Maximizing Multithreaded Multicore Architectures through Thread Migrations

Carmelo Acosta †, Francisco J. Cazorla †⋆, Oliverio J. Santana †, Ayose Falcón ‡
Alex Ramirez †⋆, Mateo Valero †⋆

† Departament d'Arquitectura de Computadors
Universitat Politecnica de Catalunya
Barcelona, Spain
{cacosta,fcazorla,osantana,aramirez,mateo}@ac.upc.edu

‡ Barcelona Research Office
HP Labs
ayose.falcon@hp.com

⋆ Barcelona
Supercomputing
Center

Abstract

Heterogeneity in general-purpose workloads often end up in non optimal per-thread hardware resource usage. The current trend towards multicore architectures, containing several multithreaded cores, increases the need of a complexity-effective way to expose the heterogeneity in general-purpose workloads to the underlying hardware, in order to obtain all the potential performance of these architectures.

In this paper we present the Heterogeneity-Aware Dynamic Thread Migrator (hDTM), a novel complexity-effective hardware mechanism that exposes the heterogeneity in software to the hardware, also enabling the hardware to react to the dynamic behavior variations in the running applications. By means of core-to-core thread migrations, the hDTM mechanism strives to perform the desired behavior transparently to the Operating System.

As an example of the general-purpose hDTM concept presented in this paper, we describe a naive hDTM implementation for a Power5-like processor and provide results on the benefits of the proposed mechanism. Our results indicate that even this simple hDTM implementation is able to get close to hDTM's goal, not only avoiding losses due to bad thread-to-core assignments (up to a 25%) but also going beyond the best static thread-to-core assignment upper limit.

1 Introduction

The recent advances in process technology are resulting in an increasing number of transistors on chip. As the number of transistors on chip dramatically increases, the issue of how to employ so many transistors in a complexity-effective way becomes even more important. In this context, clustering techniques [14, 4, 16] have proved to be an efficient solution to design high-performance processors that reduce area, energy consumption, chip temperature, and verification cost requirements with respect to traditional monolithical designs.

There are several proposals in the prior work [6, 11, 1, 12] that propose clustering techniques. As we apply aggressive clustering on a processor design, we cover the path between Simultaneous Multithreading Processors (SMT) [24, 25, 26], which share all the available resources between all the existing threads, and Chip Multiprocessors (CMP) [13, 7], which have separate processing elements that just share the second level cache. Recently, the trend has lead to an hybrid approach: multithreaded-multicore architectures, which has been reflected in industry with the appearance of SMT [15] and multithreaded-multicore [9] processors.

However, clustered SMT processors may not be enough to obtain complexity-effective processor designs. A general-purpose clustered processor statically partitions the available resources into equal clusters during the design stage. The problem is that such a general-purpose clustered processor is designed to execute a wide range of different applications, and thus it is not tuned to optimally execute any particular program. Therefore, it may not optimally cover the combined requirements of applications with different resource requirements. Furthermore, differences in resource requirements are present not only when comparing different applications but also within a single application, which behavior varies as the execution flows.

In order to avoid these inappropriate per-thread hardware resource assingment, some recent work [11, 1, 10] consciously incorporate this heterogeneity-awareness into the hardware itself. Among them, the Heterogeneous Multicore Architecture [10] is the only one that mirrors this software heterogeneity into the hardware itself, while dynamically maintaining an appropriate software-hardware matching. Instead

of partitioning the processor resources into equal clusters, the heterogeneous multicore architecture partitions the resources into clusters having different sizes. In this way, the architecture is able to assign each executing application to the cluster that is more suited to fulfill the application requirements and, at the same time, maximizing the overall processor performance, just suffering from an additional cost in terms of sampling phases. Current processor designs, such as the Power5 processor [9], involve limited heterogeneity-awareness. The processor, comprised of homogeneous cores, is aware that there are some specific and pathologic cases that do not benefit from SMT, and thus it supports the single-threaded (ST) execution mode, implementing seven different thread-priority ST execution modes. However, no further resource reassignment is made as the execution flows and workload requirements vary.

In this paper, we propose the Heterogeneity-Aware Dynamic Thread Migrator (hDTM), a novel complexity-effective hardware mechanism that exposes the heterogeneity present in software to the hardware. An approach based on fixed-time intervals avoids the need of additional and costly sampling phases. Besides exposing the heterogeneity, our mechanism gives the hardware full control to react to dynamic variations in running application behavior in multithreaded-multicore architectures. The hDTM is a general-purpose mechanism, valid for any multithreaded-multicore architecture, regardless the specific core characteristics such as heterogeneous or clustered cores. In this paper, we present both the hDTM concept and a specific hDTM implementation, including results that show its effectiveness.

