A Complexity-Effective Simultaneous Multithreading Architecture

Carmelo Acosta † Ayose Falcón ‡ Alex Ramirez † Mateo Valero †

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

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

Abstract

Different applications may exhibit radically different behaviors and thus have very different requirements in terms of hardware support. In Simultaneous Multithreading (SMT) architectures, the hardware is shared among multiple running applications in order to better profit from it. However, current architectures are designed for the common case, and try to satisfy a number of different application classes with a single design. That is, current designs are usually overdesigned for most cases, obtaining high performance, but wasting a lot of resources to do so.

In this paper we present an alternative SMT architecture, the Heterogeneously Distributed SMT (hdSMT). *Our architecture is based in a novel combination of SMT and clustering techniques in a heterogeneity-aware fashion. The hardware is designed to match the heterogeneous application behavior with the statically and heterogeneously partitioned resources. Such a design is aimed for minimizing the amount of resources wasted to achieve a given performance rate. On top of our statically partitioned architecture, we propose an heuristic policy to map threads to clusters so that each cluster matches the characteristics of the running threads and overall hardware usage is optimized.*

We compare our hdSMT architecture with a monolithic SMT processor, where all threads compete for the same resources, and with a homogeneous clustered SMT, where resources are statically and equally partitioned across clusters. Our results show that hdSMT architectures obtain an average improvement of 13% and 14% in optimizing performance per area over monolithic SMT and homogeneously clustered SMT respectively.

Keywords - SMT, CMP, Clustering, Complexity-Effective, Heterogeneity-Awareness, Mapping Policies.

1 Introduction

The needs of today's multi-programmed workloads put pressure on microprocessor design towards high-performance and high-throughput machines. Thus, new approaches have arisen aimed at such a multithreaded scenario, improving traditional superscalar processor capabilities. Simultaneous multithreading (SMT) [20, 18, 19] and chip multiprocessors (CMP) [11, 6] are two of these approaches. The first one evolves the traditional superscalar architecture by sharing all the processor resources among more than one running thread. The latter relies on simpler cores, replicating them on a single chip and allocating running threads to these cores. Each one represents a different approach to optimize the performance that a fixed transistor budget can produce: A big machine where every resource is shared versus several simpler machines where the sharing locality is restricted. But they also imply a commitment: the single thread high-performance of SMT, at a complexity cost, against the low complexity but limited single-threaded performance of CMP. However, there is also a wide spectrum in between SMT and CMP approaches as we vary the amount of shared resources on chip [4].

As the number of transistors on chip increases, the issue of how to employ them to achieve the highest performance potential gets renewed importance. The design complexity has to remain under reasonable costs while it achieves the highest performance potential. But achieving high levels of both performance and throughput from a given hardware budget at a reasonable complexity cost is not an easy task. Employing the additional resources to simply stretch traditional structures, such as instruction queues, is not conducive towards building highly pipelined processors with short clock periods. Besides, power and thermal considerations also have to be taken into account since future microprocessors' designs are likely to be limited by them. In this sense, the reduced complexity of the CMP approach combined with the resource exploitation capacity of the SMT approach seems an appealing alternative.

General-purpose designs treat applications homogeneously, although not all applications have the same behavior. In fact, we can find a huge inter-application heterogeneity in current multi-programmed workloads, which can be measured in terms of memory misses, branch mispredictions or instruction level parallelism (ILP) among others. This heterogeneity results sometimes in applications executed using an amount of resources and power that is not cost-effective with the performance obtained. The techniques applied in SMT processors to reduce the contention over shared resources between conflictive applications [17, 5] can help to face up this heterogeneity in application behavior. However, instead of helping to reduce the design complexity they even increase it. In the CMP approach, each running application is assigned to one of the homogeneous cores in which the hardware has been partitioned. The design complexity is kept low at the cost of a static limitation to the hardware that each application can use, equal for all applications. By making the hardware conscious of this inter-application heterogeneity, or heterogeneity-aware, we could keep design complexity reasonably low without losing too much performance. The central insight behind a heterogeneity-aware design is giving each application access to a cost-effective amount of resources.

