Fast and Scalable Thread Migration for Multi-Core Architectures

Miguel Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
miguel.pereira.rodrigues@tecnico.ulisboa.pt

***Abstract*—With the breakdown of the Dennard Scaling and the consequent Dark Silicon advent, heterogeneous computing arised as a popular solution to tackle the associated power and energy constraints. However, by additionally allowing the migration of application threads to their most appropriate cores, performance gains and energy efficiency levels can be further increased. Unfortunately, the considerably large overheads imposed by typical software-based thread migration procedures only allow migrations at a coarse-grained level, limiting the effectiveness of using such techniques. Accordingly, this document proposes a fast and efficient hardware-based thread migration mechanism easily plugged-in into any core architecture. To minimize the thread migration overhead and latency, the proposed approach considers both soft- and hard-migration procedures, and adopts a conventional most recently used prediction scheme to identify cache blocks that should be migrated along with thread context. Experimental results show that the proposed scheme requires few hardware and energy resources, while allowing to attain migration latencies of 20-30 cycles (a high improvement regarding state-of-the-art commercial solutions which require over 20.000 clock cycles) and to reduce post-migration overheads in up to 60%, making such mechanism particularly appropriate for exploiting short-lived application phases.**

1. Introduction

With the advent of many-core chips to allow the scaling of the processing performance, the consequent increase of the number of transistors within a single chip led to difficult challenges related to power and thermal issues. In particular, the advent of the Dark Silicon phenomenon introduced a growing gap between the number of transistors that fit in a chip and the number of transistors that can be simultaneously used, due to thermal and power constraints [9]. To overcome such an issue, several approaches have been proposed, including the selective dimming of some transistors, or the gating of processing components. Heterogeneous computing architectures pose an important role in this matter, by allowing distribution of the computation to the more energy efficient processors [18]. The use of heterogeneous GPP architectures has even made its way to industry, with ARM big.LITTLE [10] being the most widely known processor. However, to fully harness the advantages of heterogeneous architectures, application threads need to be frequently and efficiently migrated between heterogeneous cores, according to the requirements of each individual processing phase.

Typical thread migration mechanisms rely on software based approaches to transfer the thread context via shared memory [1], [10], [13], [17]. This can be done at OS time slice intervals or by using specific load and store instructions. However, while such approaches allow addressing thermal issues [19] and solving hardware fault tolerance [16] problems, they do not allow exploiting the high variability of fine grained thread phases (10^3 to 10^4 cycles, [15], [17]), thus significantly limiting the attainable performance and energy efficiency levels.

Faster migration mechanisms have been proposed at the cost of additional hardware [4], [15]. Besides the extra hardware, the main drawbacks of such implementations are the poor scalability and/or the implicit requirements of major microarchitectural modifications. Furthermore, decreasing migration latency does not linearly increase the frequency at which threads can be migrated, due to the increased sensibility to post-migration overheads.

Post-migration overheads arise from what is usually denoted by "cold resources" in the architecture, such as the loss of caches contents, branch predictor, etc. In fact, although the data from these structures does not necessarily need to be migrated along with the thread (it can be dinamically rebuilt after migration) it usually imposes a significant application slowdown [3], especially for short-lived threads.

To overcome such issues, a Fast and Scalable Thread Migration (FSTM) mechanism is herein proposed that not only allows low-latency thread migrations, but also reduces overheads imposed by cold cache resources. FSTM is suitable for homogeneous or heterogeneous multi-core architectures and can be easily plugged-in into many-core processor aggregates, composed of any arbitrary number and core types, hence being scalable and flexible.

2. System Overview

A thread migration can be roughly defined as moving a thread from one execution environment to another, such as transferring a thread between cores. Thread migration mechanisms typically consist in the following steps:

1) Generate a thread interruption;
2) Flush the pipeline and the cache write buffer;
3) Copy the thread context to memory;
4) Load the thread context from memory.

TABLE 1: Application slowdown due to cold cache effects. Clock frequency at 1GHz.

		Migration period (clock cycles)		
	Caches	2K	10K	50K
BackProp (3.2ms)	Warm	0.01	0.00	0.00
	Cold L1	0.24	0.07	0.04
	Cold L1+L2	2.64	0.50	0.32
Kmeans (1.7ms)	Warm	0.05	0.02	0.01
	Cold L1	1.21	0.28	0.07
	Cold L1+L2	9.60	1.78	0.28
LavaMD (4.3ms)	Warm	0.04	0.01	0.01
	Cold L1	0.42	0.16	0.08
	Cold L1+L2	4.70	0.67	0.30
LUD (8.2ms)	Warm	0.03	0.01	0.00
	Cold L1	2.16	0.44	0.09
	Cold L1+L2	**10.06**	6.99	0.37

The Operating System (OS) scheduler is responsible for generating the thread interruption, which informs the hardware to stop fetching new instructions. This results in flushing the processor pipeline, allowing to safely gather the correct thread state and context. Such thread state usually includes the general purpose register file and specific purpose registers (e.g. Program Counter). However, exchanging such context data naturally imposes time costs, which limits the resulting execution efficiency.

The main causes for such thread migration time overhead can be classified in direct and indirect effects. Thread migration direct effects include flushing the pipeline and transferring the thread context. Indirect effects arise from loss of state of caches, branch predictors and instruction queues, usually denoted as cold resources. Together, these effects cause a significant application slowdown when a thread is migrated to a different core.