2 The hDTM mechanism

The inherent heterogeneity in typical application behaviour [17, 20] yields workloads with requirements difficult to be appropriately served by a static hardware. The consequences of inappropriate software to hardware resource allocation are even more severe in multicore architectures, since hardware has been statically partitioned into cores. The Heterogeneity-Aware Dynamic Thread Migrator (hDTM) exposes the heterogeneity in the running software to the underlying multicore hardware. By making the hardware heterogeneity-aware and giving it full control over the on-chip resource allocation, it is possible to improve the hardware response to a highly heterogeneous execution workload.

In order to make the hardware heterogeneity-aware, it should be able to detect this heterogeneity using dynamic information. As stated in [19], we think that a single and low-complex to track metric is required. This metric should reflect accurately the phase varying behavior in running applications. However, we do believe that metrics independent of any individual architecture will not allow a specific piece of hardware to optimally react to heterogeneous software behaviour. Moreover, it is highly probable that our design constraints lead us to a multipurpose heterogeneity-awareness scenario, where the hardware must be aware of multiple metrics (at least one per heterogeneity source). As an example, to make a processor aware of the heterogeneity in both performance and temperature will require at least two metrics, one per heterogeneity source. In this sense, the hDTM implementation presented in this paper is single purpose, that is, it is strictly aimed to improve the total processor throughput. Multipurpose implementations are out of the scope of this paper and postponed to future work.

The hDTM mechanism is depicted in Figure 1. It is implemented as a piece of hardware with $T \times N+1$ inputs and T outputs, where T represents the number of running threads and N the number of tracked metrics. Notice that N will coincide with the number of heterogeneity sources whenever a single metric per source could be found. In the hDTM implementation presented in this paper, a single metric is used to track heterogeneity in throughput, and thus $N=1$, while T ranges from 2 to 4. The additional input (indicated as $+1$ in Figure 1) indicates the number of running threads. The hDTM first detects and classifies the heterogeneity in each of the sources it has been designed for. This heterogeneity is represented in terms of program phases, with specific program phases per each heterogeneity source.

In the hDTM implementation presented in this paper, the heterogeneity in throughput is represented by running threads passing from high-ILP (ILP) to memory-bounded (MEM) phases and vice-versa. Once detected and classified the heterogeneity in the time instant t, we need to know how this behaviour will vary in time instant $t+1$. To do so, a Next Program Phase Predictor (Np3) is used. We use the predicted behaviour as input to a microarchitecture-specific control hardware, the Thread Migration Arbiter. This arbiter has full control over thread migrations from core to core. It is designed to react to the heterogeneity in the running software and proceed according to the specific microarchitecture optimal decision in each case. In order to keep the complexity-effectiveness in the hDTM, we employ

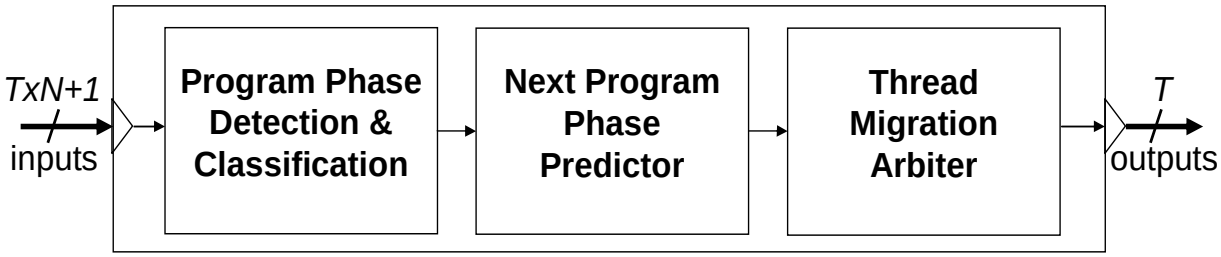

Figure 1. The hDTM mechanism.

a fixed-time-interval-based approach. This approach both reduces the amount of work done by the mechanism (once per fixed time interval) and avoids the need of additional and costly sampling phases.

In the following subsections it is explained in detail each of the three componets that comprise the hDTM mechanism, as depicted in Figure 1. Finally we discuss the complexity-effectiveness of the hDTM mechanism.

2.1 Program Phase Detection and Classification

The hDTM mechanism starts detecting heterogeneity in each of the heterogeneity sources it has been designed for (e.g throughput, chip surface temperature, etc). A program phase classification is needed as well as an effective way of detecting variations in program phases, for each source. In order to be appropriately monitored by a piece of hardware, the continuous time space is split into fixed intervals, or migration intervals. For each of these intervals, a per-thread behavior classification will be obtained by this first stage of the hDTM mechanism.