In this paper, we propose the *Heterogeneously Distributed Simultaneous Multithreading (hdSMT)* architecture, a novel heterogeneity-aware SMT architecture that combines SMT and clustering techniques. We review the open issue of on chip resource distribution and propose a simultaneous multithreading processor in which all the pipeline stages but the fetch stage have been heterogeneously clustered —pipelined from now on—, making up a multipipeline SMT processor. Besides the fetch engine, all the pipelines share the memory subsystem —including L1 caches— and the register file. Consequently, the single-thread performance is not hampered by a memory or register file static distribution as could happen in a CMP processor. In this architecture the heterogeneity of the typical software that will be executed on the processor is analyzed and mirrored itself in the hardware. Thus, the heterogeneous behavior of the running applications is matched with the heterogeneous hardware, mapping software needs to heterogeneously partitioned hardware resources. This hardware configuration also allows reducing the contention between conflictive threads as occurs in SMT processors. Thus, the application-to-pipeline matching process also intends to put conflictive threads in different pipelines, in order to reduce a counterproductive interaction.

2 The hdSMT Architecture

The foundations of the hdSMT architecture are comprised of a threefold combination of well known principles and techniques: *SMT*, *clustering*, and *heterogeneity-awareness*. An hdSMT processor proposes a multithreaded alternative that lays on the spectrum that extends in between SMT and CMP processors. As evaluated in [4], there are multiple possible hardware configurations in between SMT and CMP processors, as we vary the amount of resources shared among the execution cores. However, the heterogeneity in applications' behavior makes vary the hardware requirements among different applications. This heterogeneity may turn the evenly clustered approaches in [4] into not optimal. To better profit from the available hardware it should be heterogeneously clustered and the applications appropriately matched with the clusters according to their needs. The hdSMT architecture maximizes the available hardware budget by taking into account the heterogeneity in this way.

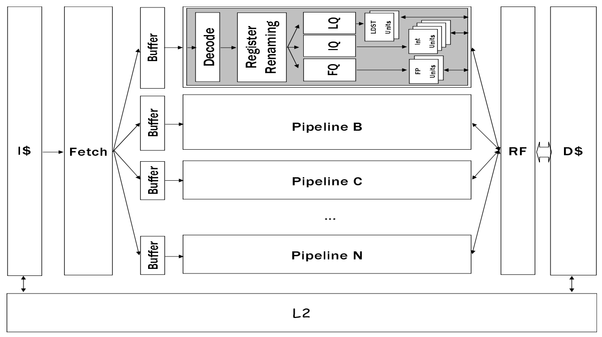


Fig. 1. The hdSMT Architecture.

The hdSMT architecture overview is depicted in Fig. 1. As in a conventional SMT processor, all threads share the caches, register file, and fetch engine. However, the rest of the pipeline stages and resources are arranged in heterogeneous clusters (or pipelines). So, each pipeline comprises all the pipeline stages of the conventional processor but the fetch stage. Each pipeline also has got its own private instruction queues, renaming map tables and functional units. The size and number of these resources may vary from pipeline to pipeline. Additionally, each thread's instructions are stored in a private reorder buffer (ROB), one per thread.

In this clustered multithreaded architecture, entire threads are assigned to pipelines according to heterogeneity. This implies that there are no dependencies between instructions in different clusters, since all instructions from a single thread are mapped to the same pipeline. The heterogeneity-aware fetch engine strives to match both the needs of each running application and the interaction among each application with the heterogeneously distributed hardware. This software-hardware mapping is performed each time the job scheduler of the operating system selects a new bunch of active threads. The whole subsequent execution of the workload is done according to this mapping. We describe more in detail the mapping policy in Section 2.1.

The number of hardware contexts and width of each pipeline may vary from pipeline to pipeline. So, an hdSMT microarchitecture may be comprised of both narrow single-threaded and wide multithreaded pipelines, as well intermediate pipelines. Depending on the resource needs of each application and the interaction between application behavior, more than one application may be mapped to a single pipeline. This distribution of the hardware contexts along the chip can be profited to turn off idle pipelines whenever the number of running applications does not reach the number of hardware contexts. This is also applied in the Heterogenous Multi-Core architecture [7], turning off idle heterogeneous cores. The main difference of our proposal in this sense is that we can still use the whole budget of physical registers and memory space to improve the performance of the running applications, since they are shared by all pipelines.

Notice that multipipeline-awareness in hdSMT uncovers new fetch policies not available in conventional SMT processors. The shared fetch engine is limited by the number and width of the instruction cache ports. However, the number of instructions that each pipeline accept per cycle may vary from pipeline to pipeline. In order to decouple the fetch engine from the characteristics of each specific pipeline it feeds, some small buffers are added before each pipeline (see Fig. 1). Thus, the fetch engine inserts in-order the fetched instructions at its own rate while each pipeline extracts in-order instructions according to its width. The fetch policy takes into account these buffers in order to appropriately balance the instructions fetched among the pipelines. Depending on the pipeline set characteristics, this may result in a wider global decode bandwidth since all pipelines are fed from their private buffer each cycle.

