Pull based Migration of Real-Time Tasks in Multi-Core Processors

1. Problem Description

The complexity of uniprocessor design attempting to extract instruction level parallelism has motivated the computer architects to leverage parallelism through multiple simple cores on a single chip. Also, with continuous advancement in chip technology chip multi-processors (CMP) have become a reality. Multicores are becoming ubiquitous, not only in general-purpose but also embedded computing. However, on such platforms prediction of timing behavior of real-time tasks 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.

2. Related Work

Real-time tasks are usually periodic in nature and have to be completed before a predefined deadline. Missing a deadline could have serious consequences for hard real time systems. Recent work has shown that the impact of task migration could lead to increase in the execution time starting from 1% percent to 56% [1]. However, in that work a push model has been discussed that modifies the contemporary micro-architecture to enable the cache controller of source core, where the task is currently running and will stop execution, to migrate valid cache lines of the task to the target core, where the task will resume execution. This work overlaps the slack time between subsequent executions of the task on two different cores with migration of valid cache lines such that the target cache is already warmed up before the task starts executing on the target. This prevents the cache warm up from increasing the execution time of the migrated task.The primary disadvantage of the push model is that the contemporary architecture is incapable of pushing the cache lines from source core to target. Hence, the push model requires significant change in the micro-architecture.

3. Pre-fetch Thread based Pull Model

We propose to develop a pull model to migrate the cache lines of the migrated task through memory read requests issued by the target instead of a push request.
Our experimental model is a SMP based architecture. This choice is made so that the design can exhibit properties similar to the contemporary Tile-based [2,6] architecture minus interconnects and directory. It then excludes the complexity introduced by interconnects and uncovers the predictability challenge caused by cache misses only. So, the simulated environment will be a CMP, where each core is a SMT processor[3] that can run two contexts simultaneously. Since such cores are already present, a complete software solution will be one where the scheduler activates a pre-fetching thread at the target as soon as it decides to migrate a task. This pre-fetching thread can run independently of the task that is currently executing on target. This pre-fetcher thread may get the information about the critical regions of the task from the RTOS which it can then use in migrating cache lines. This thread has the lowest priority in the schedule of

the target core. This allows the prefetcher thread to use the idle time on the target core to migrate the cache lines from source to the target. The prefetcher thread is killed as soon as the migrated task is invoked at the target. Thus, the prefetching thread overlaps the migration of cache lines from source to target with the slack time available at the target core. Hence, if the task migration is made predictable, then the scheduler can make an optimal judgement on a core's capacity to admit a task and also, on the task to be migrated. However, the study of increase in execution time experienced by the concurrently running tasks due to contention at memory hierarchy is out of scope of this work.

4. Design Details

4.1. Push Model Simulator Design

The thread migration implemented in [1] assumed only a single thread. Each task is a function call like in case of a cyclic executable. The thread migration was performed by a system call, which stalled the fetch stage and after a designated number of cycles switched the thread from one core to another. So, the current implementation of the simulator is a cyclic executable. Thus, one of the integral parts of this project work is to incorporate capabilities of the SESC simulator to support a scheduler as an independent thread.

4.2. Scheduler Design Extension to SESC Simulator

Following is the design of the scheduler that is being incorporated with SESC simulator

4.2.1. Initialization: The main thread acts as the scheduler. It is pinned on a particular core and does not migrate during the lifetime of simulation. Before, introducing any scheduling routines, it reads a file which contains the specifications of the tasks, like "task name", "period" and "relative deadline". The main thread spawns these threads and suspends all of them using sesc_suspend() system call before entering the scheduler routine. The main thread runs uninterrupted for the whole duration of the simulator. This is because of the following reason:
One might consider that the scheduler can be activated by timer interrupts. However, this means that at some point the scheduler is going to sleep. In situations where all the cores are idle, the scheduler along with the idle cores will not have any tasks executing. This poses an issue with SESC, because the simulations finish executions when there are no tasks running on the simulator. This can be solved by guaranteeing that at any time at least one of the cores is kept busy. Hence, the choice of allowing the scheduler run uninterrupted for the lifetime of simulation was made based upon the simplicity of the design. We are not doing any power study, which can be affected by the proposed design.

4.2.2. Scheduler: The scheduler routine makes a decision on the tasks to be invoked and resumes their execution on their specified cores. Like in most pre-emptive scheduling techniques, the scheduling decisions are made when
(a) A new job is invoked
(b) A currently executing job finishes execution.

The scheduler uses the periodicity information of each task to perform scheduling operations for events specified by (a). However, (b) requires the information of the completion of jobs. On completion, each job updates a unique control variable in the global memory space. This triggers the scheduler to perform scheduling operations. Also, the tasks wait on this control variable value to be reverted when a new invocation of the task needs to be executed.

