UNIVERSITAT POLITÈCNICA DE CATALUNYA

PHD THESIS
2009

Heterogeneity-Awareness in Multithreaded Multicore Processors

Author:

Carmelo Alexis Acosta Ojeda

Advisors:

Mateo Valero Cortés
Universitat Politècnica de Catalunya

Alex Ramirez Bellido
Universitat Politècnica de Catalunya

Francisco J. Cazorla Almeida
Barcelona Supercomputing Center

A thesis submitted in fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

A mis padres, Purita y Carmelo, y a Hema,
por todo el amor que me dan día a día.

Agradecimientos

Esta tesis doctoral ha supuesto uno de los hitos más importantes de toda mi vida. A lo largo de todos estos años he madurado no solo como profesional sino como persona. Como en la vida misma, ha habido un poco de todo, momentos buenos y momentos malos. Incluso, como no, ha habido momentos en los que parecía que nunca llegaría este día. En ocasiones no somos capaces de ver la luz al final del túnel y nos da la sensación de estar solos y perdidos vagando a la deriva. En esos momentos te planteas el sentido de todo, incluso de la vida misma. Hoy echo la vista atrás lleno de alegría y pienso que todo tiene su razón de ser, lo bueno y lo malo. Las grandes lecciones de la vida suelen costar un poco, pero una vez aprendidas te permiten disfrutar con mayor ilusión y alegría de las cosas buenas que ésta tiene que ofrecerte . . . si estas atento para que no se te escapen.

En este sentido quiero dar mi más profundo y sincero agradecimiento a mi futura esposa Hema, el amor de mi vida, y a mis padres, Purita y Carmelo, mis referentes en la vida que lo son todo para mí. Gracias por haber estado siempre ahí, apoyándome y confortándome en los momentos malos, y dándome tanto amor como me dais y me seguís dando. Desde lo más profundo de mi corazón . . . gracias, os quiero.

Esta tesis doctoral me ha cambiado la vida, en todos los sentidos. Empezó por hacerme cambiar de ciudad de residencia, propiciando mi emancipación. Si bien 24 años es en general una buena edad para emanciparse y "dejar por fin tranquilos a los padres", hay que reconocer que con los tiempos que corren no suele ser lo habitual. Tener que mudarme de Las Palmas de Gran Canaria, la ciudad que me vio nacer y a la que siempre llevo en el corazón, a Barcelona, una gran urbe muy lejos de todo y todos los que conocía, supuso un gran reto. Doy las gracias en este sentido a Enrique Fernández y a Mateo Valero por creer en mí, ayudarme y permitirme vivir esta experiencia.

Por suerte, el hecho de dejar atrás tanto y a tantos, incluyendo a mi paciente pareja, se hizo más llevadero gracias a la gente tan estupenda con la que me encontré en Barcelona. A algunos ya los conocía de antes, como a Xavi Verdú y Fran Cazorla, dos buenos amigos a los que conocí haciendo la carrera de Informática en la ULPGC.

La vida siempre te depara sorpresas y muchas de ellas son incluso agradables. Así, si bien dejé atrás muchos amigos en Barcelona conocí a muchas personas excepcionales, entre las que me enorgullece decir que encontré grandes amistades. Siempre recordaré con nostalgia y una sonrisa en los labios aquellas partidas al *Quake* con Ayose Falcón, Oliver Santana, Daniel Ortega, Jesús Corbal, Fernando Latorre, Llorenç Cruz y Ramón Canal. Más adelante, el *Quake* daría paso al *Need for Speed* y al *Worms* y nuevos "contendientes", como Jaume Abella, y otros no tan nuevos como Marco Galluzzi, Ale García, y Tana se unirían a esos momentos especiales que compartíamos y que hacían las jornadas de trabajo más llevaderas. En este sentido, contar con Tana, amigo de toda la vida al que "medio convencí" para venirse a Barcelona, supuso un gran apoyo para sentirme más "como en casa". Gracias por estar ahí.

Recuerdo también con especial añoranza aquellas gratas sobremesas, primero en el Nexus y más tarde en la cocina del departamento, tanto en su antigua ubicación en el edificio D6 (actual sala de impresoras) como en su nueva ubicación en el C6. A todos los anteriores he de unir en la lista a Josep María "Josepe" Codina, Enric Gibert, Jordi Guitart, Alex Pajuelo y Rubén Gran. A todos ellos doy las gracias por hacer de mi paso por el DAC una experiencia tan agradable y memorable.

Y como olvidarme de esos cafés y paseos en los que hablábamos un poco de todo, desde métricas de simulación hasta filosofía de la vida. Seguramente me dejaré a muchos y muchas en el tintero, pero la lista no puede prescindir de figuras como Germán Rodríguez, Isidro González, Gemma Reig y Miquel Moretó. Agradezco también a mis "profesores particulares" de catalán . . . gracias Josepe Codina y Miquel Moretó por ser tan pacientes con mi incipiente catalán. En este apartado quiero hacer una mención especial para una gran amiga que encontré en Barcelona. A ella le debo mucho, pues como buena amiga supo confortarme en los momentos difíciles y ayudarme a superarlos. Gracias Bea Otero por estar ahí y poder contarte entre mis amigas.

Aunque no lo parezca, entre partidas de videojuegos, charlas en la sobremesa y cafés también hubo tiempo para investigar y sacar adelante una tesis doctoral. Llegados a este punto, quiero dar mi más sincero agradecimiento a mi director de tesis, Mateo Valero, por creer en mí desde el principio, incluso cuando ni siquiera yo mismo creía en mí. Esta tesis no hubiera sido posible sin él y desde estas líneas quiero expresar mi gratitud. Profesionalmente, soy lo que soy gracias a él, así que . . . muchas gracias, Mateo.

No quiero olvidar mi agradecimiento a mis co-directores de tesis, Alex Ramírez y Fran Cazorla. A lo largo de todo este tiempo juntos he aprendido mucho de ambos. También quiero agradecer en este sentido a Ayose Falcón por su aportación a mi tesis en su etapa como co-director de la misma. Como profesional, he madurado mucho junto a tan buenos investigadores. Parte de lo que soy hoy también se lo debo a ellos.

Pero como decía desde un principio, esta tesis doctoral ha supuesto grandes y profundos cambios en mi vida, no solo en el aspecto profesional. Con los años he llegado a considerar a Barcelona como “mi hogar” (de hecho ya me he establecido aquí con mi pareja con la que ya he pasado por el rito-bancario de “ . . . hasta que el Euribor nos separe . . . ”). Ambos hemos tenido la suerte de encontrarnos con bellísimas personas que han conseguido hacernos sentir como en casa. Ahora somos canario-catalanes y orgullosos de serlo. Entre la lista de personas que han hecho más agradable nuestra estancia en Barcelona me gustaría citar, entre otros, a Sara (Saray) Guardias, Alba Tizón, Ana Belén Rodríguez, Ángel Melgar, Pilar Boira y Trini Carneros. Tanto a los que aquí aparecen como a todos y todas los demás, gracias por hacer más amena esta etapa de mi vida. Sin vosotros no hubiera sido lo mismo.

Afortunadamente, “emigrar” a Barcelona no supuso perder a los amigos de Canarias solo reduce la cantidad de momentos que puedes compartir con ellos. A lo largo de estos años, en los “ires-y-venires” Barcelona-Gran Canaria, he seguido manteniendo dichos amigos y tengo la suerte de poder decir que incluso he podido incrementar ese número. Me gustaría citar entre otros a Selene, Davinia, Mónica y Zaida Cabello, Jony, Dani Montelongo, Mónica Suarez, Ana, Jaime, Dani, Ruti, Raquel y Rosi. Gracias a todos y todas por hacer de cada viaje a Canarias a lo largo de estos años un momento inolvidable. Ocupa un lugar destacado en esta lista de amigos Román, algo más que un amigo . . . si fuéramos hermanos seguro que no nos llevábamos tan bien. Gracias por estar siempre ahí . . . como espero que estés en una celebración que tenemos aún pendiente, “padrino”.

Y no puedo terminar sin agradecer entre otros a Stan Lee, por enseñarme que independientemente de la ropa que lleves, si eres bombardeado por rayos gamma y te hacen cabrear, siempre acabarás con unos pantalones cortos azules/violeta “irrompibles”. A J.R. Tolkien bueno, más bien a Peter Jackson que al otro nunca lo tragué demasiado, por recordarme que nunca compre un anillo en la Tierra Media, pues las devoluciones son complicadas (Presentar una solicitud en persona a Monte del Destino S/N, TM0 Mordor, Tierra Media). Y finalmente a George Lucas, por enseñarme que no todo aquel que es seducido por el lado oscuro frecuenta los locales de “ambiente”.

Abstract

As enter into the so-called *Billion Transistor Era*, with billions of transistors on a single chip, the Computer Architecture faces new challenges. The performance achievable by traditional superscalar processor designs does not scale with the increasing transistor count due to limitations imposed by the *Instruction Level Parallelism (ILP)*. As a consequence, *Thread Level Parallelism (TLP)* has become a common strategy for improving processor performance. Since it is difficult to extract more *ILP* from a single application, multithreaded processors focus on the processor throughput, executing multiple applications instead. Obviously, multiple execution threads coming from a single application may be simultaneously executed, but sometimes it is not that trivial exploiting *ILP*: we can not simply rely on *Parallel Programming*. As the number of cores on a single chip increases, the Computer Architecture community wonders whether this new trend towards having hundreds of cores on a single chip, also called *many-cores*, is worthwhile.

The complexity of state-of-the-art designs is translated into power and thermal challenges. Power efficiency can often be traded for performance or cost benefits. With the increasing power density of modern circuits, as the number of transistors per chip scales (Moore's Law), power efficiency has increased its importance. Thus, current processor designs must be *complexity-effective*: i.e. *get the highest throughput possible with the lowest power consumption*. In addition, power dissipation issues constrain the designs of the next processor generations. The quest for simpler ways of increasing the processor throughput under a reasonable power cost is on the way.

In this thesis we analyze the heterogeneity in the behavior of applications and match it with the processor design itself. We show that this heterogeneity turns current general-purpose processors overdesigned for most cases. We also show that current *multithreaded multicore (CMP+SMT)* processors are not explicitly aware of this software heterogeneity, that is they are not *Heterogeneously-Aware*. We propose architectural changes in order to turn *Heterogeneously-Aware* the *CMP+SMT* processors. Our proposals strive to improve the *complexity-effectiveness* of future generations of *CMP+SMT* processors.

Contents

Chapter 1

Introduction

The process technology advances are propelling the computer industry towards the so-called *Billion Transistor Era*. Optical and lithographic improvements allows that every two or three years the industry produces a new level of manufacturing technology that shrinks die area by a factor of two for the same number of transistors. Figure 1.1 shows the feature size and gate lengths of various processes Intel expects to put into production every two years through this decade. In addition, the size comparison of these features and the influeza virus is shown in Figure 1.1 as an illustrative example of the process technology's potential.

The arrival of the *Billion Transistor Era* is also impeled by the development of new materials in the industry. Figure 1.2 shows a Transmission Electron Microscope (TEM) photo. The left side depicts a closeup of a transistor in Intel's 90nm process. The image is really only about 1/10th of the actual channel length. The little round structures in Figure 1.2 are atoms, and they are only about 0.3 nanometers apart in the silicon substrate, which has a highly regular structure. The size of the insulating SiO_2 gate dielectric layer is only about 4 or 5 atomic thick. While transistors get faster at these smaller dimensions, leakage current becomes a much greater problem, and the gate structures become much more fragile. The right-hand side uses a new insulating material with a higher dielectric (K) value, and it can be much thicker and stronger, yet still maintain the same fast electrical properties as the SiO_2 while reducing gate leakage by two orders of magnitude.