2.1 Mapping policies in hdSMT

The impact of static partitions of the hardware may be either productive or counterproductive depending on the resource partitioned. Thus, as showed in [12], while statically partitioning instruction queues provides good performance, an static division of the issue bandwidth has a negative impact on throughput.

In order to avoid this penalty, the hardware in the hdSMT architecture is heterogeneously distributed and a thread-to-pipeline mapping policy is applied. The success in avoiding this negative effect will depend on the ability of the mapping policy to map high-performing threads to wide pipelines, to profit from their wide issue bandwidth.

In this work we have used a simple profile-based heuristic policy that uses the memory behavior of each thread to do the mapping. By means of profile information, the active threads are arranged by the number of data cache misses and assigned to the pipelines. The full mapping process is as follows:

1. Arrange all active threads by the number of data cache misses in a list (T). The first thread in T is the one with the lesser number of misses.
2. Arrange all pipelines by their width in a list (P). The first pipeline in P is the widest one.
3. Map the first thread in T to the first pipeline in P.
4. If this is the first assignment, and there are more available hardware contexts than active threads then remove the top of the list P.
5. Remove the top of the list T.
6. If all the hardware contexts of the pipeline in the top of the list P are busy then remove the top of the list P.
7. If list T is not empty continue in step 3.

Regarding interaction among applications, it is assumed that applications with a similar number of data cache misses behave similarly and therefore can share a single pipeline. Thus, the negative scenario in which applications with a bad memory behavior hinder applications with a good memory behavior is avoided. In this sense, our mapping policy assumes that adjacent applications in the list T behave similarly and consequently could share a single pipeline.

In order to match each application with the appropriate pipeline, our mapping policy makes this simple assumption: the number of data cache misses of an application is inversely proportional to the pipeline width required. The more data cache misses occurred during an application execution, the more resources will be held by that application while each miss is resolved. By doing so, we expect to match each application with the most appropriate pipeline, that is the one in which it is obtained the highest performance but involving the lowest resource budget.

3 Area Cost Model

In this work we evaluate different microarchitectures, which involve different hardware budgets. Since comparing the results produced by microarchitectures with different amount of resources may be quite unfair, we need some complexity measurement to guide this evaluation. Quantifying complexity is a tricky task and giving a single and comparable measurement is even harder to accomplish. In this paper we follow a quite generalized approach and use the area (in mm^2) of the processor as a metric of its "complexity". Although complexity is not proportional to area in all cases, it gives a quite accurate idea of the resultant complexity and is reasonably easy to be measured.

To estimate the area of each configuration we employ the *Karlsruhe Simultaneous Multithreaded Simulator* [14, 15, 16]. On top of this area estimation tool we develop our area cost model. Since both hdSMT and SMT approaches share the same register file and caches, we have removed them from the model to simplify the results. However, since in hdSMT these resources are shared among all pipelines, the additional logic cost is taken into account. It is added to the execution core of each pipeline, as additional hardware for data access. The hdSMT fetch engine also needs some additional logic. Although its characteristics are similar to the SMT one, multipipeline support requires some extra logic. Taking into consideration Burns and Gaudiot work in quantifying SMT layout overhead [1, 2], we have extrapolated single to multipipeline environment area overhead from single to multithreading environment. So we have estimated the area overhead of the execution core within each pipeline in a 10%. The conventional SMT fetch engine area overhead, when applied to a hdSMT multipipeline environment, has been estimated in a 20%.

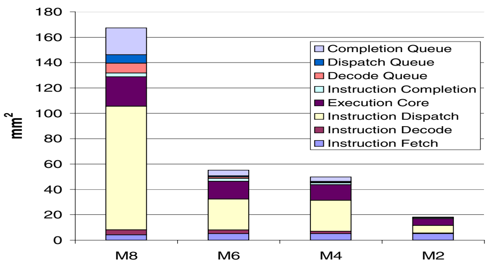
In our evaluation, we use four different models of pipeline, named *M8*, *M6*, *M4*, and *M2*. The number in each model name gives a hint of the amount of resources it has been devoted. Our conventional monolithic SMT baseline processor is represented by the M8 model. The remainder models represent pipelines with reduced resources budget with respect to the baseline. The functional units are among the private resources of each pipeline. In order to choose the most appropriate number of functional units for each pipeline, we evaluated the performance obtained as we reduced them, starting from the baseline model (M8). With all other resources changed to the pipeline new values, in each case it was chosen the number of functional units that kept the slowdown below the 2%.