Application slowdown in single-threaded applications increase with the migration frequency [6]. To experimentally illustrate such an issue, zero latency thread migrations were used to quantify the overhead introduced by cold cache effects as presented in Table 1. These results were obtained in gem5 simulator [2] using an ARM Out-of-Order processor and running the Rodinia benchmarks suit [5].

The obtained results show almost no performance degradation for warmed-up caches, where the observed degradation mainly comes from flushing the pipeline. However, for the highest migration frequency (corresponding to 2K clock cycles between thread migrations), cold L1 caches result in slowdowns ranging from 0.24x (in BackProp) to 2.16x (in LUD). The problem is even worse when cold L1+L2 caches are evaluated where the slowdown can exceed 10x.

The observed results show that not only a low latency migration mechanism is required for highly frequent thread migrations, but an efficient method for dealing with cold cache resources is also required. The herein proposed FSTM mechanism represents a Fast and Scalable Thread Migration to circumvent this problem. In the remaining of this section, the hardware structures sustaining such migration mechanism will be described.

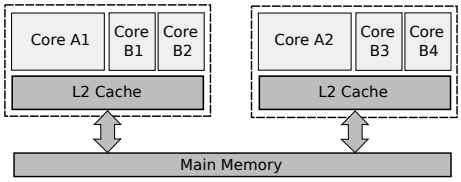


Figure 1: System with 2 cores of type A and 4 cores of type B grouped into 2 clusters. Each core has private L1 caches and shared L2 caches within a cluster.

2.1. System Baseline

The proposed structure targets efficient thread migrations on conventional homogeneous or heterogeneous processor aggregates equipped with a shared memory to ensure inter-core communication. Additionally, it assumes the cores to be grouped into clusters, in order to provide a better organization and enable heterogeneity at a cluster granularity. An example of such system is depicted in Figure 1.

Due to the increased heteogeneity, thread migrations either occur within a cluster (intra-cluster) or across cluster boundaries (inter-cluster). Intra-cluster migrations are assumed to be much more frequent than inter-cluster ones but the latter are still supported at the cost of additional latency (driven by the increased physical distance). It is also assumed the presence of a dynamic ISA translator when dealing with heterogeneous ISAs [7].

Additionally, the proposed system baseline assumes the presence of a thread scheduler to trigger the migrations. Although the integration of FSTM with the OS main scheduler is discussed in Section 2.3, other types of schedulers (hardware or software based) can be used. This happens because FSTM is implemented as a passive mechanism, simply receiving the migration commands and executing them in an efficient way.

2.2. Proposed Migration Mechanisms

The idea behind FSTM consists in streaming the thread contents through specific hardware communication channels, in order to avoid the pollution of shared caches with thread context data. Such hardware (depicted in Figure 2) includes Input and Output Buffers to interface the cores with a Thread Router (to ensure the thread transfer between the cores) and a Thread Storage, especially designed to temporarily store evicted threads, avoiding data cache pollution and allowing for faster context switches.

To further minimize the migration overheads, the herein proposed approach is not limited to conventional thread migration styles where the processor pipeline needs to be flushed before starting the actual migration procedure. Since this may be a limiting factor for applications with rapid phase changes (pipeline flushing can lead to considerable overheads, especially for out-of-order processors with long pipelines), the proposed approach considers initiating the migration of thread contents while the thread is still being executed. As a consequence, not only there is a considerable

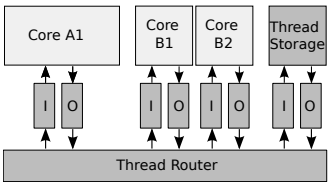


Figure 2: FSTM hardware structures connected to a cluster with 1 core of type A and 2 cores of type B. Input (I) and Output (O) Buffers used as interface.

reduction of the thread migration latency, but it also allows anticipating the scheduling decisions (whenever possible).

To further detail the operation of the proposed FSTM mechanism, Figure 3 depicts the thread migration procedure from core I to core II. When signaled for migration (time t_0 in Figure 3), FSTM starts transferring the thread state to the Output Buffer, including not only the general and special purpose registers, but also the state of other thread resources, such as cache contents. To minimize the amount of resources to be transferred, the proposed FSTM mechanism also integrates a set of snooping mechanisms (detailed in section 3) that are especially devised to keep track of the registers and cache contents that are actually used by the running thread.

When triggering migrations, the scheduler must detail the type of migration, which can be soft or hard. Soft migrations indicate FSTM to transfer thread state while executing the thread, overlapping thread migration and execution. On the other hand, hard migrations force the fetching unit to stop fetching new instructions, thus flushing the pipeline. Naturally, hard migrations must always follow soft migrations in order to stop the thread execution. In this example it is assumed a soft migration (triggered at time t_0) followed by a hard migration (triggered at time t_2). After flushing the pipeline and transferring all thread contents (time t_3), core I becomes idle and is ready to load a new thread.

In the meanwhile (time t_1), the first contents of thread A arrive at the Input Buffer of the targeted core II (remember that thread contents are streamed instead of bulk transferred), after being properly routed by the Thread Router. Such structure interconnects the Input and Output Buffers by using communication channels that may be pipelined (if necessary) to maintain high clock frequencies.

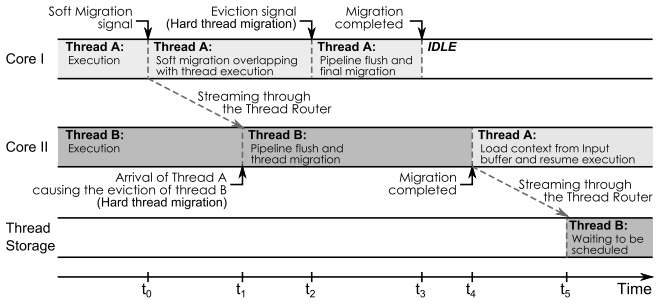


Figure 3: Temporal diagram of thread migration.