From a Computer Architecture's point of view, the future does not look so promising. Since the appearance of the first *Superscalar* design in the 60's, computer architects have striven to employ the increasing hardware resource count to boost the performance of applications. Thus, many processors exploit *Instruction Level Parallelism (ILP)* to execute

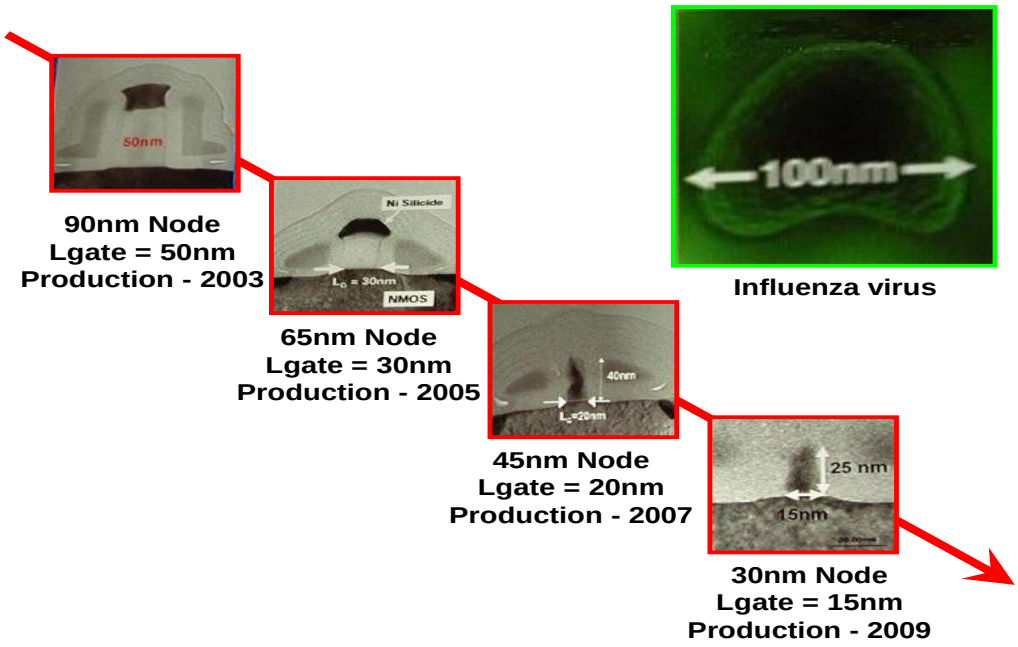
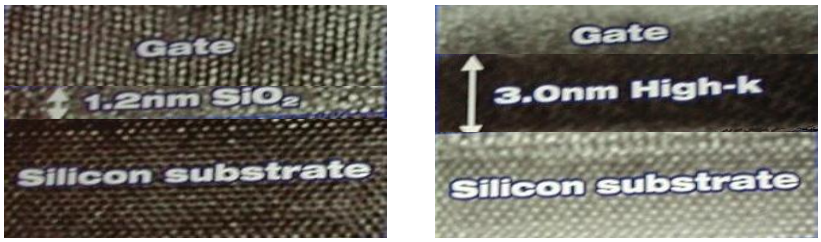


Figure 1.1: Process Advancements Fulfill Moore's Law.



	90nm process	Experimental high-k
Capacitance	1x	1.6x
Leakage	1x	<0.01x

Figure 1.2: Nanotechnology Gate Dielectrics.

several instructions from a single stream (thread) in parallel. However, there is only a limited amount of parallelism available in each thread due to *data* and *control dependences*, among other factors [76] :

1. *Control dependences*: every time a control flow instruction changes the flow of instructions to a new target instruction, it takes several cycles to start fetching from that target, which degrades the number of *Instructions committed Per Cycle (IPC)*.

2. *Data dependences*: data dependences limit the *IPC* as well since an instruction can only start its execution when all its input dependences are resolved. For short-latency operations the out-of-order mechanism of current *Superscalar* processors hides part of this latency. However, when the processor experiences a long-latency operation, i.e., a miss in the outer cache level, this mechanism is not able to hide it causing a stall of the processor. Literature claims these dependences to comprise probably the ultimate frontier of Parallelism: the *Memory Wall*.

Computer architects use many hardware resources in order to reduce the effect of these problems, e.g., bigger and more complex branch predictors to control dependences and deeper windows to further exploit *ILP* when a long-latency instruction is executed. However, *data* and *control dependences* significantly limit performance, degrading the performance/cost ratio of processors.

Since it is difficult to extract more *ILP* form a single program, architects opted for execute multiple programs. Thus, *Thread Level Parallelism (TLP)* rapidly became a common strategy for improving processor performance. *Multithreaded (MT)* processors constitute a solution to improve the performance/cost ratio of processors, allowing threads to share hardware resources. There are several categories of *MT* processors, each dealing with the above problems in a different way. The classification of *Multithreaded* processors is not well established. In this thesis, we have used a classification similar to that presented in [73, 74], as explained in Figure 1.3. In this figure, A, B, C and D represent four different applications. White squares denote unutilized slots.

1. In a *Superscalar* architecture, like the *Intel Pentium III* [3], only one thread is running at a given time.

2. In a *Multicore* processor, like the *Intel Core 2 Duo* [77], resources are not shared between threads[1]. Each thread uses a different set of resources.

3. In a *Coarse-Grain Multithreaded* processor [10, 66], like the *IBM Northstar/Pulsar* [1] , threads share more execution resources than in a *Multicore* processor. Instructions

[1]These applications likely share some levels of the cache hierarchy.

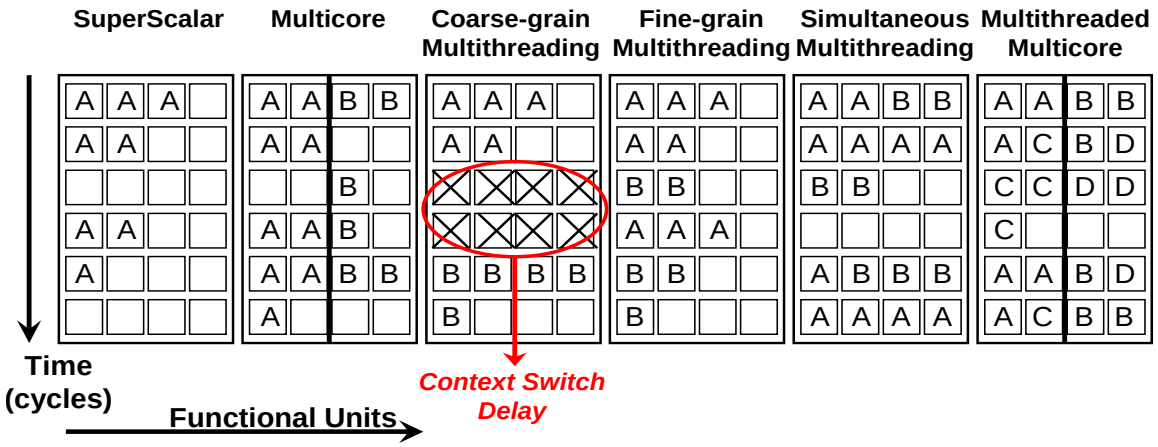


Figure 1.3: A possible classification of Multithreaded Architectures.

can be issued from a single thread in a given cycle. A *Coarse-Grain* processor switches to a different thread when a thread experiences a long-latency operation, e.g., an outer cache miss. This allows the processors to hide part of the latency of long-latency operations.

4. *Fine-Grain Multithreading* [12, 28, 36, 61]: The main difference between *Coarse-Grain* and *Fine-Grain Multithreading* is the granularity at which context switches occur. In a *Fine-Grain Multithreaded* processor context switches are caused by other, not necessarily long-latency, events, e.g. branch misprediction. In this way, *Fine-Grain* processors can hide the latency of short-latency operations. Another difference between *Fine-Grain* and *Coarse-Grain Multithreading* is that the latter approach switches between threads much more frequently than the former approach. As a result, in *Fine-Grain Multithreading*, like the *Sun UltraSparc T1* [7] and *T2* [4], context switches have lower cost (probably 1 cycle) than in *Coarse-Grain Multithreading*.

5. The main characteristic of *Simultaneous Multithreading (SMT)* processors [30, 43, 72, 71, 79], like the *Intel Pentium 4* [5], is their ability to issue instructions from the different threads in the same cycle. As a result, *SMTs* not only can switch to a different thread to use the idle issue cycles in a short-latency operation (like *Fine-Grain Multithreaded*), but also fill unused issue slots in a given cycle. Executing several threads at the same time provides *TLP* in addition to *ILP*. This parallelism comes from the additional parallelism provided by the freedom to fetch instructions from different independent threads, and from mixing them in an appropriate way in the processor.

6. The *Multithreaded Multicore* processors, like the recent *IBM POWER5* [60] and *POWER6* [39], represent the lattest incorporation to the *MT* group. The increasing transistor count on die has made possible to build a *Multicore* processor in which each of its execution cores implements *SMT* feature. The example on the right side of Figure 1.3 shows a 2-core implementation with 2-hardware contexts per core. The private hardware resources within each execution core are shared among the two applications running simultaneously.

Regarding *control dependences*, *MT* processors reduce the dependence of throughput on branch prediction accuracy. That is, branch prediction accuracy is not of the utmost importance when running multithreaded applications [47, 53]. This is mainly due to the fact that the opportunity of fetching from several threads reduces the percentage of speculative instructions on a wrong path [67].

Regarding *data dependences*, *MT* processors have shown to be successful in reducing the effect of data dependences [24, 42, 70]. This is due to the ability of *MT* processors to execute instructions from several threads[2].

Given all these advantages of *MT* processors, current trends in Computer Architecture show that forthcoming processor generations will involve some form of multithreading [11, 41]. In fact, many of the main processor vendors already have some multithreaded processors. Some examples are the *Intel Pentium 4* [5], a dual thread *SMT*, the *Intel Core 2 Duo* [77], a dual core processor, the IBM *POWER5* [60] and *POWER6* [39], dual core processors comprised of 2-context *SMT* cores, and the *Sun Niagara T1* [7] and *T2* [4], with eight 4 and 8-context *Fine-Grain Multithreaded* cores respectively.

The processor generation's state-of-the-art also reveals a trend towards increasing the exection core count on a single chip [69]. Potentially down the road, assuming a continued trajectory, the current trend could lead to the development of a massive core future whereby one chip could contain thousands of processing cores. We would then jump to a new step in the *MT* roadmap: The *Many-Core* Processors.

With this sea change in the architecture of the hardware, we are witnessing the Software Community wrestling with a massive shift from serial-based thinking to parallelism. However, the general feeling in the Software Community reveals a quite blunt and negative reaction to this grandiose trajectory that the Hardware Community has set in motion. With this type of feedback coming from the Software Community, could it be we are witnessing the end of an era?

[2] *MT* is orthogonal to the out-of-order mechanism of the processor, if it exists.

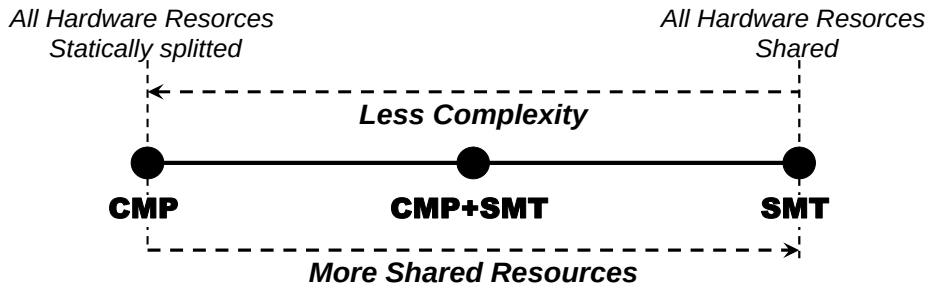


Figure 1.4: A continous spectrum of Multithreaded approaches.