4.2.3. Migration of the thread: The migration of the threads is an event that needs to be fabricated. The processor cycle when the migration has to take place will be in the input file along with other task information. Once, the scheduler notices that the processor cycle to have crossed the specified cycle value, it will move the task from the source core to target core.

4.2.4. Prefetch thread: Each task has a prefetch routine that contains calls to a tight prefetch loop that issues loads to prefetch cache lines. The arguments of the routine are the base address and stride of the critical region that needs to be prefetched. This thread is spawned at the target core with lesat priority as soon as a thread is migrated. Thus, this prefetch thread makes use of idle time it gets before the invocation of the task on target core to prefetch the cache lines. However, this prefetch thread is killed by the scheduler as soon as it detects the invocation of the migrated task on target core.

5. Infrastructure

This project involves microarchitectural modifications. Thus, we will use SESC simulator [4] to design the system. We will use WCET benchmarks from Malerdalen for testing the correctness of our modifications and effectiveness of our model.

6. Contributions to Simulator Design

The design of the prefetch thread based pull model is the core idea. However, primary design component of the work is multi-core Rate Monotone Scheduler on SESC.
There have been several contributions made by this work to the SESC infrastructure. Following are the key design changes made over the existing push model

(1) Each task is a separate thread that allows the design to be independent of a cyclic scheduler.
(2) The scheduler runs on single designated core. Instead of scheduling routine being invoked on each core, the scheduler monitors the progress of each thread from that core. This avoids caches of the cores running the tasks being polluted by the scheduler's memory contents.
(3) The execution of threads is stopped by sesc_suspend and resumed by sesc_resume system calls. In push model this was done by stalling the fetch stage. Since, the push model had only one thread, any suspension of thread would lead the simulator to exit. However, in current implementation the scheduler thread is active the whole duration of the simulation. Thus is allows us to use sesc_suspend on all the tasks and prevents the simulation from exiting even when all cores running tasks become idle.
(4) The simulator supports thread migration through a moving the task descriptors from one core's queue to another. This allows the scheduler to migrate the task at the time

instant when the request for migration is being made. In Push model design, the task was migrated at the instant when the task is to be invoked at the target.
(5) The simulator has an added system call that allows one thread to kill another thread. This is required when the prefetch thread is needed to be killed at the invocation of the migrated task on target. This is also allows the scheduler to kill all the task threads when the simulation is deemed to be complete.
(6) System calls have been added to the simulator which allows a thread to activate/deactivate data collection for specific L2 caches as in when required.
(7) The default scheduling actions of SESC have been disabled such that on any pre-emption, the scheduler is in complete control of scheduling decisions. Earlier, on any pre-emption of a task from a core, SESC finds a sleeping thread and schedules it onto the core disregarding what is being intended by the scheduler. This posed problems while pre-empting tasks and even while killing prefetch thread.

Apart from implementation of multi-core Rate Monotone Scheduler and contributions made to the SESC simulator, a tight prefetch thread has been designed which dedicates certain registers for prefetching. This allows us to complete the task of prefetching by issuing a single load per prefetched cache line.

7. Simulation Results

7.1 Impact of task migration

It is important to analyze that whether the dilation in WCET of tasks due to their migration is significantly high to cause deadline misses. This is shown in Table 1.

Banchmarks	No Task Migration [cycles]	WCET after Migration [cycles]	Increase in WCET [percent]
Cnt	2014615	2309191	14.62195
Stats	15192208	16158644	6.361393
Crc	7034	8342	18.59539
Matmult	954190	963272	0.951802

Table 1. Impact of task migration on WCET

Table1. shows the WCET before and after task migration, of benchmarks listed in first column, in second and third column, respectively. The third colum shows the percentage increase in WCET time of each of the benchmarks. As shown in [1], there is a significant increase in WCET of benchmarks that have algorithmic complexity of O(n). Task migration has less impact on WCET of matmult because of high reuse of the same matrices as it's algorithmic complexity is of $O(n^3)$. However, any increase in one task's execution time can cause the task or subsequent tasks to miss their deadlines. Thus it becomes imperative to bring the dilation caused by task migration to minimum and thereby predictable range.

7.2 Performance Impact of Pull based Cache Migration

The impact of the prefetch thread is shown in figure1.