The arrival of thread A to the targeted Input Buffer will cause the eviction of the running thread B whenever the just arrived thread A has a higher priority. In the depicted example, this case is assumed and a new hard migration (thread B) is triggered via a thread eviction signal. However, should thread A had a lower priority than thread B, the latter would continue executing until completion or being interrupted. In either case, thread A needs to wait at the Input Buffer for the core to become free. As soon as the core is free (time t_4), thread A can be loaded and resume its execution. After a while (time t_5), thread B arrives to the Thread Storage, where it will wait to be scheduled onto a new core thus being migrated to it and resuming execution. Naturally, depending on the scheduler decisions, the eviction of thread B from core II, could also originate its direct migration to another core (e.g., core I), instead of being moved to the Thread Storage.

2.3. Integration with the OS Scheduler

To control the threads execution and to trigger thread migration commands, a tight relation between FSTM and the OS scheduler must exist. Although different scheduling strategies (master) may be employed, especially considering that FSTM is a passive mechanism, the following description will focus on its integration with the Completely Fair Scheduler (CFS), as used by modern Linux kernels.

In particular, CFS aims at ensuring fairness between running applications while also guaranteeing interactive performance. This is attained by creating a time ordered red-black tree of the active (runnable) tasks for each core, which is then used to schedule (prioritize) threads execution. Although such mechanism allows ensuring fairness at a coarse-grained level (miliseconds), it leads to unsustainable overheads when scheduling threads with rapid phase changes and difficults an efficient exploitation of heterogeneous architectures.

To overcome such an issue, the presented proposal envisions the usage of auxiliary hardware schedulers underneath the software layer (see Figure 4), which are responsible for managing threads at an intra-cluster level, while the OS scheduler performs the scheduling at an inter-cluster level.

Although the description of such schedulers is out of the scope of this manuscript, its general operation is depicted in Figure 4. Accordingly, while the software layers (OS) are responsible for assigning tasks to clusters and for managing

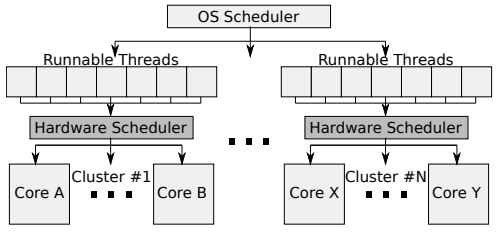
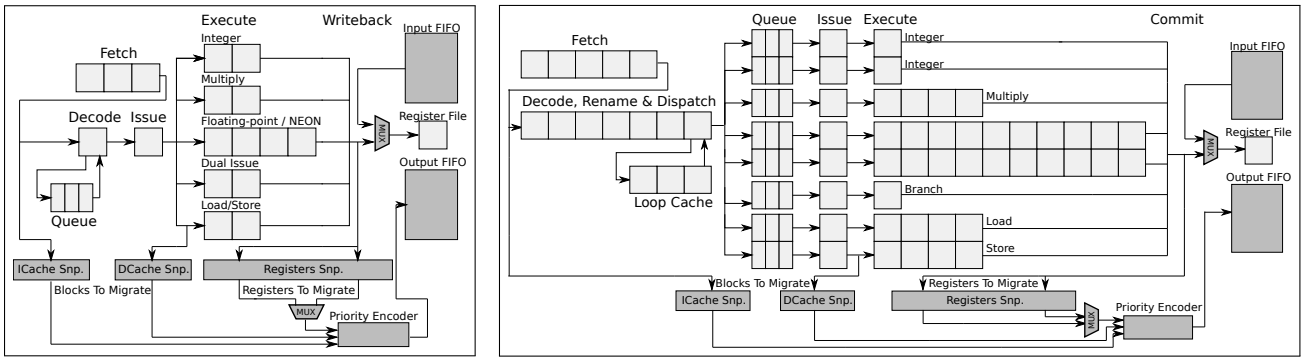


Figure 4: Multi-layer scheduling using auxiliary hardware schedulers within clusters.



(a) In-order core.

(b) Out-of-order core.

Figure 5: Integration of the FSTM modules in a heterogeneous processor.

exceptions (e.g., triggered by page faults), the per-cluster hardware schedulers are responsible for implementing the CFS decisions by assigning tasks to cores (according to specific policies related to the core's architectures, computational resources, thermal/power restrictions, etc.), and by also collecting important run-time information, in order to identify phase changes and to trigger migrations between heterogeneous cores.

3. FSTM Implementation

Contrasting to what happens with conventional migration approaches, the efficiency of the proposed mechanism is maximized by selectively transferring the subset of registers or cached contents that are actually being used by the thread under consideration. The following subsections describe these mechanisms, together with the communication infrastructure that supports such transfer.

3.1. Selective migration of thread context

To keep track and selectively transfer the subset of registers and cache lines being used by the thread, FSTM requires a snooping mechanisms for registers and cache contents. Figure 5 depicts an application example where an in-order processor and an out-of-order processor are aggregated in an heterogeneous multi-core. Cache snooping is implemented by `ICache Snp` and `DCache Snp` modules (see gray boxes), while `Register Snp` module selectively selects the set of general- and special-purpose registers in use. Then, a `Priority Encoder` is used to order the transfer of the thread context to the Output FIFO.