As Computer Architects we can not unilaterally decide the future of the whole Computer Community. Sometimes we have to sit back and consider alternative ways of reaching our goal; ways that take into account the perspective of other communities. In this sense, if we have a look at the executed applications it is straightforward that they have an *heterogeneous* behavior, as we will see in Chapter 3. This *heterogeneity* can be found comparing the behavior of both different applications and different portions of execution within the same application. It is then logical that we should start designing processors explicitly aware of this *heterogeneity* in the software they execute. We call this new approach *Heterogeneity-Aware Architectures*.

1.1 Simultaneous Multithreading and Multicore Processors

Simultaneous Multithreading (SMT) [71, 72, 79] and *Multi-Core* processors, or *Chip Multiprocessors (CMP)* [48, 29], represent opposity edges of the same continuous spectrum, as shown in Figure 1.4. 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 execution 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 [21].

Multithreaded Multicore (CMP+SMT) processors represent a new trend in industry.

The advances in process technology have made possible to replicate multiple execution cores with *SMT* facilities on a single chip. In this processors, the whole transistor count is splitted among all the constituent cores, reducing the overall complexity of the chip. However, each of these cores can simultaneously execute multiple threads in order to boost the throughput of each core. The use of *SMT* within each core allows to increase the resource budget of the execution cores without severely increase the possibility of resource underuse. Notice that while *CMPs* mainly rely on *TLP SMTs* help to balance both *TLP* and *ILP*.

In this thesis it is explored the continous spectrum shown in Figure 1.4. On the one hand, new architectures are proposed that lay on the same spectrum but emphasize the complexity-effectiveness of the processor design. On the other hand, some proposals are given to improve the performance of current and future *CMP+SMT* processors. It is also detected a potential hazard of current *CMP+SMT* designs; a solution that does not involve excessive complexity is proposed.

1.2 Heterogeneity-Awareness

As mentioned earlier, the behavior of the applications is inherently heterogeneous. We deeply analyze this heterogeneity in typical general-purpose workloads in Chapter 3. In advance, we could say that different behaviors can be identified comparing both different applications and different portions of the same applications' execution. Consequently, the hardware support required for each application may vary as applications exhibit different behaviors. However, current architectures are designed for the common case. Homogeneous designs hold sway in the current state-of-the-art, like the *Intel Core 2 Duo* [77] and the IBM *POWER5* [60] and *POWER6* [39]. However, some vendors have already realized the benefits of heterogeneous microprocessors. Thus, the *Cell* [49] processor, first released in 2005[3] and used in the *PlayStation 3* video game console, is comprised of 1 master *PowerPC* processor that feeds 8 slave *SIMD* accelerators, that make extensive use of the *Altivec ISA*.

In this thesis the *Heterogeneity-Awareness* is a key factor in the processor design. As far as we would be able to identify heterogeneous behaviors in applications and match them with the appropriate amount of resources, it is possible to envision more complexity-effective processors. In this sense, we give proposals built on top of both heterogeneous and homogeneous hardware layouts. In all cases, the objective is the same: consume less power maintaining a similar performance level than bigger and more complex machines.

[3]This thesis started in 2003.

1.3 Thesis Contributions

The main contribution of this thesis is that we introduce for the first time the concept of *Heterogeneity-Awareness* in *Multithreaded Multicore* processors; as well as mechanisms that make use of this concept to yield both more complexity-effective and productive machines. We define as *Heterogeneity-Awareness* the processor feature that explicitly takes into account the heterogeneity in the behavior of the running applications. This heterogeneity in software is matched with an heterogeneous hardware, or heterogeneous application assignment over an homogeneous hardware layout. The better the matching the better the results, since we would be assigning each application exactly the amount of resources needed according to its requirements during that time slice.

The main purpose of this thesis is to explicitly reflect the *Heterogeneity-Awareness* concept in the design of the *Multithreaded Multicore* processors, with a twofold objective. First, to improve the *complexity-effectiveness* of current and future designs, in order to fulfill the harder power and thermal constraints that industry is leading Computer Architecture to. Second, to improve the throughput obtained in both current and future *complexity-effective* processors. Being aware of the heterogeneity in the software executed allows to react accordingly, improving the performance of available resources by performing a better resource sharing; that is, giving each application exactly the hardware resources needed for each time slice's requirements.

We show that by reflecting the heterogeneity in software on the hardware itself, and performing the appropriate matching, it is possible to achieve our first objective, namely to improve the *complexity-effectiveness* of current and future designs. Moreover, it is possible to improve the performance of a purely homogeneous *SMT* machine appropriately distributing the workload among the available homogeneous resources on the hardware. We also envision the gradual transition of current state-of-the-art homogeneous *CMP+SMT* processors to future heterogeneous *CMP+SMT* processors, in which the *Heterogeneity-Awareness* allows greater improvements in terms of both complexity-effectiveness and throughput. The full list of the contributions of this thesis is enumerated following:

1. *The hdSMT Architecture*. To accomplish complexity-effectiveness in hardware designs we combed some wide regions of the continuous spectrum that lie in between the *CMP* and *SMT* approaches. We first made an exhaustive analysis of the heterogeneity in hardware and its relation to software. Then, we employed the conclusions of this analysis to establish the foundations of the *heterogeneously distributed SMT (hdSMT)* architecture, that allows to improve the *complexity-effectiveness* of the processor design. We show that the proposed *hdSMT architecture* has pretty much potential than current *monolithic SMT* processors.

2. *The TCA Algorithm*. Since the hardware distribution proposed by our first contribution, the *hdSMT* architecture, may look difficult to be handled by current CAD/CAM and layout verification processes, we then moved to a more feasable layout: a *CMP+SMT* processor, using the *IBM POWER5* [60] as point of reference. The lack of *Heterogeneity-Awareness* in an homogeneous *CMP+SMT* processor, comprised of *SMT* cores with 2 hardware contexts, generally turns into a throughput degradation. Although its hardware does not directly reflect the *Heterogeneity-Awareness* concept, as the *hdSMT* does, it is posible to add slight modifications that turned such a processor into a more *Heterogeneity-Awareness* machine. In this sense, we proposed one of the main contributions of this thesis: the *Thread to Core Assignment (TCA) Algorithm*. Involving a negligible overhead, the *TCA Algorithm* boosts the throughput of current and future *CMP+SMT* processors by explicitly exposing the heterogeneity in software to the hardware, and appropriatelly matching them. We show evidences that confirm the *TCA Algorithm* supposes a quite significative *Heterogeneity-Aware* improvement for state-of-the-art processors.

3. *The hTCA Algorithm*. Once shown that even state-of-the-art homogeneous *CMP+SMT* processors may be improved turning them *Heterogeneity-Awareness*, we envision the next straightforward step in processor designs. Thus, once our processor is *Heterogeneity-Awareness*, by means of a proper management of the *TCA*, we introduce some amount of heterogeneity in the hardware itself. This additional heterogeneity is aimed at allowing a better matching between software requirements and hardware facilities. In this sense, we propose the *heterogeneous TCA (hTCA) Framework*. Involving some minor hardware additions and assited by an *hTCA Algorithm*, the hTCA Framework proves to expose the *complexity-effectiveness* to the user, being possible to dynamically decide the *degree of complexity-effectiveness* in our executions.

4. *The MFLUSH mechanism*. Finally, we analyze further considerations arised while moving from single-core to multi-core scenario. We realized that some well-known *SMT* techniques were altered in this transition. In particular, the *FLUSH* [70], mechanism proves to yield worse results than the *ICOUNT* [72] policy[4]. As a solution, we proposed the *MFLUSH* mechanism, an *Heterogeneity-Aware* mechanism that yields good results in current and future *CMP+SMT* processors. The *MFLUSH* mechanism adapts the *FLUSH/STALL* philosophy to a highly variable multithreaded multicore scenario, adapting its response according to the memory banks and traffic contention.

[4]Built on top of *ICOUNT*, the *FLUSH* mechanism was developed to improve the *ICOUNT* response to long-latency loads, which degrade its throughput.

Although not considered as a thesis contribution itself, I would like to emphasize the special effort put on the *MPsim*, a highly-flexible Simulation Tool specifically designed for this PhD dissertation, that allows to cover a very wide design space. Such a tool is desirable in order to face up the researching of coarse regions of the continuous spectrum between the *CMP* and *SMT* approaches. The *MPsim* has evolved throughout the whole thesis and continues evolving. It already has gone beyond the scope of this PhD dissertation, becoming the main tool used by a group of researchers spread over the Computer Architecture Department (DAC) of the Polythecnic University of Catalonia (UPC), the Barcelona Supercomputing Center (BSC) and the University of Las Palmas de Gran Canaria (ULPGC). A detailed description of the *MPsim* Simulation Tool can be found in the Apendix.

1.4 Thesis Structure

This thesis is organized in chronological order, in a similar fashion as the research was done. The only exception is the *MPsim* simulator, which was evolving (and continues evolving) as the thesis proceeded. Regarding the *MFLUSH* policy, it was developed in parallel to *TCA Algorithm* and *hTCA Framework*. It raised from the observation of the poor results obtained in our first simulations of the *FLUSH* policy in multithreaded multicore scenarios.

We started analyzing the design space that lays in between *SMTs* and *CMPs*. From this analysis we identified the main problems to be faced up by the introduction of the *Heterogeneity-Awareness* concept in current architectures. This was done firstly from an *SMT-like* approach and later from a *CMP-like* approach, both converging to an intermediate point in the continuous *SMT-CMP* design space. The final idea is to improve state-of-the-art processors by introducing the *Heterogenity-Awareness* concept in both software and hardware.

This thesis is structured as follows:

1. *Chapter 2* is devoted to explain our experimental environment. This includes both the simulation tools and the benchmarks used in this thesis. Since each specific experiment throughout the whole research may have specific methodology issues, we cover here the common methodology issues, postponig

   Among the simulation tools used we put special emphasis on the *MPsim* simulator, since this tool has covered the whole research and continues evolving with a long-term life expentacy.

2. *Chapter 3* defines the *Heterogeneity-Awareness* concept in detail. This concept emerges from a deep analysis into the heterogeneity exhibited by current[5] applications and their relation with the main processor resources. This chapter also shows our first contribution to meet the *Heterogeneity-Awareness* concept : the *hdSMT Archicteture*.

3. In *Chapter 4* we analyse the main challenge faced up by state-of-the-art *CMP+SMT* processors, like the *IBM POWER5* [60] and *POWER6* [39], which are not *Heterogeneity-Aware*. We then identify the need of a new layer in the OS scheduling process in order to make *CMP+SMT* processors *Heterogeneity-Aware*. Finally, we propose our second contribution, the *TCA Algorithm*, as candidate to manage the additional scheduling layer in *CMP+SMT* processors.

4. *Chapter 5* presents the last *Heterogeneity-Aware* contribution of this thesis: the *hTCA Framework*. In this chapter we envision the next straightforward step in processor designs, that is moving to *Heterogeneity-Aware* Architectures with an heterogeneous layout. The *hTCA Framework* represents a first step into a new generation of *Heterogeneous* and *Heterogeneity-Aware* processors, in which the *complexity-effectiveness* involved into the *Resource Sharing* step of the *OS Scheduling Process* is explicitly exposed to the user. The user can then dynamically specify the desired *degree of complexity-effectiveness*.