Our area cost model considers the total area as the sum, for all constituent pipelines, of the instruction fetch, decode, dispatch, execution core, and instruction completion stages plus the decode, dispatch, and completition queues. In hdSMT and homogeneously clustered SMT configurations, comprised of combinations of M6, M4, and M2 models, only one instruction fetch stage is included in the total area calculus. In Fig. 2.(a) the amount of resources devoted to each pipeline model is shown. Additionally, we have assumed a per-thread 256-entry ROB in all configurations, both SMT and hdSMT. In Fig. 2.(b) we show the area estimation of each model according to our area cost model. All estimations have been made in 0.18 μm, as in [1], to ease our area overhead extrapolations. Notice that in Fig. 2.(b) M6, M4, and M2 pipelines are accompanied by an instruction fetch stage a 20% bigger than the baseline (M8) one. Each of them represent in fact an hdSMT processor with a single pipeline, the one measured in each case.

Finally, as shown in Fig. 2.(a), our SMT baseline (M8) is not able to execute more than four threads. Although adding additional hardware contexts increases the total area of an SMT processor, as Burns and Gaudiot evaluate in [1], we assume no additional area overhead for this model when adding two additional hardware contexts; in order to execute workloads of six threads on it.

	M8	M6	M4	M2
Hardware Contexts	4	2	2	1
Max. Instr./cycle	8	6	4	2
Max. Threads/cycle	2	2	2	1
Queues (IQ/FQ/LQ)	64	32	32	16
Integer Func. Units	6	4	3	1
FP Func. Units	3	2	2	1
LD/ST Units	4	2	2	1

a) Resources.



b) Area estimation.

Fig. 2. Pipeline models.

4 Simulation Setup

We use a trace driven SMT simulator derived from SMT-SIM [18]. The simulator consists of our own trace driven front-end and an improved version of SMT-SIM's back-end, that provides multipipeline support among others. Both the monolithic SMT and the multipipeline configurations have an 8 stage pipeline depth. Our simulator also permits execution along wrong paths by having a separate basic block dictionary in which information of all static instructions is contained.

As mentioned earlier in Section 3, we use four different models of pipelines: M8, M6, M4, and M2. Our monolithic SMT baseline processor is represented by model M8. Additionally to model characteristics shown in Fig. 2.(a), the baseline configuration, used in both monolithic and multipipeline configurations, is shown in Table 1. Besides, since hdSMT requires additional logic to handle multipipeline register file sharing we duplicate the number of cycles required by register reads/writes in hdSMT configurations. Thus, register reads/writes have a latency of 1 cycle in case of a monolithic SMT processor as against the 2 cycle latency of the hdSMT processors.

In our experiments, we adopt the FLUSH [17] fetch policy for the baseline (M8) case. This fetch policy, built on top of ICOUNT 2.8 [19], predicts an L2 miss every time a load spends more cycles in the cache hierarchy than needed to access the L2 cache. In case of L2 miss, the instructions after the L2 missing load are flushed, and the offending thread is stalled until the load is resolved. Thus, the resources used by the offending thread are freed and it does not compete

Branch Predictor	perceptron (4K local, 256 perceps)
BTB	256 entries, 4-way associative
RAS*	256 entries
ROB Size*	256 entries
Rename Registers	256 regs.
L1 I-Cache	64KB, 2-way, 8 banks
L1 D-Cache	64KB, 2-way, 8 banks
L1 lat./misspenalty	3/22 cyc.
L2 Cache	512KB, 2-way, 8 banks
L2 latency	12 cyc.
Main Memory Latency	250 cyc.
I-TLB/D-TLB/TLB missp.	48 ent. / 128 ent. / 300 cyc.

Table 1. Simulation parameters (resources marked with * are replicated per thread)

for new resources until the load is resolved. This allows the other threads to proceed while the stalled thread is waiting for the outstanding cache miss.

In all other cases, we adopt the L1MCOUNT fetch policy, a variant of the DCache Warn fetch policy [3]. This fetch policy keeps track of the number of inflight loads. Threads are arranged by the number of inflight loads they have and given fetch priority accordingly. Threads with fewer number of inflight loads have priority. In case of equal number of inflight loads, threads allocated to wider pipelines have priority over those in narrower pipelines. Finally, in case of pipeline coincidence, the ICOUNT 2.8 policy is applied. Regardless of the fetch policy, all simulations are limited to 8 instructions fetchable per cycle, from a maximum of 2 threads. In order to decouple the shared fetch engine from each pipeline characteristics, we have put a buffer between the fetch engine and each pipeline (see Fig. 1). The size of these buffers is 32 entries, for M6 and M4 pipeline models, and 16 entries, for M2 pipeline model.