Furthermore, to allow the overlapping of the thread migration procedure with the thread execution, the modified logical registers are marked in an auxiliary migration scoreboard table (composed of two `dirty` bits per logical register), either at the writeback stage (in-order processor) or at the commit stage (out-of-order processor) of the pipeline.

However, although this strategy allows reducing the thread migration latency, it may give rise to data hazards at the destination core, due to the writing of new contents to a register which has already been migrated. To overcome this issue and also to avoid constant migration of the same register, which may be modified several times before thread eviction, the following strategy is adopted: i) the first `dirty` bit is used to keep track of the registers that have been modified, and is used to control the registers that are transferred upon reception of the thread migration signal; ii) the second `dirty` bit is used to mark registers that have been modified after thread migration starts, and is used only after pipeline flushing, in order to migrate the remaining thread context.

For the migration of the cache contents, which would lead to substantial migration overheads, a similar strategy is herein adopted. However, instead of transferring the whole cache contents, a *Most Recently Used* prediction scheme is used. Although other more sophisticated policies could be used, an important compromise must be reached between the migration of cache contents (which increase migration latency) and the mitigation of the thread slowdown imposed from uninitialized cache contents. The adopted strategy consists in using a `valid` bit that indicates which cache blocks (from either Instruction or Data caches) have been touched in a time window preceding the current clock cycle, taking advantage of temporal locality. Upon migration, the selected cache blocks are streamed alongside with the processor registers, in order to reduce the cold cache effects in the migration. Naturally, enlarging such time window will increase the number of migrated blocks, which may (or may not) be beneficial considering the cost and overhead of thread migration and the accuracy of the prediction.

The actual migration of the thread context, as marked in the migration scoreboard table, can be made by any order. However, the proposed approach adopts a priority encoder to give precedence to the general- and special-purpose registers. In particular, since the cache contents can be dynamically rebuilt at the destination core, this approach ensures that the thread can start executing at the destination core as soon as its architectural state is received. Additionally, the priority encoder performs a second important role in thread migration, by creating labels that identify which contents are being transferred, which are used at thread arrival to write the data in the appropriate locations.

3.2. Migration Channels

Figures 5a and 5b show how the Input and Output Buffers are connected to the cores, providing a mean to store and load the threads. Such buffers are implemented by using a dual clocked FIFO, to support per-core Dynamic Voltage Frequency Scaling (DVFS).

Figure 6 shows how the Thread Router multiplexes the communication channels (wires) between Output and Input FIFOs, routing the threads according to migration parameters (thread origin and destination). Hence, the Thread Router is responsible for checking the existence of threads awaiting at the Output Buffers and, whenever required, migrating them accordingly. Such structure is mainly implemented with multiplexers to choose from which Output Buffer is a thread migrating from, sending the migrating thread to the correct destination. Furthermore, a mutex is made available to manage concurrent requests. In particular, the Thread Router locks such mutex to the first migrating thread arising from the Output Buffers, solving conflicts with a round-robin priority scheme.

Just as an ordinary core, the Thread Storage is connected to the routing infrastructure through an Input and Output Buffer. Despite the several alternatives that could be adopted to implement this element, a direct mapped memory was herein adopted, in which the threads have a specific location to be stored, according to their ID, which is dynamically assigned by FSTM (not to be confused with the ID used by the OS). Such approach eases the management of the threads locations when triggering a migration from the Thread Storage. During the course of such migration, an auxiliary hardware vector indicating which threads exist and are valid inform if such migration is valid (e.g. trying to migrate a thread from the Thread Storage when the thread is not there is not possible). If such migration is allowed, thread contents are streamed through the Output Buffer and the migration will elapse in a similar way as when threads migrate from processing cores.

When an arriving thread (at the core Input Buffer) should be loaded, a convenient set of multiplexers is used to choose the origin of data being stored at the register file: either from the Input Buffer or from the core writeback/commit stages (see Figure 5). Hence, such multiplexers forward the thread contents from the core Input Buffer into the internal registers and caches, according to the labels that are migrated alongside with the registers and cache contents. As soon as the thread has finished loading all its registers, it may resume its execution, independently on the initialization of other resources, which may be loaded from outer cache levels or from the Input Buffer, whichever is faster.

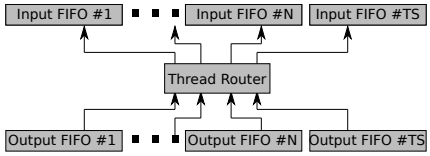


Figure 6: Interfacing FIFOs between cores and Thread Storage (TS), managed by Thread Router.

TABLE 2: Parameters of the used CPU model.

Parameter	L1	L2
Size	32 kB	512 kB
Cache Line Size	64 B	
Associativity	4	8
Latency	4	20
MSHRs	4	11
Write Buffers	16	9

Clock Freq.
1 GHz

Prefetcher
None

3.3. Communication Protocol

The Wormhole Switching protocol was chosen to stream threads across cores, as it introduces very small overhead. It only adds a header (with destination) and a tail (signaling the end of thread) to the data packet corresponding to the thread being migrated. Such tail is necessary because the thread packet size is not fixed. Furthermore, a thread ID must be sent alongside with the thread contents, to provide the necessary information to the Thread Storage. Such ID is dynamically assigned and identifies the thread in FSTM channels (not to be confused with the thread ID assigned by the Operating System). This protocol specific information is added at the beginning of a migration, before sending the thread through the Output Buffers.