5. In *Chapter 6* we analyse the main challenges faced up when moving from single-core *SMT* processors to *Multithreaded Multicore (CMP+SMT)* processors, as seems to happen nowadays according to the current trend in industry. In parallel to both the TCA Algorithm and hTCA Framework we identify the need of more *Heterogeneity-Awareness* in well-known *SMT* Instruction Fetch policies when applied to the new multithreaded multicore scenario. In particular, we propose the *MFLUSH* policy as a solution to the static response to a highly-variable multithreaded multicore scenario of prior *FLUSH* [70] *SMT* policy.

6. *Chapter 7* shows the conclusions of this thesis.

[5] By the year 2003, when this thesis started, the most referenced benchmarks in General-Purpose Computer Architecture came from the *SPEC2000 Benchmark Suite*. We do believe that the conclusions obtained during this PhD dissertation may be applied to more recent benchmark suites. However, such a verification is left for Future Work.

Chapter 2

Experimental Framework

This chapter is devoted to explain the evaluation tools we have used in order to analyse the design space and evaluate our proposals. We show the benchmarks used for that purpose as well.

2.1 Simulation Methodology

During the research covered by this thesis a great number of experiments were performed. Each of these experiments involved thousands of simulations, each one comprising several hundreds of millions of simulated instructions. As a consecuence, it was critical to reach some commitment regarding the computational cost constraints. In this sense it was decided to opt for the *trace-driven* simulation methodology, to be employed in our experiments.

In order to benefit from the *trace-driven* reduced computational cost, without severely compromising the accuracy of the results obtained, the simulation tool was adapted accordingly. Thus, the simulator employed 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.

Benchmark name	Remarks	Input	Language	Fast Forward (Millions)
164.gzip	Data compression utility	graphic	C	68.100
175.vpr	FPGA circuit placement and routing	place	C	2.100
176.gcc	C compiler	166.i	C	14.000
181.mcf	Minimum cost network flow solver	inp.in	C	43.500
186.crafty	Chess program	crafty.in	C	74.700
191.parser	Natural language processing	ref.in	C	83.100
252.eon	Ray tracing	cook	C++	57.600
253.perlbmk	Perl	splitmail.535	C	45.300
254.gap	Computational group theory	ref.in	C	79.800
255.vortex	Object Oriented Database	lendian1.raw	C	58.200
256.bzip2	Data compression utility	inp.program	C	13.500
300.twolf	Place and route simulator	ref	C	324.300

Table 2.1: FastForward used for each Spec INT 2000 Benchmark.

2.2 Benchmarks

In the experiments performed during this research we use the *SPEC2000 benchmark suite*[1]. From them we collected traces of the most representative 300 million instruction segment of each benchmark, following the idea presented in [55]. Whenever a benchmark is used more than once in a single workload, each additional instance is forwarded 1 million instructions more than the prior one (marked with a $+1$ in the workload definition). 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. Fortran programs are compiled with the DIGITAL Fortran 90/Fortran 77 compilers. The fast forwards applied to each application, in order to obtain the traces, are shown in Tables 2.1 and 2.2.

In the study of the workloads's heterogeneity benchmarks from the *SPEC2000 benchmark suite* are divided into two groups based on their cache behavior, as shown in Table 2.3. Since we employ a great variety of processor and memory configurations in our experiments we defined a single and easy-to-handle benchmark classification. To establish such a classification we use for each benchmark the results of a single-thread execution in a typical superscalar processor with an L2 cache of 512 KB. This L2 Cache size comes from the observation of an state-of-the-art processor like the *IBM POWER5*, that have four *SMT* hardware contexts and a shared L2 cache of approximately 2MB.

[1]By the year 2003, when this thesis started, the most referenced benchmarks in General-Purpose Computer Architecture came from this benchmark suite. Due to the analysis of the applications involved, we did not migrate to the next release in 2006.

Benchmark name	Remarks	Input	Language	Fast Forward (Millions)
168.wupwise	Quantum chromodynamics	wupwise.in	Fortran77	263.100
171.swim	Shallow water modeling	swim.in	Fortran77	47.100
172.mgrid	Multi-grid solver in 3D potential field	mgrid.in	Fortran77	187.800
173.applu	Parabolic/elliptic partial differential equations	applu.in	Fortran77	10.200
177.mesa	3D Graphics library	frames100 + msea.in	C	294.600
178.galgel	Fluid dynamics: analysis of oscillatory instability	gagel.in	Fortran90	175.800
179.art	Neural network simulation; adaptive resonance theory	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10	C	13.200
183.equake	Finite element simulation; earthquake modeling	inp.in	C	27.000
188.ammp	Computer vision: recognizes faces	ammp.in	C	13.200
189.lucas	Computational chemistry	lucas2.in	Fortran90	30.000
191.fma3d	Finite element crash simulation	fma3d.in	Fortran90	10.500
200.sixtrack	Particle accelerator model	sixtrack.in	Fortran77	173.500
301.apsi	Solves problems regarding temperature, wind, velocity and distribution of pollutants	apsi.in	Fortran77	192.600

Table 2.2: FastForward used for each Spec FP 2000 Benchmark.

Consequently, for single-thread executions we used a quarter of the L2 cache size of the *IBM POWER5*, that is 512KB. From the results obtained after simulating 300 millions of instructions selected according to the idea presented in [55], and according to the benchmark taxonomy applied in [20], we classify each program as *Memory Bounded (MEM)* whether its L2 cache miss rate is greater or equal than 1%, *CPU Bounded (ILP)* otherwise. The L2 cache miss rate is calculated with respect to the number of dynamic loads. According to the constituent benchmarks, we denote each workload as *MEM* or *ILP* whether all their benchmarks belong to the respective group. In presence of both *MEM* and *ILP* benchmarks we denote the resulting workload as *MIX*.

Benchmark type	Benchmark name	L2 cache miss rate
INTEGER	mcf	29.6
	twolf	2.9
	vpr	1.9
	parser	1.0
	art	18.6
FP	swim	11.4
	lucas	7.47
	equake	4.72

(a) Memory bounded benchmarks

Benchmark type	Benchmark name	L2 cache miss rate
INTEGER	gap	0.7
	vortex	0.3
	gcc	0.3
	perl	0.1
	bzip2	0.1
	crafty	0.1
	gzip	0.1
	eon	0.0
FP	apsi	0.9
	wupwise	0.9
	mesa	0.1
	fma3d	0.0

(b) CPU bounded benchmarks

Table 2.3: Cache behavior of each benchmark in a 512Kb L2 cache.

2.3 Complexity-Effectiveness Metrics

Whenever *Complexity-Effectiveness* is involved in a research it generally arises the issue of comparing the relative complexity involved in several architectural proposals. Since the results, obtained with multiple and different microarchitectures involving different hardware budgets, have to be compared some complexity measurement is required to guide such unfair comparisons. The most of the times it is straightforward that larger hardware budgets would yield higher throughput/performance; directly comparing their rough throughput/performance would not lead to a reasonable comparison. So, there is no point on directly comparing the performance obtained using an 8-wide Out-of-Order Superscalar processor with that of a 1-wide In-Order Uniscalar processor. In any case, the throughput/performance obtained for a single workload/application is only comparable as a relative measurement involving both throughput/performance and the complexity involved in its execution.

Quantifying complexity is always a tricky task and giving a single and comparable measurement is even harder to accomplish. A quite generalized approach [59, 64, 65] to estimate the complexity involved in any proposal establishes a direct relation between *complexity* and *area* (measured in mm^2). 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.

During this thesis it was employed the *Karlsruhe Simultaneous Multithreaded Simulator* [59, 64, 65] to estimate the area required by different microarchitectures. This chip

space and transistor count estimation tool receives its input from the baseline architecture and the configuration file of the microarchitecture performance simulator sim-outorder of the *SimpleScalar Tool Set*. The estimation tool yields a pre-silicon chip space and transistor count estimation and allows to compare different microprocessor configurations with respect to their potential chip space requirements. The estimation method, which is the basis of this tool, is validated by configuration parameters of a real processor, yielding a transistor count and a chip space estimation very close to the real processor numbers.

2.4 Cache Configuration

During the development of this thesis we have employed a wide range of different cache configurations. The different cache alternatives employed include monolithic and multibanked, single and multiported, and first and second on-chip hierarchical levels. Since the size of the workloads considered ranges from 1 to 32 running programs, each cache configuration employed strives to assure a minimum cache share. Otherwise, the negative effects of an insufficient cache share may alter the experiment results and, as a consequence, the conclusions obtained. Thus, for each program running on an experiment's workload the cache hierarchy simulated tries to assure at least twice the first level of cache share accessible in the second cache level (e.g., using 4 private L1 caches of 4KBs for a 4-core CMP implementation, with a total thread count of 4, we would employ a minimum L2 cache size of 32KBs). The size of each cache used, split into different access banks, is then set according to the number of running applications.

For each cache configuration employed in an experiment, some additional size-related parameters must be defined, such as access delay. In order to appropriately set these configuration parameters, regarding the access delay to each of these banks, it was employed *CACTI 3.2* [68].

2.5 MPsim

In order to evaluate all the contributions proposed during the reseach time covered by this PhD dissertation it was required a simulation tool which provided high flexibility. The selected simulator should allow to simulate both single-core and multi-core processor implementations, including homogeneous and heterogeneous clustered multithreaded implementations. It must also offer a wide range of research (i.e., allow multiple simulation alternatives to cover a wider design space) with a simple interface. Due to these special requirements it was developed the *Multi-Purpose (MPsim)* simulation tool,

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	15 cyc.
Main Memory Latency	250 cyc.
I-TLB/D-TLB/TLB missp.	48 ent. / 128 ent. / 300 cyc.

Table 2.4: Simulation parameters (resources marked with * are replicated per thread)

a highly-flexible tool that allow researchers to cover wide ranges of the design space. Using the MPsim Simulation Tool it may be easily simulated the execution of multi-threaded multicore scenarios involving very different processor layouts, from clustered Superscalars/SMT processors to full-fledged multithreaded multicore processors or even many-cores. Full details of the *MPsim* are given in the Appendix.

Unless explicitly indicated otherwise, all execution cores and memory subsystems used in the microarchitectures evaluated throughout this research have a similar configuration, shown in Table 2.4. In some cases, this configuration is used as a baseline reference when reducing the amount of resources per execution core and in other cases it is simply altered. Use the simulation parameter information shown in Table 2.4 as a common reference throughout the remainder of this document.

Regarding simulation itself, in a wide range of *SMT* experiments it is required to compare the results (in terms of committed instructions) using different Instruction Fetch (IFetch) policies. Since IFetch policies alter the amount of instructions committed per each thread we opted to take *IPC* measurements during fixed amount of simulation cycles: the very same amount of cycles starting from the very same execution point. Consequently, with similar constraints the one with higher results (i.e., higher IPC) would be the best since is able to commit more instructions under similar conditions. The main reason to settle for this approximation, instead of using a more reliable measurement system like FAME [32], is the simulation time. Considering that a wide design space exploration, like the one done during this research, is likely to involve hundreds of thousands of simulations, the simulation time per experiment should be a prime concern. Consequently, such an approximation had to be taken in order to keep a reasonable computational cost.

Chapter 3

Heterogeneous Simultaneous Multithreading Processors

Today's application behavior is inherently heterogeneous. This heterogeneity spreads out applications at two levels: *inter-application* and *intra-application* level. In this chapter it is analyzed the application heterogeneity and how this heterogeneity affects the design of the main structures of current processors. Thus, while increasing the size of the instruction queues may yield considerable benefits for some applications, like *181.mcf* and *175.vpr*, others may experience no significative improvements, like *252.eon* and *186.crafty*.