We use the SPEC2000 integer benchmark suite. From them, we have collected traces of the most representative 300 million instruction segment of each benchmark, following the idea presented in [13]. 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. Tables 2 and 3 show the workloads used in our simulations. We have used workloads including 2, 4, and 6 threads. Workloads are classified according to the characteristics of the included benchmarks: with high instruction-level parallelism (ILP), with bad memory behavior (MEM), or a mix of both (MIX). Due to the characteristics of SPECint2000, with few benchmarks that are really memory bounded, MEM workloads are only feasible for 2 and 4 threads.

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, four and six threads. This means that each simulation finishes as soon as one thread contained in the evaluated workload finishes executing 300 million instructions.

Wld	Benchmarks	T	Wld	Benchmarks	T
2W1	eon, gcc	I	4W1	eon, gcc, gzip, bzip2	I
2W2	crafty, bzip2	I	4W2	crafty, bzip2, eon, gzip	I
2W3	gap, vortex	I	4W3	gap, vortex, parser, crafty	I
2W4	mcf, twolf	M	4W4	mcf, twolf, vpr, perlbmk	M
2W5	vpr, perlbmk	M	4W5	vpr, perlbmk, mcf, twolf	M
2W6	vpr, twolf	M	4W6	gzip, twolf, bzip2, mcf	X
2W7	gzip, twolf	X	4W7	crafty, perlbmk, mcf, bzip2	X
2W8	crafty, perlbmk	X	4W8	parser, vpr, vortex, twolf	X
2W9	parser, vpr	X	4W9	vpr, twolf, gap, vortex	X

Table 2. Two and four threaded workloads (I=ILP, M=MEM, X=MIX)

Wld	Benchmarks	T
6W1	gzip, gcc, crafty, eon, gap, bzip2	I
6W2	gcc, crafty, parser, eon, gap, vortex	I
6W3	gzip, vpr, mcf, eon, perlbmk, bzip2	X
6W4	vpr, mcf, crafty, perlbmk, vortex, twolf	X

Table 3. Six threaded workloads.

4.1 Microarchitectures and Metrics

In our experiments, we evaluate several multipipeline microarchitectures, both homogeneously and heterogeneously distributed. All these multipipeline microarchitectures are implementations of the hdSMT architecture[1]. In Fig. 3 we show the area estimation of all microarchitectures evaluated. Beneath each area estimation appears the microarchitecture name, which stands for the number and type of pipeline models involved. So, the 2M4+2M2 microarchitecture is comprised of 2 pipelines of type M4 plus two pipelines of type M2 (see Fig. 2 for specific details of each pipeline type). From left to right, the first microarchitecture (M8) in Fig. 3 represents our monolithic SMT baseline. The next two microarchitectures (3M4 and 4M4) are homogeneous clustered hdSMT microarchitectures. Finally, the last three microarchitectures represents the truly hdSMT microarchitectures. According to Fig. 3, all but two microarchitectures (4M4 and 1M6+2M4+2M2) require less area than the monolithic SMT baseline. That is, they are “simpler” than the SMT baseline.

For each microarchitecture we evaluate its performance (IPC) for all workloads. However, since each microarchitecture has a different resource budget and consequently a different performance potential, we also take into account the complexity involved. In order to make a fairer comparison we combine the performance and the complexity of each microarchitecture in a single metric. Thus, in this paper we also provide results measured in Performance per Area, which is obtained by dividing the resulting performance of a microarchitecture by its area (in mm^2). This additional metric allows to evaluate the “complexity-effectiveness” of each microarchitecture.

[1]Although the homogeneous ones do not obey the hdSMT principle of heterogeneous distribution of resources.

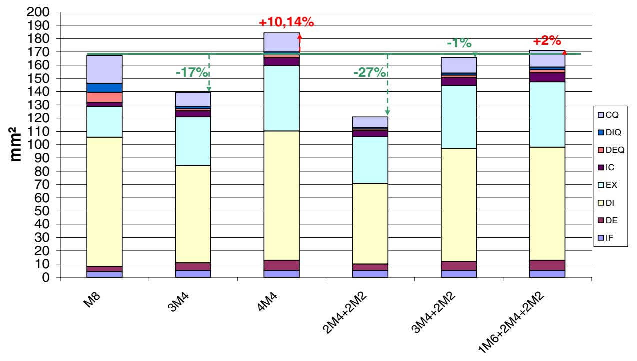


Fig. 3. Area estimation of evaluated microarchitectures.