In the throughput-based implementation presented in this paper, we classify threads according to their memory behavior. In particular, we distinguish bad memory behavior program phases, i.e. memory-bounded (MEM), from good memory behavior program phases, i.e. high-ILP (ILP). We employ the second level cache (L2) miss ratio of each thread as an indicator of its memory behavior. Consequently, two metrics are monitored per thread, but resulting in a single metric per thread *(N=1)*: the number of L2 misses and the number of total L2 accesses, which yield the L2 miss ratio. In order to effectively distinguish between ILP and MEM program phases, we use a fixed threshold to classify the L2 miss ratio of each thread into ILP or MEM. Therefore, programs with an L2 miss ratio below 1% are considered

to go through an ILP phase, and MEM otherwise. Once each thread is classified as in an ILP or MEM program phase, this classification is passed to the next hDTM stage, the Next Program Phase Predictor (Np3).

2.2 Next Program Phase Predictor

In order to appropriately react to running software behaviour variations, it would be highly desirable to know beforehand how this behaviour is going to vary in the immediate future. Prior work [5, 21] has studied the detection of program phase changes and evaluated its importance. Basic block proportions, program working sets, and conditional branch counts have proved to accurately identify large scale program phase variations. However, they might not completely fit in an hDTM mechanism. The specific heterogeneity sources to be tracked in each case could require specific metrics to be appropriately detected. Moreover, the granularity required for each source may differ from the large scale program phases covered by prior work and even require a different granularity per source. Therefore, in order to appropriately react to each heterogeneity source, the processor needs at least a specific metric per each source, also tracking specific program phases per source. As an example, a multipurpose performance and temperature-aware processor could experience ILP-MEM performance program phase variations, as well as LOW-MEDIUM-HIGH temperature program phase variations. In each case, the desired hardware reaction will depend on the specific combination of program phases (e.g. ILP + MEDIUM).

The Next Program Phase Predictor (Np3) comprises the second stage of the hDTM mechanism. It predicts the heterogeneity variation in the next time interval (close future) per each heterogeneity source covered by the hDTM mechanism. In a multipurpose hDTM, there is one Np3 per heterogeneity source. Each of them receives T inputs and yields T outputs, where T represents the number of running threads. When a time interval ends, the inputs reflect the per-thread program phase classification corresponding to the time interval *t*, that is, the finalized interval. The outputs reflect the per-thread program phase classification prediction for the time interval *t+1*, that is, the next interval. This stage of the hDTM mechanism can be implemented in a wide range of possibilities, from simple approaches with limited accuracy to more sophisticated approaches, having high accuracy at the cost of higher complexity. A

detailed study of the possible implementations and their importance to the final results is objective of future work.

In the throughput-based implementation presented in this paper, we implement a Last Interval Np3. The predictor statically assumes that the program phase classification during the last interval will continue during the next interval. This is the simplest Np3 implementation in which Np3 inputs are directly connected to Np3 outputs.

2.3 Thread Migration Arbiter

Once the variations in the behavior of the running threads are predicted, it is time to react accordingly. The Thread Migration Arbiter is the responsible for performing the desired hDTM behaviour to stand up each specific heterogeneity source. It receives the per-thread program phase prediction for the next interval and it yields the core in which each thread will execute during the next interval. The arbiter is also responsible for actively migrating each thread according to both its inputs and its internal microarchitecture specific information. In a multipurpose hDTM mechanism, a Thread Migration Arbiter is needed per each heterogeneity source. These arbiters might yield simple independent per-source responses or more complex dependent inter-source responses, depending on the desired hDTM behaviour. The complexity of this arbiter will directly depend on the complexity involved in the desired hDTM behaviour. So, we could design arbiter implementations ranging from simple fixed tables to complex finite-state machines.

While the first two stages of the hDTM mechanism are independent of the specific multicore microarchitecture (except for the metric used for each heterogeneity source monitored), this stage is microarchitecture specific, that is, it is explicitly designed for a specific multicore microarchitecture. In fact, the arbiter is designed to optimally react to program behaviour variations in a specific microarchitecture. The thread migrations triggered by the arbiter take into account specific microarchitecture information such as the number of cores and the number of hardware contexts in each core.