From the study of the application heterogeneity in current typical workloads it may be asserted that forthcoming processor generations should take into account this heterogeneity; that is, being *"Heterogeneity-Aware"*. In this chapter we deeply analyze the main proposals in the *Heterogeneity-Aware Architectures* field. They all seek to yield *complexity-effective* executions, giving each application exactly the hardware it requires for an optimal execution. By clustering some of the main processor structures some of these proposals go along the sometimes fuzzy frontier that differentiates the *SMT* and *CMP* paradigms.

The present study of the heterogeneity in the *SMT* processors ends up with the first thesis contribution: the *heterogeneously distributed Simultaneous Multithreading (hdSMT)* architecture. The *hdSMT* architecture is based on a novel combination of *SMT* and *clustering* techniques in an heterogeneity-aware fashion. The results included in the *hdSMT* evaluation enclosed, including both performance and performance per area evaluations, show the *hdSMT* benefits when optimizing performance per area over both monolithic and homogeneously clustered *SMT* processors.

3.1 Application Heterogeneity

The heterogeneity in the application's behavior is not a new issue in the Computer Architecture field. From the very first *Superscalar* processor to the modern *Simultaneous Multithreading (SMT)* and *Chip Multiprocessors (CMP)*, it has been realized that processor resources are not equally used by the running applications. In fact, this is one of the fundamentals that led to *Multithreading (MT)*. Since not all *Superscalar* processor resources are used by a single application, they may be shared with additional active threads in the same processor.

In order to make a better use of the available resources, multithreaded architectures need to perform the resource distribution among co-scheduled tasks. This scheduling step is known as *Resource Sharing*. In a *CMP* processor this step is implicitly performed, since the processor resources are statically splitted into replicated single-thread cores; only L2 caches are typically shared among all running applications. However, the heterogeneity in the behavior of different applications, that is the *inter-application heterogeneity*, may turn an static hardware partition into non-effective for some workloads. While some applications' execution may be hampered by such a partition, others may waste hardware resources within a single-thread core. *SMT* processors solve this problem by dynamically sharing all processor resources among all active threads. Thus, *Thread Level Parallelism (TLP)* is exploited without renouncing to single thread *Instruction Level Parallelism (ILP)*. However, an inappropriate *Resource Sharing*, generally performed by the *Instruction Fetch (IFetch) Policy*, may hardly affect the system throughput in an *SMT* processor. Resource conflicts may occur when several applications, with conflictive behaviors, are executed together in the same *SMT* processor. In this sense, the literature is plenty of techniques [19, 20, 24, 25, 70, 71, 72] that try to reduce these kind of conflicts.

Inter-application heterogeneity represents only one half of the heterogeneity present in the behavior of current applications. In fact, applications do not behave the same during the whole execution [56]; that is they experience *intra-application heterogeneity*. Due to this fact, while a great amount of processor resources are wasted during some *execution phases* they are pushed to their limits during other *execution phases* of the same application. The straight conclusion is that the most appropriate amount of resources for a single application execution can not be expressed as a single number —it varies along its execution. According to this conclusion, it may be inferred that forthcoming processor generation designs should be conscious of this application heterogeneity, that is they should be *Heterogeneity-Aware*. Both *inter-* and *intra-application heterogeneity* should be explicitly taken into account in *Heterogeneity-Aware* designs to better profit the available hardware resources.

In this chapter both kind of application heterogeneity are evaluated —*intra-* and *inter-application heterogeneity*. We focus on the *SMT* approach, since it represents the most prone to suffer the negative effects of *application heterogeneity* —multiple running threads sharing all the processor resources. From this evaluation it is justified the need of *Heterogeneity-Aware architectures*. Then, it is proposed the first contribution of this thesis: the *heterogeneously distributed SMT (hdSMT)* Architecture. In this novel *Multithreaded (MT) Architecture* the hardware is heterogeneosly distributed along the chip's surface. The heterogeneity in software is then matched with the appropriate cluster of resources in order to maximize the execution's *complexity-effectiveness*.

3.1.1 Heterogeneity Considerations in the Processor Design

General-purpose microprocessors are built up from an on-chip transistor budget with the goal of maximum performance for all applications. As the process technology advances, the amount of available transistors on a single chip increases. The advances in the process technology has kept an steady improvement rate for the last decades. The *Moore's Law* describes this important trend in the history of computer hardware: *"the number of transistors that can be inexpensively placed on an integrated circuit is increasing exponentially, doubling approximately every two years*[1]*"*. The observation was first made by Intel co-founder Gordon E. Moore in a 1965 paper [45, 46]. The trend has continued for more than half a century (See Figure 3.1) and is not expected to stop for at least another decade, and perhaps much longer [35].

As the number of transistors on a single chip increases, the issue of how to effectively employ them to improve the applications' performance gains importance. In the last decades we have witnessed many architectural approaches to exploit the ever-growing amount of transistors on a single chip. From the early *Scalars* to the modern *Multithreaded Multicores*, the processor design has always striven to yield the highest performance possible with the available hardware budget. Nowadays, the power and temperature constraints in the state-of-the-art processors are somehow turning the performance primary goal into a *complexity-effectiveness* search. A single processor design should obtain the highest performance reachable for a fixed hardware budget, for a wide range of applications, involving the least power consumption possible. The processor design has also to balance the heat dissipation throughout the chip's surface so that harmful *hotspots* are prevented. Whenever an small portion of the chip experiences an exhaustive utilization, the heat generated supposes a challenge for the chip heat dissipation system. This

[1] Although originally calculated as a doubling every year, Moore later refined the period to two years. It is often incorrectly quoted as a doubling of transistors every 18 months.

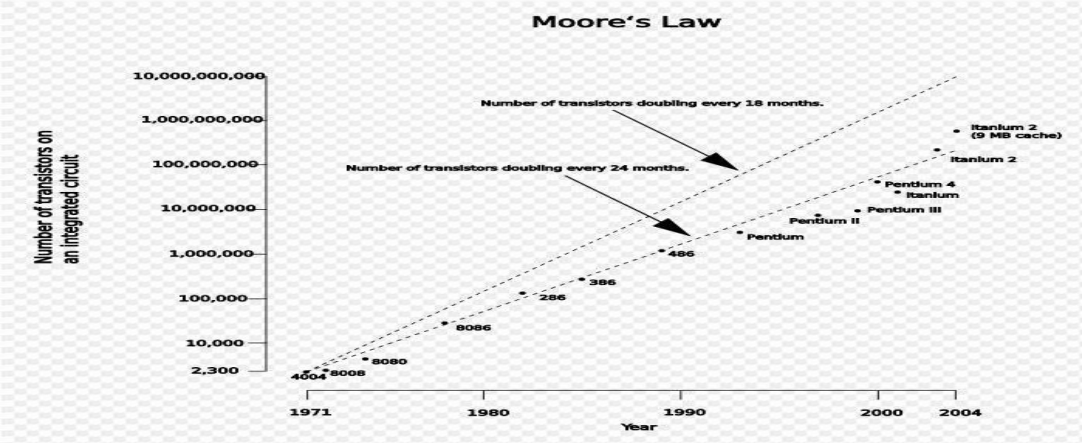


Figure 3.1: Moore's Law.

situation tends to occur with some important processor structures which also have the disadvantage of possessing low design regularity, like the *Instruction Queues* and *Reorder Buffers* used in *Out of Order* execution pipelines.

The *Scalar* processors executed one instruction at a time. In these processors, each executed instruction typically manipulates one or two data items at a time. In contrast, each instruction executed by a *Vector* processor operates simultaneously on many data items. The *Superscalar* processors arised as a sort of mixture of the two. While each instruction processes one data item, the addition of multiple redundant functional units within each CPU allowed the *Superscalar* processors to execute multiple instructions at a time; thus multiple instructions can process separate data items concurrently. Seymour Cray's *CDC 6600* (1965) is often mentioned as the first *Superscalar* design. It is not until the late 80's that appeared the first commercial single chip superscalar microprocessors: The *Intel i960CA* (1988) and the *AMD 29000-series 29050* (1990) microprocessors.

The *Superscalar* CPU design emphasizes the instruction dispatcher accuracy, allowing it to keep the multiple functional units in use at all times. This has become increasingly important when the number of units increased. While early *Superscalar* CPUs had two ALUs and a single FPU, a modern design like the *PowerPC 970* (2002) includes four ALUs and two FPUs and a couple of SIMD units too. If the dispatcher is ineffective at keeping all of these units fed with instructions, the performance of the system will suffer altogether. The introduction of better conditional *Branch Predictors*, like the *gshare* [44], *bimodal* [81], *2bcgskew* [54], *stream* [52], and the *perceptron* [34] predictor, consider-

ably improved the achievable performance. Reducing the amount of instructions executed along the wrong path allows more aggressive execution pipelines, simultaneously dispatching more instructions per cycle. Better *branch predictions*, together with the *Out of Order execution* [31, 62], were crucial in the search of wider execution pipelines.

In a *Superscalar* CPU the dispatcher reads instructions from memory and decides which ones can be run in parallel, dispatching them to redundant functional units contained inside a single CPU. Therefore a *Superscalar* processor may be envisioned as having multiple parallel execution pipelines, each of which is processing instructions simultaneously from a single instruction thread. This seemed for a time the best choice to invest the hardware budget on within each chip. Employing the additional transistors to enlarge the main processor structures allows to increase the number of parallel execution pipelines within the processor. However, architects soon realized that the performance achievable by this execution scheme does not scale with the available transistors due to the limitations imposed by the *Instruction Level Parallelism (ILP)*. Regardless of the additional transistors employed to design a more aggressive execution pipeline, the application characteristics finally impose a hard limit to the achievable performance. Furthermore, this limit is different depending on the specific characteristics of each given application; that is depends on *inter-application heterogeneity*. Thus, while devoting the additional transistors to enlarge some processor structures, like the *Instruction Queues*, could yield benefits for some applications, for others we could experience diminishing returns.

As a consequence of the hard limitations imposed by the *ILP*, the *Thread Level Parallelism (TLP)* has become a common strategy for improving processor performance. Since it is difficult to extract more *ILP* from a single program, multithreaded processors rely on using the additional transistors to obtain more parallelism by simultaneously executing several programs. This strategy has led to a wide range of multithreaded processor architectures like *SMT* [71, 72, 80], *CMP* [48], or combinations of both. They all extend the *Superscalar* design by simultaneously sharing the processor resources among multiple running applications. Whenever the *ILP* of a single application prevents from having busy all available resources in the processor, the idle resources are devoted to other applications, which are simultaneously run on the same processor. The main difference between the *SMT* and *CMP* approaches resides in the amount of on-chip processor resources shared among all running applications. Thus, while a typical *CMP* implementation only shares the outter on-chip cache layer (typically the L2 Cache), an *SMT* processor shares all processor resources. Due to the inherent heterogeneity in the application's behavior, the resource utilization pattern of each running application may collide during the execution ending up with resource contention. Since they share more resources among the running threads than *CMP* processors, these resource conflicts affect more severely to *SMT* processors. Thus, the literature is plenty of techniques [19, 20, 24, 25, 70, 71, 72] that try to

	M8	M4	M2
Queues Entries (int / fp / ld-st)	64	32	16
Renaming Registers	256	128	64
Number of Contexts	4	2	1
Processor Width	8	4	2
Max. Number of Instructions/cycle (per thread)	8	4	2
Max. Number of Threads/cycle	2	2	1
Execution Units (i = int, f = fp, l = ld/st)	6 i, 3f, 4l	3 i, 2f, 2l	1 i, 1f, 1l
Fetch Policy (IC = ICOUNT)	IC 2.8	IC 2.4	IC 1.2

(a) Processor model configuration.