4. Experimental Evaluation

In order to evaluate the proposed FSTM mechanism, the widely adopted gem5 simulator [2] was used in its *Syscall Emulation* mode, to model a real multi-core processor. In particular, the considered processing structure mimics a dual-core aggregate composed by two out-of-order A15 ARM cores. The adoption of such homogeneous aggregate arises from our interest in quantifying the overall processing gains that are obtained by using the proposed efficient migration scheme. For such purpose, this simulation environment ensures not only an accurate emulation of all these processors, but also a rigorous modeling and evaluation of all elements that support the migration procedure. Private L1 and shared L2 caches of these cores (see Table 2) were used by default, unless when stated otherwise. The considered parameterization considers a migration latency of 1 clock cycle for each 64-bits block, plus 7 cycles for filling the migration pipeline. The same clock frequency (1GHz) is assumed for all cores, as well as for the migration infrastructure. To emulate the interference between threads, as well as the consequent thread migrations, the adopted simulation environment also comprised appropriate mechanisms to evaluate the resulting effects of cold cache resources, by invalidating the core destination private cache prior to migration.

Four distinct and representative benchmarks from Rodinia suit [5] were used to evaluate the proposed scheme, namely BackProp, Kmeans, LavaMD and LUD, running in a single thread. The migrations were manually triggered in fixed size intervals, ranging from 2K to 250K clock cycles,

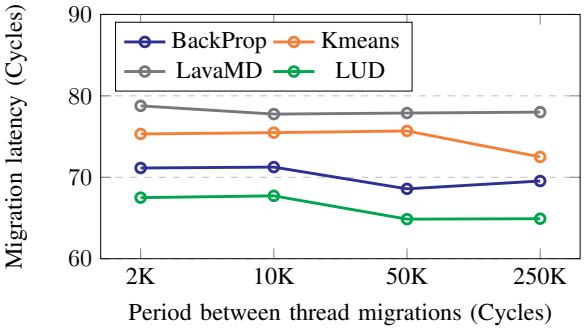


Figure 7: Migration latency using FSTM hard migrations.

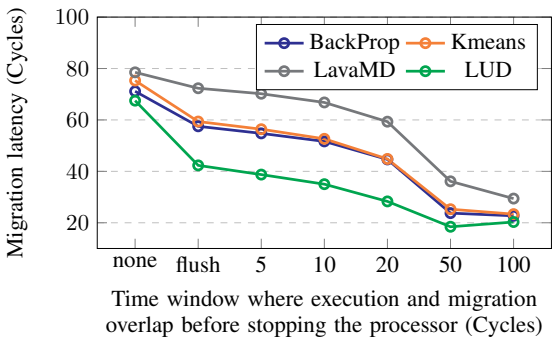


Figure 8: Migration latency using FSTM soft migrations. Migration period of 2K cycles.

in order to understand how thread migration overheads varies across different migration periods.

In order to obtain energy results, the McPAT simulator [14] was used. Since McPAT was designed to measure power of several computer architectures types by making use of different templates, FSTM was conceptually divided in two parts: the migration channel and the Input/Output buffers. Then, FSTM was emulated in McPAT by replacing the migration channel with an additional shared interconnect memory bus and by integrating the Input/Output buffers into the L1 caches.

4.1. Time Overhead

Time overheads are of major importance when considering thread migrations, due to the imposed limitations in exploiting short-lived application phases to migrate between cores. Thus, in order to increase performance and energy-efficiency, it is necessary to consider if the cost of migrating the thread is worth the gains achieved by doing so. Accordingly, this section focus on studying the overheads of hard thread migration (see Section 2.2) with the proposed FSTM mechanism.

As detailed in Section 2, migration overheads arise from both migration latency and post-migration overheads caused by cold resources. In order to quantify the migration cost, the inherent migration latencies were measured and depicted in Figure 7. The obtained results show that migration latency remains almost constant across different migration periods. That is explained by the fact that migration latencies using FSTM exclusively depend on the amount of thread contents to transfer, which are mostly independent of thread migration frequency.

Furthermore, these results also show that these migration latencies are no greater than 100 clock cycles, which represents an improvement of more than two orders of magnitude when compared with ARM big.LITTLE that has a migration latency of about 20,000 clock cycles [10] when switching between A7 and A15 cores. Such is possible because FSTM is a hardware based mechanism, whereas big.LITTLE depends on the OS software to implement this. Furthermore, by relying on the implemented snooping mechanisms to keep track of which logical registers are effectively being used, the proposed FSTM mechanism avoids transferring the complete physical register file, which would include over 650 registers (90 integer, 256 float and 320 miscellaneous/control physical registers [2], [8]) in ARM Cortex-A15. In particular, the proposed snooping technique allows for a significant reduction of up to 90% of the amount of registers being transferred, which effectively provides a large reduction in the migration latencies.

Additionally, FSTM allows to overlap migration and execution (soft migrations) to further decrease migration latency. In order to understand how the performance gains vary with the amount of time where migration and execution overlap, results were gathered in Figure 8 considering a migration period of 2K clock cycles.

The results obtained show that the usage of soft migrations is more effective as the time window where execution and migration overlap enlarges, being able to reduce migration latency by 3x when compared with hard migrations. However, even when the overlap only occurs during the pipeline flush, significant reduction can be achieved for applications such as LUD, where flushing the pipeline consumes a lot of time (40 cycles on average versus 20 on other benchmarks).

