Offline Computation of Migration Delays for Slotted-SSCM

Parallel vs. Pipelined Cache Migration: In this paper, 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 8. 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 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	1128	1366	Parallel
1,2,4	1128	1252	Parallel
2,4	912	768	Pipeline
1,3,4	1128	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 multi-core 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 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 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).

7. Discussion

We have shown that under the assumptions of Section 4.1, our push-based cache migration mechanisms can perform better than any thread-based migration scheme by reducing the migration delay while keeping it predictable. Until now, we were assuming that none of the tasks resume execution during scheduling slots while cache migration is in progress.

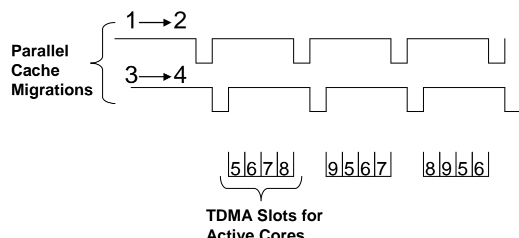

Figure 14. Cache Migration with TDMA bus

We now describe how our cache migration schemes can be incorporated into hard real-time systems such that cores not involved in cache migration may resume execution without waiting for cache migration to complete. We assume that all cache content for a given task is either available in the locked caches on the same core or are obtained from lower-level caches. In other words, we assume there is no coherence traffic between cores and our cache migration is the only means for cache-to-cache transfers.

Literature on multi-core hard real-time systems has advocated the use of TDMA-based buses to enhance predictability of memory access latency amid bus contention [29]. A TDMA-based bus allows bus access to a particular core only during designated slots in a round-robin manner. For example, given four cores and a bus slot width equal to the bus delay (B = 2 cycles), core 0 can place a request on the bus during time slots 0-2, 8-10, 16-18, etc. However, if a request arrives at cycle 1, 9, 17, etc., core 0 will not be able to place its request on the bus until its next slot. Hence, the maximum delay, Max_d, that a core may incur for a bus access, is calculated as $Max_d = B \times A_c - 1$, where A_c is the number of active cores (four in the aforementioned example).

The TDMA-based bus is operates in this manner during execution within each quantum in our system. However, the operation is modified when cache migrations are in progress (see Figure 14). The figure illustrates how our parallel migration mechanism can utilized in conjunction with actively running cores that issue memory/L2 cache requests for TDMA-based buses. Let us assume that two cache migrations are in progress between core pairs (1,2) and (3,4), respectively. Considering a cache delay access, D, of 12 cycles and a bus delay, B, of 2 cycles. The first four cycles (or two TDMA slots) in every 12-cycle period are dedicated for cache migration traffic. TDMA Slots apart from those are allotted to active cores (5, 6, 7, 8) and the memory controller (9), in a round robin sequence. Under such a scheme, the maximum delay Max_d that a core may incur is computed by the following equation:

$$Max_d = N_m \times B \times \lceil A_c / (\lfloor D/B \rfloor - N_m) \rceil + A_c \times B - 1,$$

where N_m is the number of migrations occurring in parallel.

This equation for calculating the maximum delay only applies for the duration of cache migration. Once migrations are complete, the traditional TDMA-based bus operation resumes. Using static analysis for each sub-task executing within a quantum, the worst-case bus latencies can be calculated. Notice that we assume the absence of scheduling decisions within a quantum in order to maintain safety of the system. In other words, we assume a non-work-conserving, static Pfair scheduling algorithm.

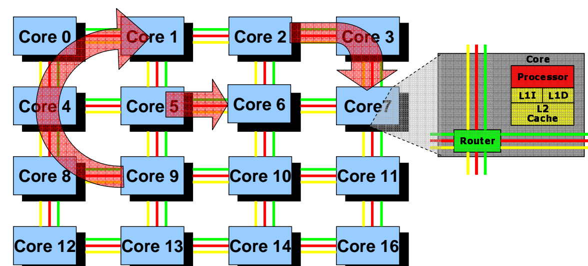

Figure 15. Cache Migration on Tile-based Architecture

In recent times, tile-based multi-core architectures have become a reality [2]. These architectures support multi-channel mesh interconnects with multi-path routing features. Multi-path routing allows multiple cache migrations to be routed on non-interfering paths. This enables multiple cache migrations to proceed in parallel. This has been shown in Figure 15, where three migrations (9 to 1,5 to 6,2 to 7) are taking place simultaneously. Our pipelined mechanism can be utilized to reduce individual cache migrations assuming requests to memory controller use a separate channel. In this case, each cache-to-cache transfer uses multiple hops instead of a single hop shown in Figure 15. The bus delay, B, thus is a multiple of the number of hops (9 to 1: 4 hops, 5 to 6: 1 hop, 2 to 7: 2 hops). Multi-path routing can further assist our push-based lock mechanism. We intend to pursue this line of research in future work.

8. Conclusion

This paper promotes multi-cores 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 multi-core platforms, optimal scheduling like PFair assumes task migration as a fundamental premise. This paper 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 by 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 reduce the overhead of extra cache set reads significantly. We also present novel applications of our migration mechanisms to contemporary multi-core real-time architectures, such as SMP with TDMA-based bus support and tile-based architectures with mesh interconnects.

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. Overall, our novel cache migration schemes provide the scheduler with opportunities to deliver deterministic and efficient cache migrations options.

References

[1] Hypertransport technology. http://www.hypertransport.org.

[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, April 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] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *IEEE Real-Time Systems Symposium*, 2007.

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

[10] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Design, Automation and Test in Europe*, pages 15–20, 2006.

[11] 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.

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

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

[14] 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.

[15] D. Hardy and I. Puaut. Estimation of cache related migration delays for multi-core processors with shared instruction caches. In *Proceedings of the 17th Real-Time and Network Systems*, Paris, France, 2009.

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

[17] 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.

[18] 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.

[19] 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.

[20] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Design, Automation and Test in Europe*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[21] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemptions. *Transactions on Embedded Computing Systems*, 2008.

[22] J. Rattner. Tera-scale research program. In *Intel Dev. Forum*, 2006.

[23] 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.

[24] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. Push-assisted migration of real-time tasks in multi-core processors. In *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 80–89, New York, NY, USA, 2009. ACM.

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

[26] 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.

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

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

[29] E. Wandeler and L. Thiele. Optimal tdma time slot and cycle length allocation for hard real-time systems. In *Proceedings of Asia and South Pacific Design Automation Conference*, Yokohama, Japan, 2006.