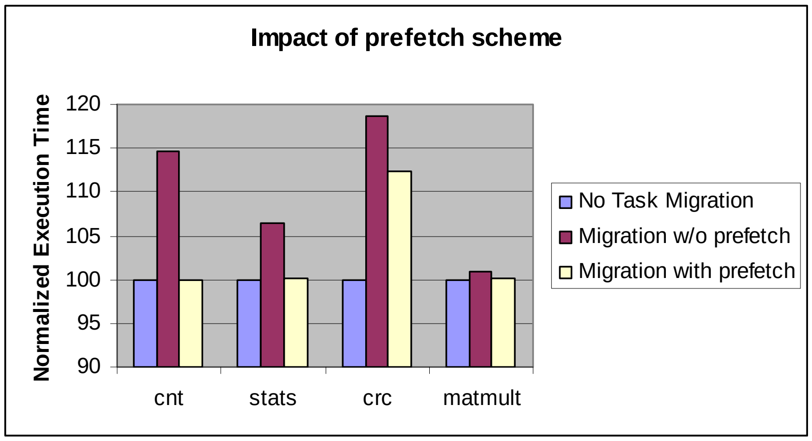

Figure 1. Performance Impact of Prefetch Thread

Figure1. shows the normalized execution time of the benchmarks in different scenarios. It can be inferred that the WCET of all the benchmarks have shown considerable performance improvement due to the prefetching. However, performance of CRC is relatively poor as compared to WCET of the task without any migration. This is because of the following reasons:

(a) CRC has a very small execution time, thus L1 misses can have considerable impact on execution time

(b) Prefetcher is unable to warm up the L1 Instruction Cache which is the major contributor to the dilated WCET of CRC after prefetching.

7.3 Predictability of Prefetch Thread Overhead

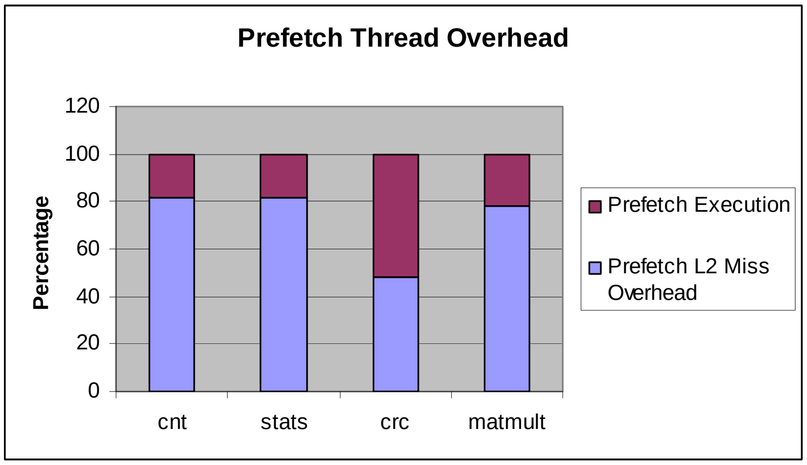

Figure2. Breakdown of Prefetch Thread Overhead

The total overhead is an important statistics since the scheduler can make use of the statistics while performing task migration. However, the overhead overlaps the slack time before the invocation of task at the target. Hence, it is more important that this overhead should be predictable. It can be assumed that the overhead should be a function of the number of cache lines being transferred. However, our results showed that this could not be held true for crc. This is because there is a significant prefetch thread execution overhead experienced by CRC as shown in Figure2. This is because, the critical regions are sparse in the memory layout, thus more number of prefetch routine calls are needed to perform prefetching for a small number of cache lines.

8. Conclusions

This work shows that there is a significant impact of task migration on the WCET. This may lead to deadline misses which are unacceptable for hard real-time systems. Thus, a prefetch thread based cache migration has been designed to minimize the dilation in WCET caused by cold cache misses. The results show that prefetch thread does not work well for tasks whose WCET can experience significant dilation due to L1 instruction cache misses. The overhead of the prefetch thread has an additional execution time overhead from the prefetch routine calls. Thus, if the task’s critical regions are sparse, then it will have a greater execution time overhead. Thus it can be minimized by laying out the critical regions in contiguous memory block.

One key disadvantage that the pull has over push is when a block belonging to a critical region does not reside in the source cache, push model can identify it at much lesser cost than our prefetch thread based pull model. Prefetch thread based pull model issues an unmodified load operation which will incur round trip to the source and even to the memory. Even with a specialized load that does not incur a memory request, a round trip to the remote cache is unavoidable. While hardware support of push model can identify the non-resident block by a lookup into the local cache and does not issue any wasteful push requests.

9. References

[1] A. Sarkar, F. Mueller, H. Ramaprasad, S. Mohan. Push-Assisted Migration of Real-Time Tasks in Multi-Core Processors. To appear in Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'09), Dublin, Ireland, June 19-20, 2009.
[2] M. Zhang and K. Asanovic. Victim migration: Dynamically adapting between private and shared cmp caches. TR 2005-064, MIT CSAIL,2005.
[3] Simultaneous Multithreading: Maximizing On-Chip Parallelism, D.M. Tullsen, S.J. Eggers, and H.M. Levy, In 22nd Annual International Symposium on Computer Architecture, June, 1995
[4] 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.
[5] Mälardalen benchmarksuite. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[6] Tilera processor family. http://www.tilera.com/products/processors.php.

Project URL : http://www4.ncsu.edu/~asarkar/CSC714/home.html

Submitted by: Abhik Sarkar
Unity ID: asarkar