5 Simulation Results

In this section, we evaluate and compare monolithic SMT, homogeneously distributed hdSMT, and heterogeneous distributed hdSMT processors. For each workload, three measurements are given. First, the BEST result, obtained using an oracle thread mapping policy, gives the maximum performance of the microarchitecture. Second, the HEUR result gives the performance obtained by the microarchitecture using the heuristic thread mapping policy presented in Section 2.1. Finally, the WORST result gives the performance obtained by the microarchitecture in case of applying in each case the worst possible thread-to-pipeline mapping. Special cases are the baseline (M8) and the two-threaded workloads of homogeneous distributions (3M4 and 4M4). Since the baseline is not multipipelined, no thread-to-pipeline mapping policy is needed and so only one measurement is given. In two-threaded workloads, when all pipelines are of the same sort the three measurements (BEST, HEUR, WORST) coincide.

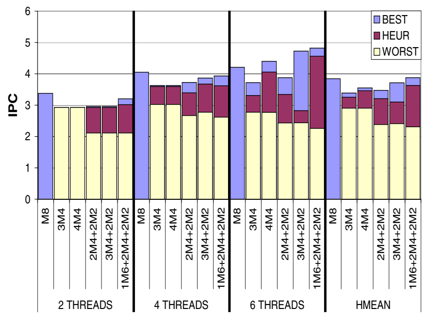
Fig. 4 shows the raw performance results (measured in IPC) for all microarchitectures evaluated. In each case, the harmonic mean of all workloads of a same type and size is shown. These results point out that, although some hdSMT results are quite similar to SMT baseline ones, the hdSMT results are exceeded by the SMT baseline ones in some cases. Comparing the baseline (M8) and best-performing hdSMT (1M6+2M4+2M2) means, we got baseline speedups over hdSMT of 5%, 4% and 15% in ILP, MEM, and MIX workloads respectively. In the first two cases, the mean performance of hdSMT is not quite bad considering that the hdSMT microarchitecture is able to execute up to 8 threads while the resource budget of the baseline (M8) in fact is not able to execute more than 4 threads (as mentioned in Section 3). Nevertheless, the ability to flush and re-execute instructions of the baseline is crucial in the MIX scenario. Although this is the general trend, notice that hdSMT is able to outperform the SMT baseline in the six-threaded ILP workload scenario (see Fig. 4.(a)).

The previous results strictly take into consideration the performance that each microarchitecture obtains executing the given workloads. However, each microarchitecture involves a different amount of resources; and a different power consumption among others. To make a fairer comparison we show in Fig. 5 the Performance per Area results for all microarchitectures evaluated. Again, the harmonic mean of all workloads of a same type and size is shown. From these results, we can infer that the hdSMT architecture achieves higher performance per area ratios than the monolithic SMT architecture, that is, better relative results than SMT using fewer resources. Comparing the baseline (M8) and best-performance-per-area hdSMT (2M4+2M2) means, we got hdSMT improvements over the SMT baseline of 15%, 18% and 10% in ILP, MEM, and MIX workloads respectively.

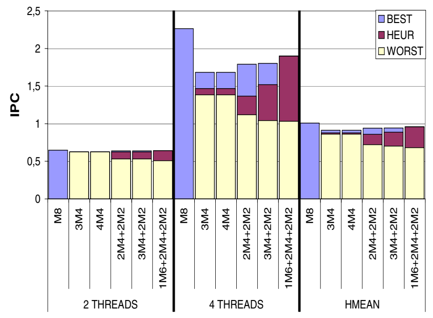
Regarding the homogeneous (3M4, 4M4) or heterogeneous distribution (2M4+2M2, 3M4+2M2, 1M6+2M4+ +2M2) of hdSMT processors, results in Figs. 4 and 5 point out that heterogeneous distributions are better than homogeneous ones. Thus, for each case there is a heterogeneous distribution that overcomes, both in terms of absolute performance and performance per area, all homogeneous distributions.

From all previous results it can also be inferred that the thread-to-pipeline mapping policy is a crucial factor in hdSMT architecture. This can be noticed by comparing the BEST and HEUR results in Figs. 4 and 5. As an example, notice that the 2M4+2M2 hdSMT microarchitecture obtains the highest performance per area ratios in all but the four-threaded MEM workload case. In that case, although the oracle mapping policy obtains a 9% improvement over the baseline, the heuristic accuracy drops to 76%, resulting in a worse result than the baseline. From Figs. 4 and 5 it is also noticeable that the effectiveness of the mapping policy depends on the specific hdSMT microarchitecture. Thus, while the heuristic applied in this work achieves 92% and 96% accuracy in 2M4+2M2 and 1M6+2M4+2M2 microarchitectures respectively, its accuracy drops to a 88% in 3M4+2M2 microarchitecture.