In the throughput-based implementation presented in this paper, we add an hDTM to a POWER5-like [9] processor (see Section 4 for specific details). Our hDTM implementation bases on the fact that

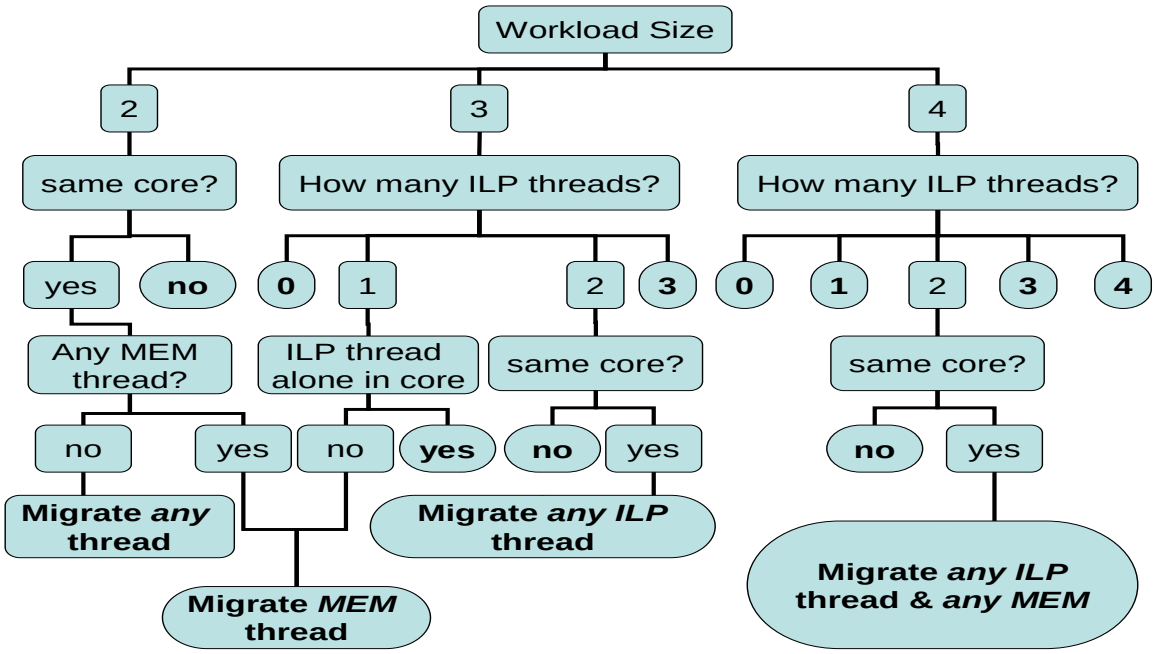

Figure 2. The Thread Migration Arbiter Algorithm.

threads without outstanding second level cache (L2) misses, or high-ILP (ILP) threads, put high pressure on the available functional units and issue bandwidth. However, they require fewer resources to exploit ILP than threads with frequent L2 misses, or memory-bounded (MEM) threads. In particular, MEM threads put pressure on instruction queues, physical registers, and the reorder buffer (ROB), while ILP threads put pressure on issue bandwidth, functional units, and register file ports. Consequently, our arbiter tries to avoid the ILP-ILP combination in any of the POWER5-like processor cores, as much as possible. The full arbiter behaviour implemented is depicted in Figure 2. Round and bold nodes represent leafs in the behavior tree.

Finally, notice that our arbiter in Figure 2 is orthogonal, or even complementary, to the specific SMT instruction fetch policy or resource allocation policy implemented in each of the cores. In fact, it strives to avoid multiple ILP threads sharing a single core, a non-profitable situation in most of the top-of-the-art instruction fetch and resource allocation policies [23, 2, 3], although the specific relation between hDTM and SMT instruction fetch and resource allocation policies is left for future work.

2.4 hDTM Complexity-Effectiveness

The first main issue about the complexity involved by the hDTM mechanism is the *granularity*, that is, how often hDTM works. The hDTM splits the continuous time space into migration intervals, and thus the hDTM will work once per interval. In other words, the hDTM cycle is $Pcycle \times migration\ interval\ length$, where *Pcycle* is the processor cycle. Since we generally define a migration interval length over 10K cycles we can assert that the hDTM mechanism is out of the processor critical path. This can be profited to both increasing the hDTM complexity, obtaining better results, or to implement it in a more complexity-effective way, such as low power or multi-clock signal frequency implementations. In this paper, we include an initial study of the impact of granularity in the hDTM mechanism, with migration intervals ranging from 10K to 1M cycles.

The second main issue about the complexity involved by the hDTM mechanism is the number of heterogeneity sources. As we have already discussed in this section, an hDTM can be designed to make the multicore processor aware of multiple heterogeneity sources (e.g. performance and chip surface temperature). However, it should be taken into account that the more sources used by the hDTM implementation, the more complexity involved.