Processor Configuration	
ROB Size / thread	256 entries
Branch Predictor Configuration	
Branch Predictor	2K entries gshare
Branch Target Buffer	256 entries, 4-way associative
RAS	256 entries
Memory Configuration	
Icache, Dcache	64KB, 2-way, 8 banks, 64-byte lines, 1 cycle access
L2 cache	512KB, 2-way, 8 banks, 12 cycles lat, 64-byte lines
Main Memory latency	100 cycles
DTLB size/ ITLB size/ TLB miss penalty	128 ent. / 48 ent. /160 cycles

(b) Baseline configuration.

Table 3.1: *Application Heterogeneity Simulation Configuration.*

reduce these kind of conflicts in *SMT* processors. In all cases, the goal it to allocate to each application the appropriate amount of hardware resources, avoiding monopolization by any individual application. Whether each application needs are appropriately matched with the allocated processor resources the system throughput may experience significative improvements. This proper match requires from a deep analysis of the application needs and the differences among them; that is an analysis of the *application heterogeneity*.

State-of-the-art microprocessors suggest a trend towards building multithreaded multicore processors with an increasing amount of multithreaded cores on-chip. As a consequence, forthcoming processor generations will face harder challenges related to on-chip resource contention. In order to appropriately handle this contention, the *application heterogeneity* should be deeply analyzed in typical execution workloads. In the following sections it is analyzed the heterogeneity in the *SPECINT2000 Benchmark Suite*, both at *inter-* and *intra-application level*. From this analysis some further processor design considerations are asserted.

3.1.2 Methodology

Table 3.1.(a) shows the three processor models simulated. From now on, the three processor models shown in Table 3.1.(a) will be referred to as *M8*, *M4*, and *M2*. These processor models are used to compare application needs and so showing *inter-application heterogeneity*. The name of each of these models give a quick idea of the resource budget comparison. Thus, in general terms, an *M8* processor has twice the hardware budget devoted for the main processor structures than an *M4*; the same happens between the *M4* and *M2* processor models.

Table 3.1.(b) shows the main parameters of the simulated processors, which have 8-stage execution pipeline. Three different processor models, with varying number of some specific resources (e.g. instruction queues, renaming registers, issue width, etc), are used to evaluate the heterogeneity in applications. Please, refer to Chapter 2 for full details on the experimental framework.

3.1.3 Inter-Application Heterogeneity

The resource utilization pattern significantly differs from one application to another. While some applications make an intensive use of some resources, like rename registers and instruction queues, others obtain good performance results with a more moderated hardware budget. As a matter of example, Figure 3.2 shows the rename registers needed by each of the SPECINT2000 benchmarks to reach a *90% of their peak performance*[2], executed in an *M8* processor (see Table 3.1) in single-thread mode. Although the rename registers are not the only critical resource in an out-of-order execution processor, they suppose a serious bottelneck whenever parallelism is to be exploited. Devoting additional resources to other critical processor resources, such as better conditional branch predictors or memory, would end up requiring to increase the amount of rename registers to increase the processor's peak performance.

Figure 3.2 shows that *176.gcc* requires only 32 rename registers. However, other applications, like *175.vpr* or *255.vortex*, require 128 rename registers. We also find a group of *moderated* applications, like *181.mcf* or *256.bzip2*, requiring 64 rename registers to obtain the *90-pp*. Let be *G1*,*G2*, and *G3* the groups of applications which require 32, 64, and 128 rename registers to reach the *90-pp*, respectively. From the results shown in Figure 3.2 it may be inferred that simultanously executing, on the same execution pipeline,

[2]Maximum performance obtained using an unlimited resource budget. In this case, the application's characteristics limit the maximum performance level reachable.

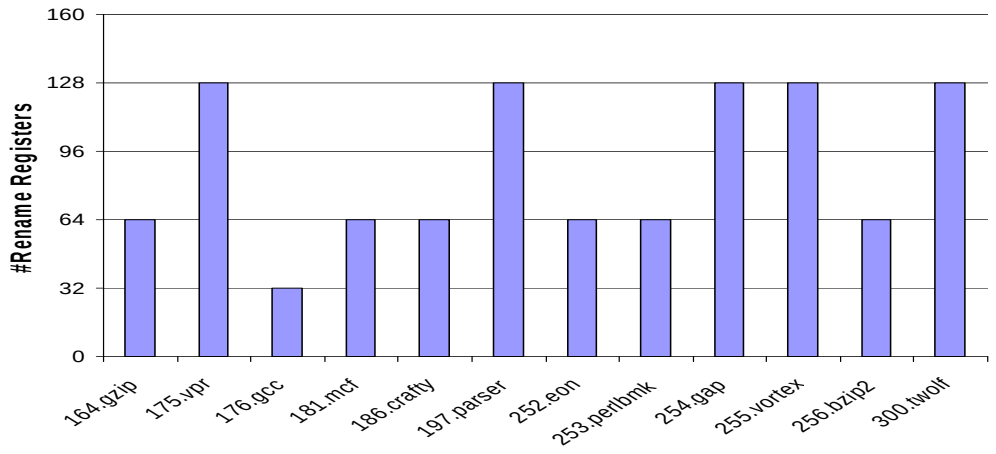


Figure 3.2: *Rename Registers needed to reach 90% Performance.*

applications from the group *G3* may yield diminishing returns. Due to their eager usage of the rename registers, multiple applications of the group *G3* may experience high resource contention, with the subsequent reduction in the system througput. Therefore, the best candidates to be simultaneously executed with a *G3* application belong to groups *G1* and *G2*. That is, those that exhibit different rename register utilization patterns.

Statically splitting the processor resources among all the active threads, as done in *CMP* processors, gets rid of the resource contention and may be beneficial in some scenarios; although it supposses a *hard commitment*. Whenever the executed workload possesses high *TLP*, and moderate per-application *ILP*, statically partitioning the processor resources may be productive. However, the differences in the resource needs from one application to another, that is the *inter-application heterogeneity*, may turn this static partition into a serious drawback. While some *high-ILP* applications may require more resources, than the ones allocated to a single execution core, other *low-ILP* applications may be wasting resources within an execution core.

As an illustrative example, Figure 3.3 shows the benefits of sharing the L1 caches among all constituent cores in a *CMP* processor comprised of four *M2* cores. That is, any of the four constituent exection cores in the CMP processor may access up to an overall L1 cache budget of 64KB. Assuming the case of single-thread execution[3], the results shown in Figure 3.3 indicate the benefit obtained whether each application may access to each of

[3]This would represent the worst scenario in a *CMP*; there is only a single ready task ready to be executed by the Operating System.

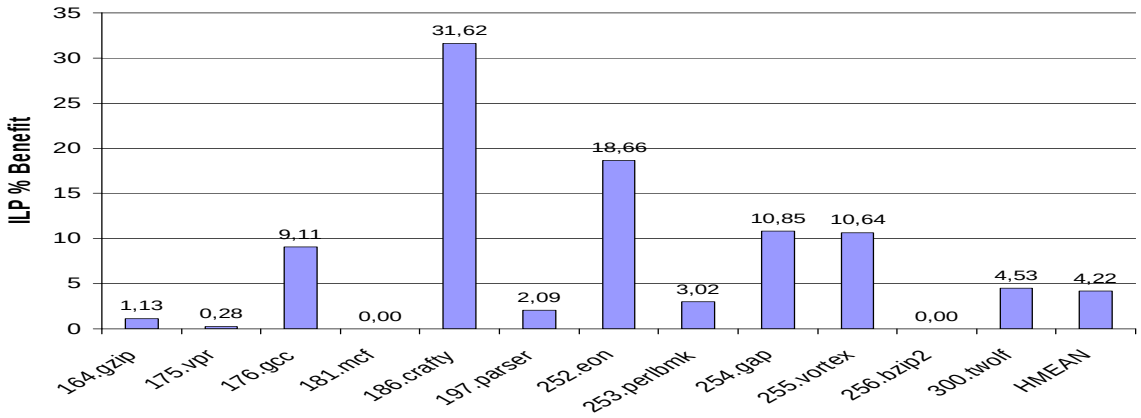


Figure 3.3: *Benefits of Sharing L1 caches in a four-cored CMP.*

the four 16Kb L1 caches[4], with a total L1 cache budget of 64Kb, compared to exclusively accessing its private 16Kb L1 cache. The results obtained significantly vary depending on the *memory footprint* size of each application (i.e., the extent of memory that it uses or references while executing), as depicted in Figure 3.3. Whenever the *memory footprint* of an application fits into a single 16Kb L1 cache, as is the case of *256.bzip2*, no benefit is obtained from sharing the remainder cache budget. However, as the *memory footprint* of each application grows the potential benefit obtained from sharing partitioned resources increases; as is the case of *186.crafty* with an improvement close to 32%. Obviously, *CMP* processors were not designed for single-thread mode; the most of the time the Operating System can provide enough ready tasks to keep busy more than execution core. However, this example illustrates the potential drawback from statically partitioning the processor resources. Whenever the Operating System in a *CMP* processor, or whatever design with an static resource partition, is unable to select for execution enough ready tasks, the system performance is hardly affected. The only running application may experience a performance degradation, due to limited access to processor resources, while other resource partitions are wasted. In this case, due to a low *TLP* in the workload, the *ILP* and specific characteristics of each application, as the *memory footprint*, may be decisive for the system performance. If this is the case, an statically partitioned hardware may suppose a serious drawback.

As Raasch and Reinhardt show in [51], there are some cases in which statically par-

[4]To ease the example, it is assumed no additional overhead when accessing to the private L1 cache from a different core.

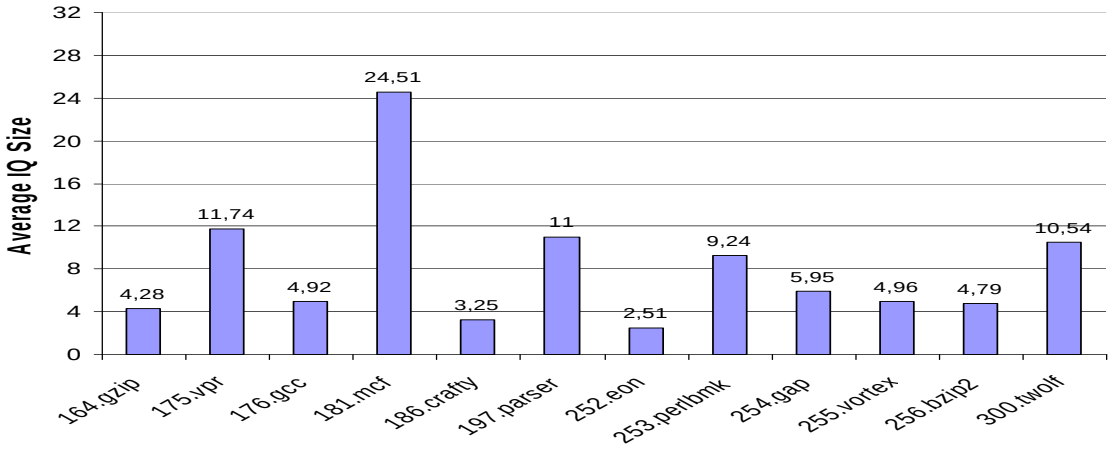


Figure 3.4: *Average IQ Size.*

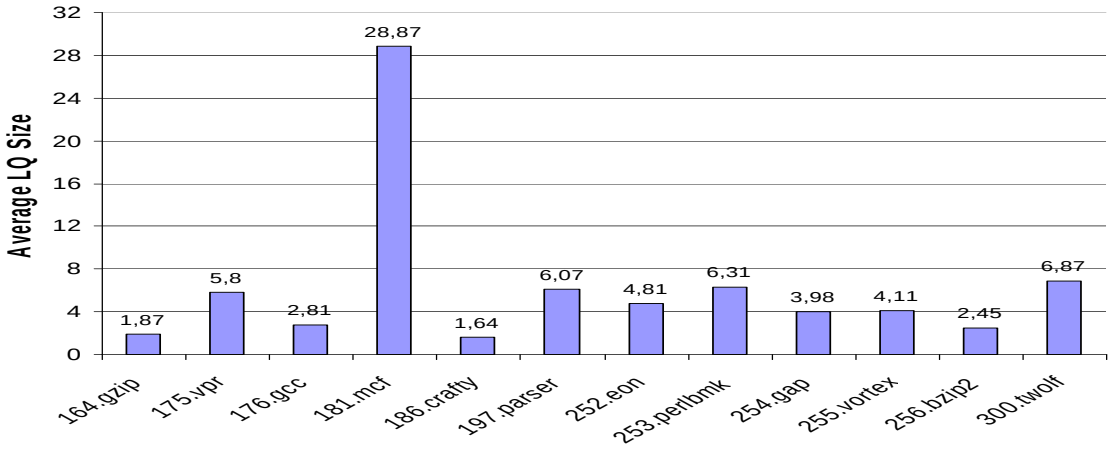


Figure 3.5: *Average LQ Size.*

titioning some of the processor structures, like the instruction queues, may be beneficial to achieve better results. Due to the applications' heterogeneous usage of some processor structures, giving a fixed resource share to each of the running applications may yield significative performance improvements. To further illustrate this heterogeneous usage, Figure 3.4 shows the average size —occupancy— of the *integer instruction queue (IQ)* of each *SPEC2000 INT benchmark*, executed in single-thread mode on an *M4* processor