Nevertheless, a trade-off must be considered when deciding the amount of time where execution and migration overlap: while a larger time window allows more registers to be speculatively transferred, it also increases the chance of such registers being updated in the meanwhile. In that case, such dirty registers need to be re-transferred after pipeline flush, thus increasing migration latency. Such event can be seen in Figure 8, where varying the time window from 50 to 100 cycles does not bring much performance benefits.

Furthermore, to evaluate the provided gains in terms of migration overheads, the cache contents corresponding to each thread were transferred via the FSTM communication channels. The selection of such contents considered the adopted *Most Recently Used* prediction scheme, where cache blocks are marked for transfer if they have been touched during a specific time window preceding the migration.

The migration overhead was measured as the difference

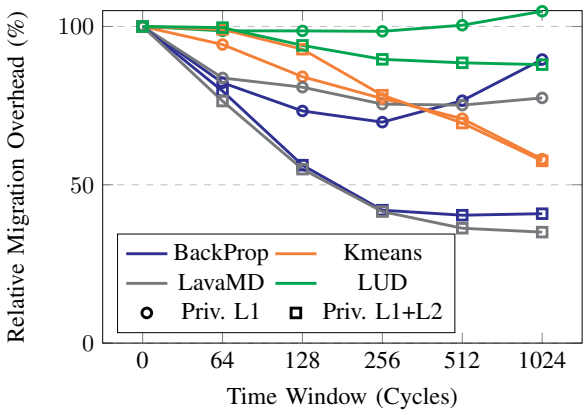


Figure 9: Migration overhead reduction due to initialization of cache contents (migration period of 2K cycles).

between the observed execution times with and without cache and registers migrations. Furthermore, the size of the time window was varied to understand its influence. Then, the obtained results were normalized according to equation (1) and depicted in Figure 9:

$$Overhead = \frac{(T_{MigCache} - T_{NoMig})}{(T_{MigReg} - T_{NoMig})} \quad (1)$$

where:

$T_{MigCache}$: Execution time with thread, cache and registers migration
T_{MigReg} : Execution time with thread and registers migration (only)
T_{NoMig} : Execution time without thread migration

On a side note, it is worth noting that the overhead reduction represented in Figure 9 represents a worst-case situation, since it assumes that all marked cache blocks are effectively migrated before the thread resumes execution. In fact, the thread may resume execution as soon as the architectural thread state is transferred, where cache contents are subsequently migrated while the thread is already executing, with data being retrieved from either the outer level caches or from FSTM channels, whichever is faster.

Despite this limitation, the obtained results show that the overhead due to cold cache effects can be effectively reduced, especially when using private L1+L2 caches. Moreover, since FSTM does not rely on any existent memory sub-system to migrate threads, its effectiveness increases as threads are more sensitive to cold resources (due to a higher cache-miss penalty).

Furthermore, it was also observed that the time window size plays an important role in this matter, since its increase causes more cache blocks to be transferred. In fact, whenever useful cache blocks are transferred, the cache hit rate will increase, thus reducing the overhead. On the contrary, transferring useless data will intensify post-migration overheads, due to the additional migration latency of transferring such blocks.

Kmeans and LUD benchmarks are two distinct examples of such events. Kmeans shows continuous improvements while increasing the time window size – even at a time window of 1024 cycles, 85% of the transferred cache blocks are used in the near future. On the other hand, LUD shows less improvements, since the accuracy of the adopted simplistic prediction is very poor – as low as 35% in the worst case, having even a worst performance with larger time windows, when compared to not transferring any cache contents. Furthermore, in the case of the BackProp benchmark using shared L2 caches, it is possible to observe that the overhead has a minimum for a time window size of 256 cycles. Such observation is consistent with the previously described tradeoff between migrating more or less blocks. The same effect is not so easily seen while using private L1+L2 caches, due to the higher post-migration overheads.

In summary, the obtained time overhead results can be summarized in three key conclusions:

1) FSTM hard migrations decrease migration latency from 20,000 cycles to 60-80 cycles, further reduced to about 20 cycles when using soft migrations;
2) FSTM with cache contents migration reduces cold cache effects, thus decreasing migration overheads.
3) FSTM does not rely on any existing memory subsystem, making it more effective as threads are more sensitive to cold cache effects.

4.2. Hardware Resources

To quantify the added hardware resources required by the proposed FSTM, all needed modules were implemented and aggregated to a cluster of MB-Lite cores [12], prototyped in a Xilinx Virtex-7 (XC7VX485T) FPGA. Convenient synthesis and post-place and route procedures were performed using Xilinx ISE 13.4 software. In accordance, the implemented baseline system consists in a variable number (N) of MB-Lite cores, each with private instruction and data cache memories, and accessing a shared memory, made available to provide inter-core communication. This system was described using generic VHDL code to ease system configuration and to provide consistency across implementations with varying parameters.

The required hardware resources were measured with and without the FSTM modules and by varying the number of cores (2, 4, 8 and 16). The results obtained with different core count were averaged since they presented an area per core almost identical in every case.