The third main issue about the complexity involved by the hDTM mechanism is the complexity of the Thread Migration Arbiters. The hDTM mechanism's complexity directly depends on the complexity involved in the response to each heterogeneity source. Besides, the complexity of each arbiter can be even higher if its response depends on others heterogeneity sources. In this sense, an independent arbiter per source does not involve the same complexity rather than *N* arbiters combined into a single and more complex one. Hence, the complexity of each arbiter depends on the number of interrelated heterogeneity sources required to produce its specific output.

The fourth main issue, and probably the most important, is the core-to-core thread migration cost. In each thread migration, all the architectural state registers (arch. Ints and FPs, program counter) have to be moved from core to core. Additionally, all the instructions in the original core are flushed, hence they have to be fetched again in the destination core. The contents of branch predictor and first level caches

are also lost when moving to a new core with its own branch predictor and first level caches, resulting in additional misses in the destination core. All these factors comprise the core-to-core thread migration cost, an overhead used by the Thread Migration Arbiter to decide, in each case, whether each migration is worthwhile.

3 Related Work

Numerous prior work [17, 20] has studied the inherent heterogeneity in typical general-purpose workloads and have presented proposals that divide programs in phases [5, 21] to both detect and predict the variations in these workloads. Basic block vectors, instruction working sets, and conditional branch counts have proved to efficiently capture, classify, and predict phase-based program behavior on the largest of time scales. However, a variable-granularity microarchitecture-specific solution could be needed in order to allow a hardware optimal reaction to heterogeneity.

Some other work [2, 3, 22, 23] has proposed instruction fetch or resource allocation policies to dynamically adapt the workload hardware resource allocation. In [23, 2], MEM+ILP thread coexistence in a single core is improved by freeing unused MEM thread resources. In [22, 3] the progress and resource utilization of jobs on the machine is dynamically monitored , dynamically adjusting the degree of multithreading and the per-thread resource allocation to improve performance while still meeting priority goals. Although a deeper analysis of the hDTM impact on the specific per-core instruction fetch or resource allocation policy is out of the scope of this paper, we believe that they can be complementary. The hDTM covers the CMP upper layer of the hardware resource allocation while each core's instruction fetch or resource allocation policy cover the SMT lower layer, inner to each core.

Some recent work [11, 1, 10] introduces the heterogenity-awareness into the processor design. Each of them present an heterogeneity-aware proposal that ranges from the statically heterogeneity-aware SMT-based approach in [1] to the dynamically CMP-based approach in [10]. In [11] a more specific clustered SMT-based approach is presented in which clusters (FrontEnds and BackEnds) are dynamically reallocated to running application according to program behavior variations. From all these works,

the Heterogeneous Multicore [10] is the only one that both reflects the heterogeneity into an heterogeneously distributed hardware and dynamically match the varying requirements of each thread with the heterogeneous hardware. However, the complexity involved in such a scheduling (multiple heterogeneous behavior matched with multiple heterogenous cores) requires additional hardware reassignment costs (sampling phases [10]). Our proposal represents a general-purpose solution. In fact, an hDTM can be added to each of these proposals to perform the dynamic hardware adaptability to changes in running application behavior. In each case the Thread Migration Arbiter is in charge of performing the desired reaction to heterogeneity. This may lead to reductions in hardware reassignment costs (sampling phases [10] replaced by a complexity-effective heterogeneity monitoring) by increasing the Thread Migration Arbiter complexity (to appropriately catch a complex heterogenous core scheduling).

Finally, mention the work in [8]. The proposed activity migration scheme is a clear example of an specific single-purpose hDTM implementation focused on the chip surface temperature as only heterogeneity source. However, depending on the purpose of each specific piece of hardware (embedded, general-purpose microprocessor, etc) both the heterogeneity sources considered and the arbiter behavior can drastically change. Notice that the purpose of the hDTM implementation will depend on each specific scenario.

4 Simulation Setup

We use a trace driven SMT simulator derived from SMT-SIM [24]. The simulator consists of our own trace driven front-end and an improved version of the SMT-SIM back-end that provides multicore support. Our simulator also permits simulating the impact of executing along wrong paths on the branch predictor and the instruction cache by having a separate basic block dictionary in which information of all static instructions is contained. With this simulation infrastructure, we simulate a POWER5-like processor, that is, a dual-core multithreaded processor. Each core implements an ICOUNT 2.8 [24] instruction fetch policy. Table 1 shows the main parameters of each core of the simulated CMP processor.