with private 64KB L1 caches. Figure 3.5 shows the equivalent results for the *load/store instruction queue (LQ)*. While the *IQ* usage exhibits a moderate variation from one application to another, the *LQ* usage is quite balanced among all applications. The instruction queue usage exhibited by the *181.mcf* represents a pathological case, in which a bad memory behavior may clog the instruction queue if unproperly handled. An static *IQ/LQ* partition gets rid of these clogs, avoiding applications from monopolizing processor structures. As a consequence, statically allocating an equal portion of the instruction queues to each simultaneously executed application provides good performance, in part by avoiding starvation. However, although not highly pronounced in the case of instruction queues, the different application usage of some processor resources, like the rename registers (See Figure 3.2), may turn an homogeneous resource partition into a bad choice.

The heterogeneous applications' behavior (i.e., *inter-application heterogeneity*) shown in all prior resource usage examples directly affects the overall system performance. Figure 3.6 shows the execution results of each *SPECINT2000 benchmark* executed in single-thread mode in each of the three processor models simulated. For each application and processor model, it is shown the performance obtained measured in *Instructions Per Cycle (IPC)*. These *IPC* values are compared with the processor's *peak performance* (the horizontal bars in Figure 3.6), obtained using all the available hardware resources in each processor model in an ideal case.

A glance at Figure 3.6 is enough to realize that not all applications exploit the available processor resources with the same effectiveness. Thus, while *256.bzip2* use the available hardware resources in an *M2* processor with an *80,5%* of *effectiveness*[5], *181.mcf* exhibits an *effectiveness* below *10%*. Therefore, devoting all *M2* processor resources to execute *181.mcf* yields a very poor *complexity-effective* execution. This situation gets worse as it is augmented the processor hardware budget. As shown in Figure 3.6, all applications experience *diminishing returns*, in terms of *complexity-effectiveness*, when moving to a bigger processor model. So, even a *high-ILP* application like *256.bzip2* experiences a reduction in its *effectiveness* of factor of $\times$2,5 when moving from an *M2* to an *M8* processor. However, the *256.bzip2*'s performance also experiences a growth of factor of 2,5$\times$ when moving from *M2* to *M8*; justifying the additional hardware budget in terms of *complexity-effectiveness*.

The straight conclusion is that the *Inter-Application Heterogeneity* may turn a given hardware budget into a *non complexity-effective* choice. On the one hand, the specific characteristics of each application determine the *application's peak performance*. In an ideal case, with plenty of execution resources, it is the *application's peak performance* the key factor that imposes the top performance limit. On the other hand, the combination of

[5]That is, the performance obtained is 19,5% lower than the processor's *peak performance*.

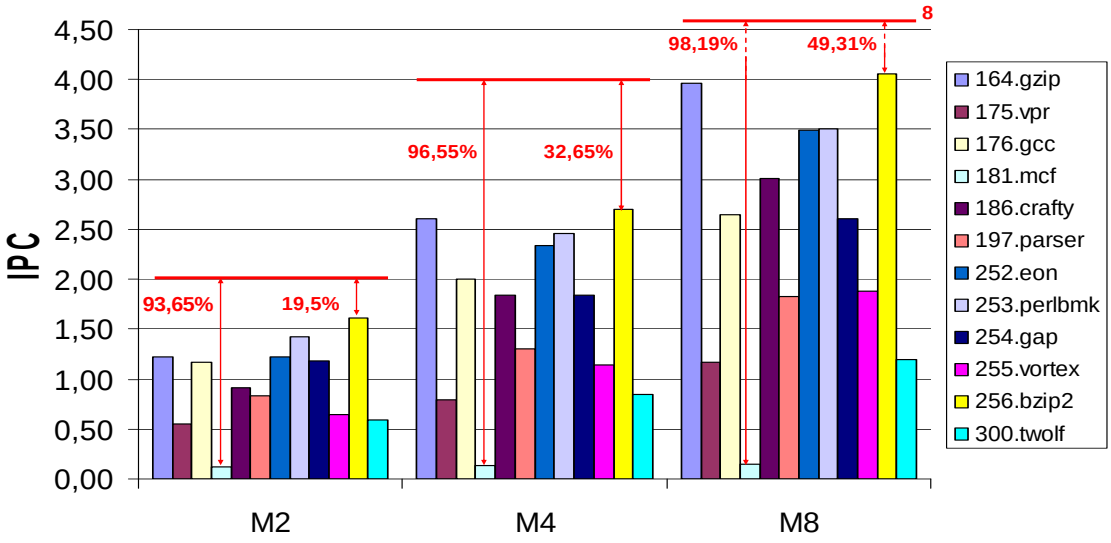


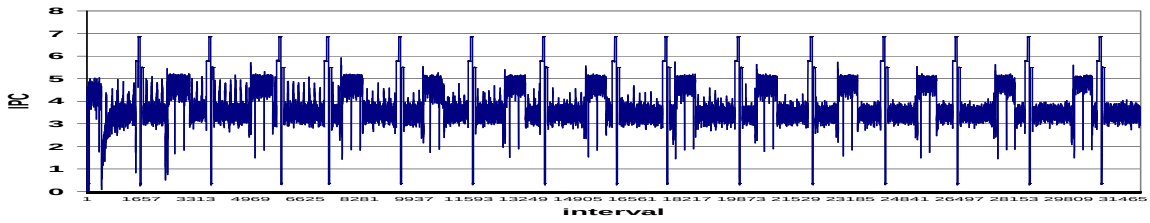
Figure 3.6: Heterogeneity at Inter-Application level.

all the hardware resources devoted to a processor define its *processor's peak performance*. The more similar both peak performance levels, application's and processor's, the more complexity-effective executions achieved. Thus, while it is worthwhile executing the *256.bzip2* on an *M4* processor, or even an *M8*, the *181.mcf* does not need more than an *M2* processor. An *Heterogeneity-Aware* architecture should explicitly take this fact into account, allocating the appropriate amount of resources to each application according to its specific needs.

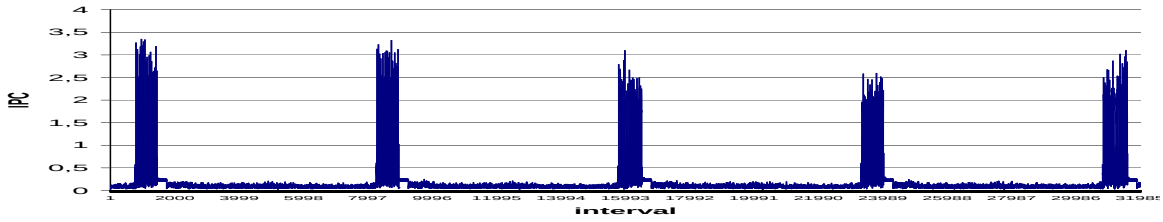
3.1.4 Intra-Application Heterogeneity

It is not necessary to compare the behavior of different applications to find heterogeneity. In fact, comparing two different *Program Phases*[6] from a single application is enough to find an heterogeneous behavior; that is *Intra-Application Heterogeneity*. To illustrate this phenomenon, Figure 3.7 shows performance histograms of some *SPECINT2000* applications, simulated on an *M8* processor in single-thread mode during an specific interval of their execution.

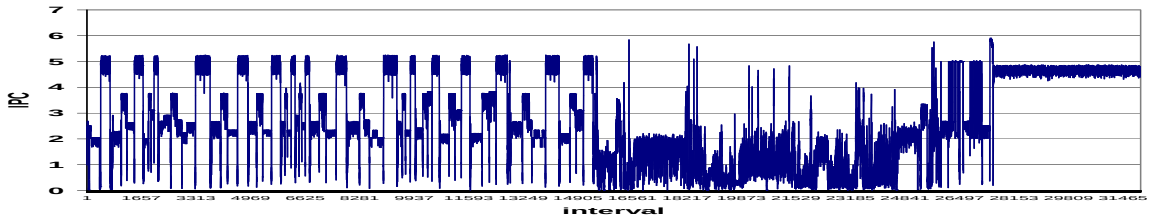
[6]That is, a differentiated portion of the execution of an application, with a particular behavior.



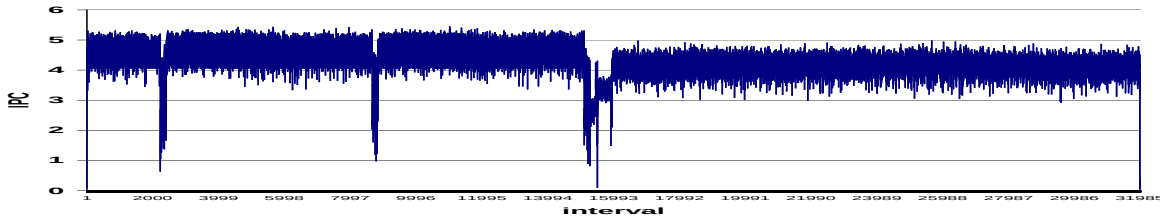
(a) 164.gzip.



(b) 181.mcf.



(c) 253.perlbmk.



(d) 256.bzip2.

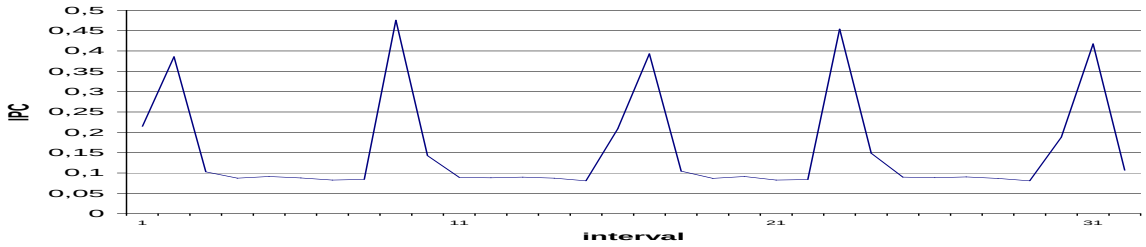
Figure 3.7: Heterogeneity at Intra-Application level.