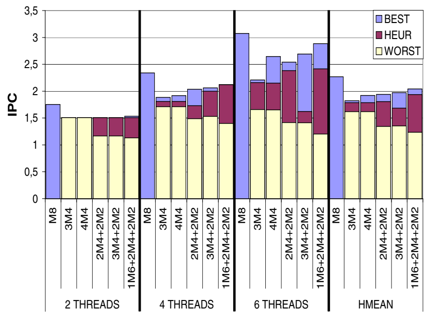
To summarize, our results point out that the hdSMT achieves its goal of minimizing the amount of wasted resources. In this sense, it obtains a 13% and 14% improvement in optimizing performance per area over monolithic SMT and homogeneously clustered SMT, respectively. Regarding to raw performance, monolithic SMT obtains in mean a 6% speedup over hdSMT. Nevertheless, hdSMT obtains in mean a 7% raw performance speedup over homogeneously clustered SMT. Finally, the results also indicate that the thread-to-pipeline mapping policy plays a very important role in hdSMT.



a) ILP Workloads.

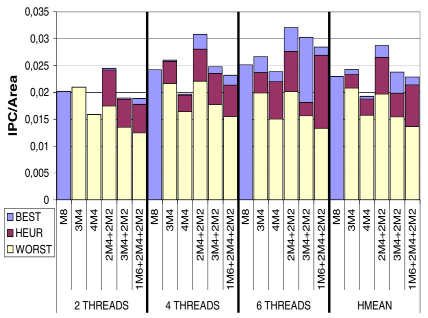


b) MEM Workloads.

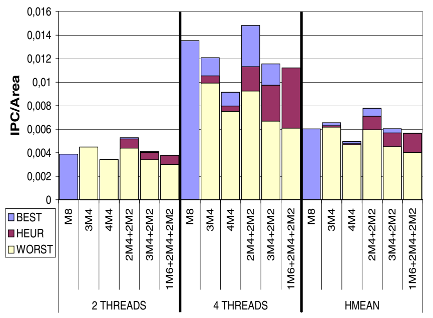


c) MIX Workloads.

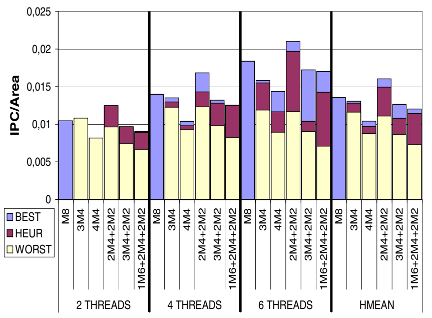
Fig. 4. Performance comparison.



a) ILP Workloads.



b) MEM Workloads.



c) MIX Workloads.

Fig. 5. Performance per Area comparison.

6 Related Work

Kumar et al. propose in [8] the Heterogeneous Multicore processor, a CMP processor comprised of heterogeneous cores. In this proposal, matching the inter-application heterogeneity with the statically partitioned hardware is, as in hdSMT, a prime issue. The main difference between them comes from their CMP/SMT inclinations. So, in case of few applications running on the processor, split resources accross CMP cores as physical registers and L1 caches are wasted in a Heterogeneous Multicore processor. On the contrary, an hdSMT can still utilize these resources to improve the performance of the running applications.

Putting aside heterogeneity-awareness, we find in the literature some prior work that study the combination of clustering and SMT techniques. Collins and Tullsen explore in [4] the relation between clustering and SMT. They show that the synergistic combination of the two techniques minimizes the IPC impact of the clustered architecture, and even permits more aggressive clustering of the processor than is possible with a single-threaded processor. In contrast, the hdSMT approach explores the benefits of a heterogeneous clustering to obtain high performance at a reduced complexity cost. Raasch and Reinhardt quantify in [12] the performance impact of resource partitioning policies in SMT machines, focusing on the execution portion of the pipeline. They found out that for storage resources, such as the instruction queue, statically allocating an equal portion to each thread provides good performance, in part by avoiding starvation. In contrast, they also showed that static division of issue bandwidth has a negative impact on throughput. SMTs ability to multiplex bursty execution streams dynamically onto shared functional units contributes to its overall throughput. As we have shown in this paper, a heterogeneous partition of issue bandwidth can be possitive if it is appropriatily matched with the heterogeneous needs of the running applications. The hdSMT approach also differs from these prior studies in the granularity applied in the cluster distribution. Notice that in hdSMT entire threads are assigned to pipelines, instead of the instruction level applied in these studies. So, dependent instructions are always executed in the same cluster, avoiding additional latencies and complexity.