Regarding migration costs, each thread core-to-core migration involves a cost equal to 100 cycles

to communicate the architectural register state through an specific register-size interconnection bus. Besides, since thread migrations involve flushing a thread in the original core, the time required to fetch again the prior instruction window in the new core is also taken into account in each migration. Finally, since a CMP processor has per-core private fetch engine and first level caches, and the hDTM does not includes additional specific hardware to migrate this sort of thread state, additional thread migration cost is incurred due to the warmup period required by the conditional branch predictor and the first level caches in the destination core.

POWER5-like core	
Pipeline depth	11 stages
Pipeline width	8
Hardware contexts	2
Queues Entries	64 int, 64 fp, 64 ld/st
Execution Units	4 int, 2 fp, 2 ld/st
Physical Registers	120 regs.
ROB Size*	100 entries
Branch Predictor	perceptron (4K local, 256 perceps)
BTB	256 entries, 4-way associative
RAS*	100 entries
L1 I-Cache	64KB, 2-way, 8 banks
L1 D-Cache	32KB, 4-way, 8 banks
L1 lat./misspenalty	3/22 cyc.
I-TLB/D-TLB/TLB missp.	48 ent. / 128 ent. / 300 cyc.
Memory	
L2 Cache	1.875MB, 10-way, 3 banks
L2 latency	12 cyc.
Main Memory Latency	250 cyc.

Table 1. POWER5-like core simulation parameters (resources marked with * are replicated per thread)

Our workloads use the SPEC2000 benchmark suite. From them, we have collected traces of the most representative 300 million instruction segment of each benchmark, following the idea presented in [18]. Each program is compiled with the *–O2 –non_shared* options using DEC Alpha AXP-21264 C/C++ compiler and executed using the reference input set. Since a complete study of all benchmarks is not feasible due to excessive simulation time, we have used the workloads shown in Table 2. We have used workloads including 2, 3, and 4 threads. Workloads are classified according to the cache behavior of the included benchmarks: those with an L2 cache miss rate higher than 1% are considered memory bounded (MEM); the others are considered high-ILP (ILP). A workload with a mixture of ILP and MEM thread is considered MIX. This is a static per-thread classification. During program execution, a MEM

# of threads	Thread type	Workload group 1	Workload group 2	Workload group 3	Workload group 4
2	ILP	gzip, bzip2	wupwise, gcc	fma3d, mesa	apsi, gcc
	MIX	gzip, twolf	wupwise, twolf	lucas, crafty	equake, bzip2
	MEM	mcf, twolf	art, vpr	art, twolf	swim, mcf
3	ILP	gcc, eon, gap	gcc, apsi, gzip	crafty, perl, wupwise	mesa, vortex, fma3d
	MIX	twolf, eon, vortex	lucas, gap, apsi	equake, perl, gcc	mcf, apsi, fma3d
	MEM	mcf, twolf, vpr	swim, twolf, equake	art, twolf, lucas	equake, vpr, swim
4	ILP	gzip, bzip2, eon, gcc	mesa, gzip, fma3d, bzip2	crafty, fma3d, apsi, vortex	apsi, gap, wupwise, perl
	MIX	gzip, twolf, bzip2, mcf	mcf, mesa, lucas, gzip	art, gap, twolf, crafty	swim, fma3d, vpr, bzip2
	MEM	mcf, twolf, vpr, parser	art, twolf, equake, mcf	equake, parser, mcf, lucas	art, mcf, vpr, swim

Table 2. Workload classification based on memory behavior of threads.

benchmark can go through numerous ILP phases and vice versa. Benchmarks in each of the four groups per workload size and type (see Table 2) have been selected randomly. In each experiment, we strictly focus on the period of time in which all the initial threads share the processor. The objective in each case is evaluating the behavior of each microarchitecture with workloads of two, three and four threads. This means that each simulation finishes as soon as one thread contained in the evaluated workload finishes executing 300 million instructions.

5 Simulation Results

Figure 3 shows the throughput results obtained, measured in instructions per cycle (IPC), after simulating the workloads in Table 2 in our processor. Each workload is simulated in all possible permutations. Hence, a 4-threaded workload [A,B,C,D] is executed three times: (A,B)-(C,D),(A,C)-(B,D) and (A,D)-(B,C), where threads in brackets represents threads in the same core. Each bar in Figure 3 represents the harmonic mean of all workloads of an specific size and type (e.g. 2W+ILP), chosed according to a given criterion. In each case, from left to right, the criteria applied are as follows: first, all best static permutations (STATIC BEST), by applying an oracle static thread-to-core assignment; next, the harmonic mean of all possible permutations (STATIC MEAN); next, all worst static permutations (STATIC WORST); next, the harmonic mean of all possible initial permutations using our hDTM implementation with a migration interval of 10K cycles (DYNAMIC 10K); the last two are similar to the fourth one, but varying the migration interval length to 100K (DYNAMIC 100K) and 1 million (DYNAMIC 1M) cycles.