Figure 10 presents the obtained measures in form of Slice LUTs, Slice Registers and RAM blocks. It is worth noting that the usage of the multiplier, barrel shifter or FPUs does not change the number of occupied Slice Registers and RAM blocks (RAMB36E1), since they are implemented as pure combinatorial blocks. The obtained results show that, in the worst case of a simple core supporting only logic and integer 32-bit add operations, the FSTM mechanisms occupy at most 35% of the used Slice Registers and 44% of the used RAM blocks, mainly due to the usage of FIFOs to

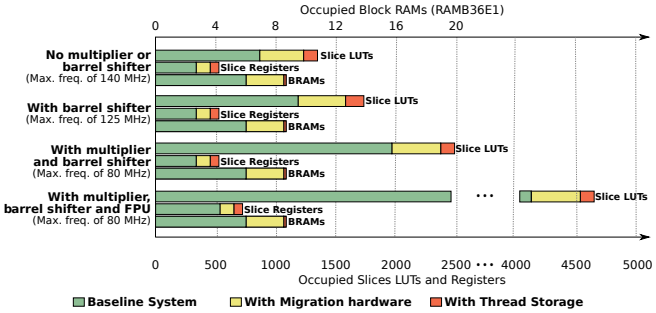


Figure 10: Required hardware resources per core type. The proposed FSTM mechanisms had no impact on the maximum operating frequency.

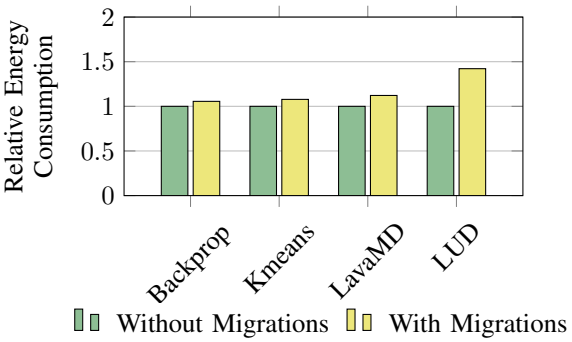


Figure 11: Dynamic energy consumption increase of the processor due to frequent migrations (migration period of 2K cycles). Only the core which is executing is considered.

implement the Input and Output Buffers, and 20% to 35% of the Slice LUTs in use.

Naturally, this overhead is less notorious as the core complexity increases, as the inclusion of an FPU demonstrates (see Figure 10). In fact, since MB-Lite cores are very simple and small, the measured overheads represent a quasi-worst case scenario. In particular, such overheads would be significantly less representative when considering typical super-pipelined and out-of-order cores supporting integer, floating point and vector arithmetic, such as the out-of-order ARM core simulated in gem5.

Moreover, the impact of the aggregation of FSTM in the maximum allowed clock frequency was also measured, by evaluating the maximum clock frequencies before adding FSTM under the four cases depicted in Figure 10. After adding the FSTM modules, it was confirmed that their inclusion does not reduce the clock frequency. This is mainly due to the configurable pipeline stages that were specially added to the FSTM paths to prevent such degradation, allowing to maintain the original clock frequencies even when using 16 cores in the system. Nonetheless, another possible option would be to use a different clock frequency domain for the FSTM communication channels, which could be adapted in real-time using DVFS techniques according to the system run-time requirements.

4.3. Energy Costs

To quantify the energy overhead FSTM represents, the gem5 [2] and McPAT [14] simulators were jointly used. Additionally, all measurements are herein presented in energy terms instead of power in order to fairly compare benchmarks (note that as there are more migrations during an application, more time it takes to finish it but the work performed is nearly the same).

Thus, in order to quantify the energy overhead caused by thread migrations, the dynamic energy consumption with and without migrations was measured considering a 2K cycles migration period (see Figure 11 and Table 3). The results obtained show that the energy consumption increases with the execution time and that benchmarks suffering more from cold resources (such as LUD), have a bigger increase in energy consumption when migrating. Thus, these results suggest that a major part of the energy overhead of migrations is caused by post-migration overheads and not by FSTM additional hardware.

Such is confirmed by the results presented in Table 3, showing that FSTM represents less than 2% of the energy consumed by the processor. Moreover, FSTM represents only 10% of the energy overhead (measured as the difference of energy consumption between migrating and non-migrating benchmarks) on average, confirming that cold resources are responsible for most of energy overhead.

Finally, the results show that the energy leakage of FSTM represents only a residual amount of the consumed energy (1% of total leakage and 0,1% of total energy). However, for applications not using migrations it is recommended to power-gate FSTM modules or use DVFS mechanisms to reduce such energy leakage to a minimum.

4.4. Related Work

To the best of the author's knowledge, the proposed FSTM mechanism is the first that implements a complete hardware-based solution for an efficient thread migration mechanism, where cache contents are streamed alongside with the register values via specific communication channels. Nevertheless, other related work concerning cold resources effects have been done by [3], [6], [11], [15] and [17].

Constantinou et al. [6] studied the performance implications of single thread migration, by showing the importance of warming-up resources to reduce post-migration overheads. They propose warming-up resources (caches and branch predictor) prior to migration, which can only be done if the destination core is idle. FSTM differs from this work by migrating thread contents alongside with the thread, being able to reduce cold cache effects even without an a priori training. This is particularly useful in multi-threaded environments, where cores are unlikely to be idle.

Brown et al. [3] study the usage of several cache working set predictors and propose a prefetching of such contents at destination core. Information indicating which contents to

TABLE 3: Energy consumption increase due to cold cache effects. Clock frequency at 1GHz and migration period of 2K cycles.

		Execution Time (ms)	Processor Energy (μJ)		FSTM Energy (μJ)	
			Dynamic	Leakage	Dynamic	Leakage
BackProp	Without Migrations	3.2	916	133	-	1.32
	With Migrations	4.0	967	165	9.86	1.64
Kmeans	Without Migrations	1.7	912	70	-	0.70
	With Migrations	3.7	1093	155	9.87	1.54
LavaMD	Without Migrations	4.3	1503	179	-	1.77
	With Migrations	6.1	1686	256	17.07	2.54
LUD	Without Migrations	8.2	3002	342	-	3.39
	With Migrations	25.9	4268	1081	60.76	10.71

prefetch is migrated with the thread and prefetching can be done in parallel with thread execution. FSTM differs from this work by moving cache contents proactively with the thread, not depending on the memory sub-system. Moreover, although this is not exploited in this work, FSTM could easily be extended to migrate the state of the prefetch history table, thus contributing to a further reduction in cold cache effects.