Other proposals are more concerned in reducing the power consumption by means of the clustering technique. Thus, Latorre et al. [9] propose a multithreaded clustered microarchitecture as a way to deal with power consumption and wire delay problems. In this microarchitecture an evenly clustered front-end maps running threads to an evenly clustered back-end where the instructions are executed. This clustering also extends to resources such as L1 caches and register file, that are spread out the different clusters. Lee and Gaudiot also propose in [10] a symmetric, dual unified cluster SMT architecture as a way of reducing power consumption without significantly reducing its performance and throughput.

As far as we know, the hdSMT architecture is the first alternative SMT architecture in which the hardware is heterogeneously clustered in order to reduce the amount of wasted resources. By an appropriate matching of the heterogeneous applications with the heterogeneously distributed hardware, hdSMT achieves a better utilization of a reduced hardware budget, resulting in a better performance per area ratio.

7 Conclusions

The heteregeneity among application behaviors turns current architectures overdesigned for most cases, obtaining high performance but wasting a lot of resources to do so. In this paper we have presented the Heterogeneously Distributed Simultaneous Multithreading (hdSMT) architecture, an SMT alternative in which the running threads are mapped to a heterogeneosly clustered hardware according to this heterogeneity. The results obtained in this work indicate that hdSMT reduce this waste of resources at reduced budget, obtaining a 13% and 14% improvement in optimizing performance per area over monolithic SMT and homogeneously clustered SMT, respectively.

In hdSMT, the thread-to-pipeline mapping policy is a prime concern. In this work, we have presented a simple profile-based heuristic policy that achieves a 92% average accuracy. Raw performance results also point out that, in future hdSMT implementations, this mapping should probably be made dynamically in order to better adapt to the dynamic changes in program behaviour during execution.

Acknowledgements

This work has been supported by the Ministry of Education of Spain under contract TIN2004–07739–C02–01, CEPBA and the HiPEAC European Network of Excellence. Carmelo Acosta is also supported by the Ministry of Science and Technology of Spain grant BES–2002–0015.

References

[1] J. Burns and J.L. Gaudiot. Quantifying the SMT Layout Overhead - Does SMT Pull its Weight? In *Proc. of HPCA-6*, pages 109–120, 2000.

[2] J. Burns and J.L. Gaudiot. SMT Layout Overhead and Scalability. *IEEE Transactions on Parallel and Distributed Systems*, 13(2):142 – 155, February 2002.

[3] F. J. Cazorla, E. Fernández, A. Ramirez, and M. Valero. DCache Warn: An I-Fetch policy to increase SMT efficiency. In *Proc. of IPDPS-18*, pages 24–34, 2004.

[4] J. D. Collins and D. M. Tullsen. Clustered multithreaded architectures – Pursuing both IPC and cycle time. In *Proc. of IPDPS-18*, pages 46–57, 2004.

[5] A. El-Moursy and D. H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *Proc. of HPCA-9*, 2003.

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

[7] R. Kumar, K. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. of MICRO-36*, 2003.

[8] 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.

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

[10] S. W. Lee and J. L. Gaudiot. Clustered microarchitecture simultaneous multithreading. In *Proc. of EuroPAR-9*, pages 576–585, 2003.

[11] 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.

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

[13] 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*, pages 3–14, 2001.

[14] U. Sigmund, M. Steinhaus, and T. Ungerer. On Performance, Transistor Count and Chip Space Assessment of Multimedia-enhanced Simultaneous Multithreaded Processors. In *Proc. of MTEAC-4*, 2000.

[15] M. Steinhaus, R. Kolla, J. L. Larriba-Pey, T. Ungerer, and M. Valero. Transistor Count and Chip-Space Estimation of SimpleScalar-based Microprocessor Models. In *Proc. of WCED-2*, 2001.

[16] M. Steinhaus, R. Kolla, J. L. Larriba-Pey, T. Ungerer, and M. Valero. Transistor Count and Chip-Space Estimation of Simulated Microprocessors. In *T. R. UPC-DAC-2001-16, UPC*, 2001.

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

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

[19] 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.

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