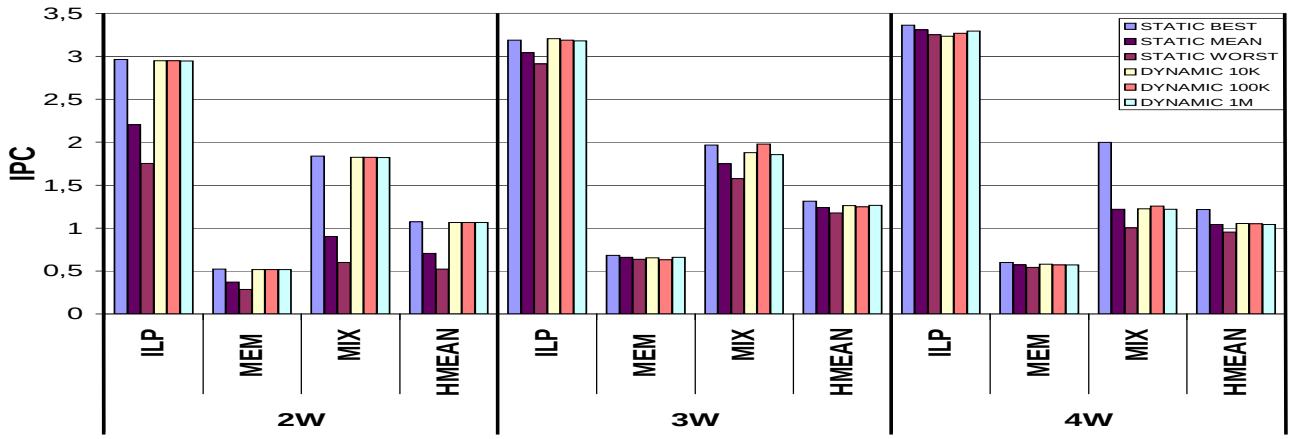

Figure 3. Throughput results.

From these results we can conclude that even a naive and simple hDTM implementation, as the one presented herein, is able to get close to its goal, obtaining results for 2Ws, 3Ws and 4Ws within 1%, 4%, and 15% best results in each case (STATIC BEST). Since 2-threaded workloads (2W) do not represent a challenge, since it is enough to put each thread in a different core, we more deeply analyze 3Ws and 4Ws. The 4Ws represent a saturated case in which all hardware contexts are busy, dramatically reducing the hDTM's leeway. From 3Ws we conclude that hDTM allows not only avoiding losses due to bad thread-to-core assignments (up to a 25% in 3W+MIX) but also going beyond the best static upper limit (3W+ILP,3W+MIX), although marginally (1%) due to the naive hDTM implementation presented in this paper.

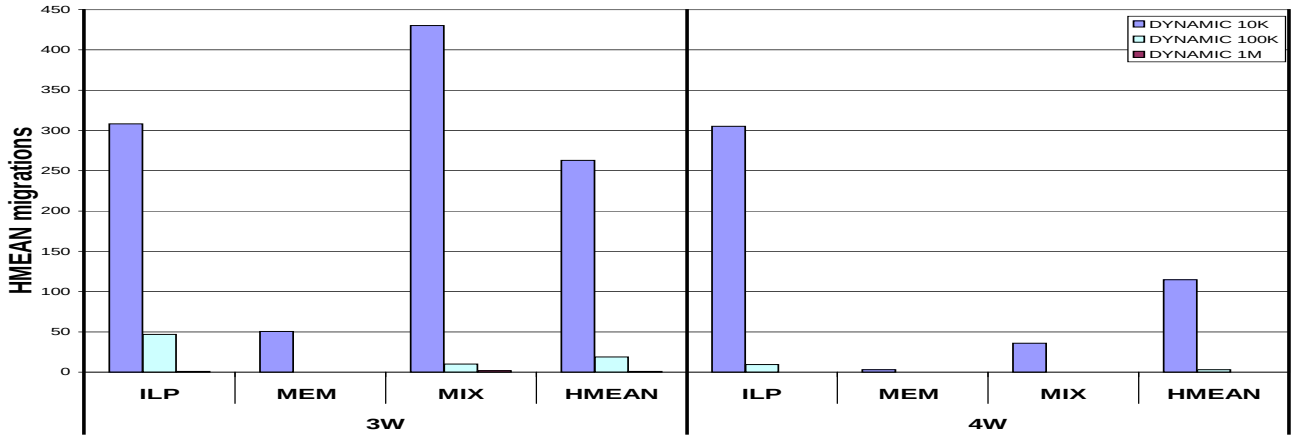

Figure 4. Migration results.