Lukefahr et al. [15] brings the concept of heterogeneity within a single composite core, by integrating an In-Order and an OOO processing structure in the same core, both sharing the same cache and branch history table. Accordingly, such composite cores allow intra-core thread migrations, solving the problem of cold resources, where only the architectural thread state must be transferred while the core is executing. FSTM differs from this work by being easily plugged into several systems, not requiring any major micro-architectural modifications.

Similarly, Rangan et al. [17] and Gutierrez et al. [11] evaluate post-migration overheads when sharing caches across cores. While such resource sharing allows to avoid cold cache effects, it limits the available architectural system scalability. Furthermore, average cache accesses are expected to be slower due to sharing the same memory cache capacity by several cores. FSTM differs from this work by easily scaling with core count and type, independently of the memory sub-system.

5. Conclusion

Heterogeneous computing has recently emerged as a viable way to overcome the power, energy and thermal constraints of modern computing systems, by providing the means to schedule the execution of the application phases to different processing cores. However, typical software-based approaches lead to migration latencies that range from thousands to millions of clock cycles. This seriously constraints the exploitation of fine-grained application phase changes, limiting the attained performance and energy efficiency levels.

Accordingly, this paper proposes a new Fast and Scalable Thread Migration (FSTM) mechanism that makes use of specialized hardware to significantly reduce thread migration latency and overhead. In particular, it relies on a scoreboard table in order to detect, in run-time, the actual thread context and therefore limit the amount of resources that would otherwise be transferred. As a result, an up to 90% reduction in thread context transfer is observed, which allows keeping the thread migration latency below 80 clock cycles for the considered benchmarks, further reduced to about 20 cycles when overlapping thread migration and execution. Furthermore, the proposed approach also adopts a conventional "most recently used" prediction scheme to identify the cache blocks that should be migrated, which allows attaining up to 60% reduction in post-migration overheads.

To evaluate the inherent overhead in terms of energy and hardware resources, the proposed FSTM mechanism was implemented in gem5 and McPAT simulators and also prototyped in a Xilinx Virtex-7 FPGA to be compared with the hardware resources of an MB-lite core. Experimental results show that, even when compared with such a small processor, there is a limited increase in the used resources and no observable impact in the operating frequency.

References

[1] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proc. 3rd Conf. on Computing Frontiers*. ACM, 2006, pp. 29–40.

[2] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[3] J. A. Brown, L. Porter, and D. M. Tullsen, "Fast thread migration via cache working set prediction," in *Proc. Int. Symp. on High Performance Computer Architecture*. IEEE, 2011, pp. 193–204.

[4] J. A. Brown and D. M. Tullsen, "The shared-thread multiprocessor," in *Proc. 22nd Int. Conf. on Supercomputing*. ACM, 2008, pp. 73–82.

[5] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. Int. Symp. Workload Characterization*. IEEE, 2009, pp. 44–54.

[6] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec, "Performance implications of single thread migration on a chip multicore," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 80–91, 2005.

[7] M. DeVuyst, A. Venkat, and D. M. Tullsen, "Execution migration in a heterogeneous-ISA chip multiprocessor," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 261–272, 2012.

[8] F. Endo, D. Couroussé, and H.-P. Charles, "Micro-architectural simulation of in-order and out-of-order arm microprocessors with gem5," in *Proc. Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation*. IEEE, 2014, pp. 266–273.

[9] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proc. 38th Int. Symp. Computer Architecture*. IEEE, 2011, pp. 365–376.

[10] P. Greenhalgh, "big.LITTLE processing with ARM Cortex-A15 & Cortex-A7," *ARM White paper*, 2011.

[11] A. Gutierrez, R. G. Dreslinski, and T. Mudge, "Evaluating private vs. shared last-level caches for energy efficiency in asymmetric multi-cores," in *Proc. Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation*. IEEE, 2014, pp. 191–198.

[12] T. Kranenburg and R. Van Leuken, "MB-LITE: A robust, light-weight soft-core implementation of the microblaze architecture," in *Proc. Conf. Design, Automation and Test in Europe*, 2010, pp. 997–1000.

[13] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *Proc. 36th IEEE/ACM Int. Symp. Microarchitecture*. IEEE, 2003, pp. 81–92.

[14] S. Li *et al.*, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. 42nd Int. Symp. on Microarchitecture*. IEEE, 2009, pp. 469–480.

[15] A. Lukefahr *et al.*, "Composite cores: Pushing heterogeneity into a core," in *Proc. 45th IEEE/ACM Int. Symp. Microarchitecture*. IEEE, 2012, pp. 317–328.

[16] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 93–104.

[17] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread motion: fine-grained power management for multi-core systems," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 302–313, 2009.

[18] M. B. Taylor, "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse," in *Proc. 49th Design Automation Conference*. ACM, 2012, pp. 1131–1136.

[19] I. Yeo, C. C. Liu, and E. J. Kim, "Predictive dynamic thermal management for multicore systems," in *Proc. 45th Design Automation Conference*. ACM, 2008, pp. 734–739.