Finally, our results indicate that there is not a clear best granularity (migration interval length) to be applied. Strictly focusing on throughput results, the 10K-cycle granularity appears as slightly better than the others; although surpassed by 100K-cycles granularity in 3W+MIX by 5% in mean. However, taking into account the harmonic mean of migrations required by each granularity, shown in Figure 4, the higher granularity (1M) clearly appears as the best choice, involving less overhead.

6 Conclusions

In this paper we have presented the Heterogeneity-Aware Dynamic Thread Migrator (hDTM), a novel complexity-effective proposal to expose the heterogeneity in the running software to the underlying multithreaded-multicore hardware. This general-purpose hardware solution monitors the running workload behavior variations and reacts accordingly, migrating threads in the underlying multicore processor. The behavior and complexity involved will depend on each specific implementation and underlying microarchitecture. The results presented herein indicate that even a naive and simple hDTM implementation is able to get close to hDTM's goal, within 1-15% best static results, not only avoiding losses due to bad thread-to-core assignments (up to a 25%) but also going beyond the best static upper limit.

Acknowledgements

This work has been supported by the Ministry of Education of Spain under contract TIN2004–07739–C02–01, the HiPEAC European Network of Excellence, and the Barcelona Supercomputing Center. Carmelo Acosta is also supported by the Ministry of Science and Technology of Spain grant BES–2002–0015. The authors also want to thank Daniel Ortega for his valuable help with the simulator.

References

[1] C. Acosta, A. Falcón, A. Ramírez, and M. Valero. A Complexity-Effective Simultaneous Multithreading Architecture. In *Proc. of ICPP-35*, 2005.

[2] F. J. Cazorla, E. Fernández, A. Ramirez, and M. Valero. Improving memory latency aware fetch policies for SMT processors. In *Proc. of ISHPC-V*, 2003.

[3] F. J. Cazorla, E. Fernández, A. Ramirez, and M. Valero. Dynamically Controlled Resource Allocation in SMT Processors. In *Proc. of MICRO-37*, 2004.

[4] J. D. Collins and D. M. Tullsen. Clustered multithreaded architectures – Pursuing both IPC and cycle time. In *Proc. of IPDPS-18*, 2004.

[5] A.S. Dhodapkar and J.E. Smith. Comparing program phase detection techniques. In *Proc. of MICRO-36*, 2003.

[6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. G. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proc. of MICRO-30*, 1997.

[7] L. Hammond, B. A. Nayfeh, and K. Olukotun. Single-chip multiprocessor. In *IEEE Computer Special Issue on Billion-Transistor Processors*, 1997.

[8] S. Heo, K. Barr, and K. Asanovic. Reducing Power Density through Activity Migration. In *Proc. of ISLPED*, 2003.

[9] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, March 2004.

[10] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proc. of ISCA-31*, 2004.

[11] F. Latorre, J. González, and A. González. Back-end Assignment Schemes for Clustered Multithreaded Processors. In *Proc. of ICS-18*, 2004.

[12] S. W. Lee and J. L. Gaudiot. Clustered microarchitecture simultaneous multithreading. In *Proc. of EuroPAR-9*, 2003.

[13] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proc. of ASPLOS-7*, 1996.

[14] S. Palacharla, N. P. Jouppi, and J.E. Smith. Complexity-effective superscalar processors. In *Proc. of ISCA-24*, 1997.

[15] Intel White Paper. Desktop Performance and Optimization for Intel Pentium 4 Processor. 2001.

[16] S. E. Raasch and S. K. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *Proc. of PACT-12*, 2003.

[17] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report CS99-630, 1999.

[18] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proc. of PACT-10*, 2001.

[19] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, November 2003.

[20] T. Sherwood, E. Perelman, Greg Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. of ASPLOS-10*, 2002.

[21] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proc. of ISCA-30*, 2003.

[22] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS Conf. Measurement and Modeling of Comput. Syst.*, 2001.

[23] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreaded processor. In *Proc. of MICRO-34*, 2001.

[24] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of ISCA-22*, 1995.

[25] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. of ISCA-23*, 1996.

[26] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Proc. of PACT*, 1995.