Each histogram in Figure 3.7 shows the performance, measured in *IPC*, reached by the first 32000 execution intervals of each application[7]. Each of these intervals comprises 1000 consecutive cycles of the application's execution. Although these histograms are not representative of the whole application execution, it is clearly noticeable that they exhibit heterogeneous behaviors. In Figure 3.7 there are examples of highly-periodic behavior, like *164.gzip* and *181.mcf*, and steadier ones, like *256.bzip2*. There are also examples of multiple *Program Phase* transitions, as is the case of *253.perlbmk* in which 3 different *Program Phases* are clearly noticeable. During this 32-million-cycle period of execution the *253.perlbmk* goes through an initial highly-variable high-ILP phase (first 15.000 intervals), a second highly-variable low-ILP phase (following 13.000 intervals), and finally an steady high-ILP phase. Comparing the variability of the results in each of these three phases, Figure 3.7.(c) also shows high heterogeneity in this sense. Thus, while the first two *Program Phases* of *253.perlbmk* exhibit a highly-variable behavior, with quick performance variations of up to 50X, the third *Program Phase* exhibits a quite steady behavior, with performance variations lower than 15%.

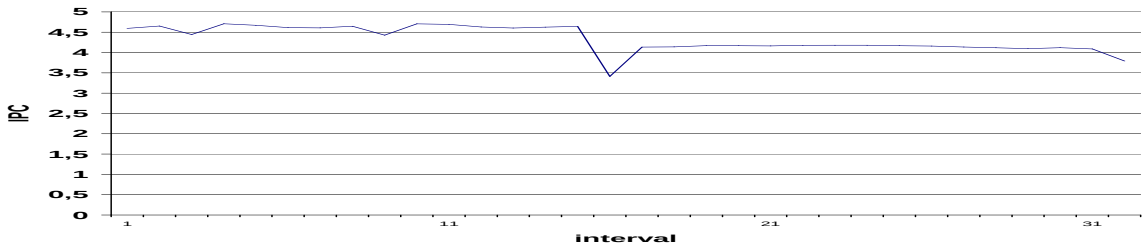
Focusing on two applications with a very different overall performance, *181.mcf* and *256.bzip2*, their execution histograms exhibit some interesting characteristics. While both applications experience similar performance variations considering average results, about 99% and 60% for *181.mcf* and *256.bzip2* respectively, they represent opposite behaviors. While the steadier and common behavior in the case of *181.mcf* is very low-ILP, in the case of *256.bzip2* is high-ILP. Nevertheless, both applications experiment isolated performance fluctuations of up to 15X and 25X, in case of *181.mcf* and *256.bzip2* respectively.

From the results shown in Figure 3.7 it may be inferred that the *Intra-Application Heterogeneity* may alter, during some periods of the execution, the complexity-effectiveness of the decisions took according to the *Inter-Application Heterogeneity*. Thus, while the most complexity-effective processor assignments for each application (*M2* for *181.mcf* and *M8* for *256.bzip2*) are still valid, there are some periods —about 1 million cycles each time— in which these assignments do not represent the best choice. Notice in Figure 3.7 that *181.mcf* experiences periodic high-ILP intervals, with an average *IPC* higher than 2. As a consequence, the *181.mcf* execution during these high-ILP intervals would be hampered if executed on an *M2* processor, with a peak performance of 2. Just the opposite happens in the 3 low-ILP intervals that exhibit *256.bzip2* in Figure 3.7.(d). Devoting a full *M8* processor during these intervals, with an average *IPC* lower than 3, involves some resource wasting. In these cases, such a resource waste (or resource lack in case of *181.mcf*) may no significatively alter the overall system performance, due to the re-

[7]For each application, the simulated execution intervals comes immediately after applying the corresponding forwarding, as shown in Chapter 2.



(a) 181.mcf.



(b) 256.bzip2.

Figure 3.8: Heterogeneity at Intra-Application level at coarser granularity (1M cycles).

duced extension of these intervals. However, in case of longer steady variations in the application's dynamic behavior, the system performance may be severely affected by this *Intra-Application Heterogeneity*.

Figure 3.7 also suggests another important conclusion regarding the duration of the intervals considered. In fact, the *granularity* at which *Intra-Application Heterogeneity* is exploited directly determines both its applicability and the results obtained. Thus, while this heterogeneity may be detected and exploited using a 1000-cycle *granularity* (as used in Figure 3.7) the resulting performance variations are not steady enough to be exploitable using a coarser granularity of 1 million of cycles, as shown in Figure 3.8. Although there are still noticeable variations in the application's behavior shown in Figure 3.8 they do not alter the complexity-effectiveness of the decisions taken from an *Inter-Application Heterogeneity* perspective; that is executing *181.mcf* and *256.mcf* on *M2* and *M8* processors, respectively. In both cases, there is not an steady interval of execution that reach an *IPC* value above the peak performance of the respective processor model.

The applicable *granularity* in each case would depend on the system specifications. Once detected the heterogeneity in the application's behavior its execution must be re-scheduled. The resource needs of each detected *Program Phase* should be matched with the available hardware partitions in an heterogeneously partitioned hardware. This matching process may involve costly migrations between different hardware partitions. For an heterogeneous behavior to be profitable, the overhead involved by the re-scheduling or migration of the applications must be low enough as compared to the length of each detected *Program Phase*.

The straight conclusion is that the *Intra-Application Heterogeneity* may alter the *complexity-effectiveness* of the decisions taken from an *Inter-Application Heterogeneity* perspective. That is, a single resource assignment, that best fits the resource needs of each application, may not be valid for the whole execution. However, the exploitability of this *Intra-Application Heterogeneity* is subject to both the *detection granularity* and *re-scheduling overhead*. First, the different *Program Phases* must be appropriately detected. Next, the cost involved in the corresponding resource reassignment must be low enough as compared to the *Program Phase* length. Thus, in order to appropriately exploit the periodic 1-million-cycle high-ILP phases detected in the *181.mcf* behavior using a 1000-cycle granularity (Figure 3.7.(b)), the cost involved by the corresponding resource reassignment (e.g., migrating from an *M2* to an *M4* core) must involve an insignificant overhead as compared to 1 million of cycles.

3.2 Heterogeneity-Aware Architectures

The *Heterogeneity-Awareness* of a processor design could be defined as the way in which it explicitly manages the *Application Heterogeneity* to achieve a *complexity-effective* execution. The degree of success in assigning to each application the processor resources it needs determines the degree of *Heterogeneity-Awareness* of each processor design. Giving each application strictly the required processor resources helps to reduce the execution power consumption without reducing the performance; that is, to improve the execution's *complexity-effectiveness*. From the *Application Heterogeneity* analysis performed in prior sections it may be inferred that an *Heterogeneity-Aware* design should take into account both *Inter-* and *Intra-Application Heterogeneity*.

The way in which the hardware is heterogeneosly distributed along the chip surface also contributes to the *Heterogeneity-Awareness* of the processor design. Partitioning hardware structures and resource pools, like the instruction queues and register files, into heterogeneous clusters, with different number, sizes, and types of these structures, helps to achieve more *complexity-effective* processor designs. Recall that the main goal of almost

all general-purpose processor consists of achieving the highest performance with the least energy consumption possible. Distributing the hardware into smaller and heterogeneous clusters, which have a lower energy consumption, and assigning the applications to the cluster that best fits the application's resource needs contributes to improve the design's *complexity-effectiveness*.

Inside the *Heterogeneity-Aware* category there could be included a wide range of processors, both single-threaded and multithreaded. In fact, an *SMT* processor implementing an *Instruction Fetch (IFetch) Policy*, that dynamically distributes the shared resources among the running applications, could be seen as some kind of *Heterogeneity-Aware* design. However, the difference between the resource sharing, performed by a typical *IFetch Policy*, and the *Heterogeneity-Awareness* lies in explicitly considering the application heterogeneity to give each application the hardware resources it needs. Typical *IFetch Policies*, like *ICOUNT* [72], try to balance the resource usage among all the running applications. Instead of giving each application the resources they need, they tend to give each application a similar portion of the available resources.

More advanced *IFetch Policies*, like *DCRA* [20], explicitly classify the running applications into *slow* and *fast*. Additionally, according to the usage of each resource type by each application, a further classification into *active* and *inactive* helps to assign each application an amount of processor resources in tune with the application's needs. Although its hardware is not partitioned, an *SMT* processor implementing *DCRA* policy may be included into the *Heterogeneity-Aware* category.

There are some examples in the literature that combine an explicit management of the *Application Heterogeneity* with an heterogeneous hardware distribution. In spite of they all strive to achieve a *complexity-effective* execution, the main difference among them lies in the *granularity* at which application heterogeneity is exploited.

The *Dual Speed Pipelines* [50] architecture can be defined as a *Superscalar* comprised of an heterogeneous set of components. As shown in Figure 3.9, different types of processor resources, such as functional units and reservation stations, are gathered into two different execution pipelines: *slow* and *fast*. The slow components of the processor can be driven at lower supply voltages and thus present an opportunity to save power; contributing to improve its *complexity-effectiveness*. However, slow components also imply lower IPCs. In order to avoid harmful performance degradations, a variant of the critical path analysis technique [75] is moved from the circuit level to the architecture level. In this architecture, a criticality predictor is used, in a cycle-by-cycle rate, to correctly identify the critical instructions. Once identified, critical instructions are dynamically scheduled on high-performance, high-power consumption components. Thus, the processor performance is retained while achieving power savings by dynamically scheduling all other

instructions to the low-power (and lower performance) execution units.

The *Dual Speed Pipelines* architecture represents an extremist example of *Heterogeneity-Aware* architecture, in which the *Intra-Application Heterogeneity* is detected at an instruction *granularity*, and measured in terms of instruction criticality. According to this heterogeneity, each instruction uses the resources that best fits with its needs, consuming the least power possible without severely compromising the system performance.

The *Heterogeneous Multicore* [38] architecture can be defined as a *CMP* processor comprised of an heterogeneous set of execution cores. In the example shown in Figure 3.10, the processor is comprised of four Alpha cores EV4 (Alpha 21064), EV5 (Alpha 21164), EV6 (Alpha 21264) and a single-threaded version of the EV8 (Alpha 21464), referred to as EV8-. Although all the cores execute the same *Instruction Set Architecture (ISA)*, there exist differences between cores regarding their raw execution bandwidth (issue width), cache sizes, and many other fundamental characteristics (e.g., in-order vs. out-of-order execution, single-thread vs. multithread execution). In this architecture, the *Operating System (OS)* is in charge of migrating the application's execution based on performance metrics. To obtain these metrics, the execution has to go through periodic *sampling phases*. During each of these *sampling phases* the applications are executed in each of the heterogeneous cores, in order to determine the one that best fits the current application behavior. According to the workload size, these *sampling phases* may suppose a significant execution overhead. In order to mitigate this overhead, that may involve millions of execution cycles, some heuristics are applied.

The *Heterogeneous Multicore* architecture represents a clear example of *Heterogeneity-Aware* architecture. The *Application Heterogeneity* is matched with an heterogeneously distributed hardware, striving to assign each application to the heterogeneous execution core that best fits the application's needs. Since each execution core involves a different energy consumption, depending on the amount of resources and complexity involved, smartly assigning applications to cores improves the execution's *complexity-effectiveness*. However, due to the *migration cost*[8] the *granularity* at which the *Application Heterogeneity* can be detected in an *Heterogeneous Multicore* architecture is limited to hundreds of millions of execution cycles. This constraint may limit the amount of exploitable heterogeneity that may be detected.

In this thesis we present the *heterogeneously distributed SMT (hdSMT)* architecture. Based on a novel combination of *SMT*, *Clustering* and *Heterogeneity-Awareness*, this

[8]Each time an application is migrated to a different core its architectural state must be saved and moved. This process involves copying the registers that keeps the application architectural state. Besides, the content of the L1 caches is lost.

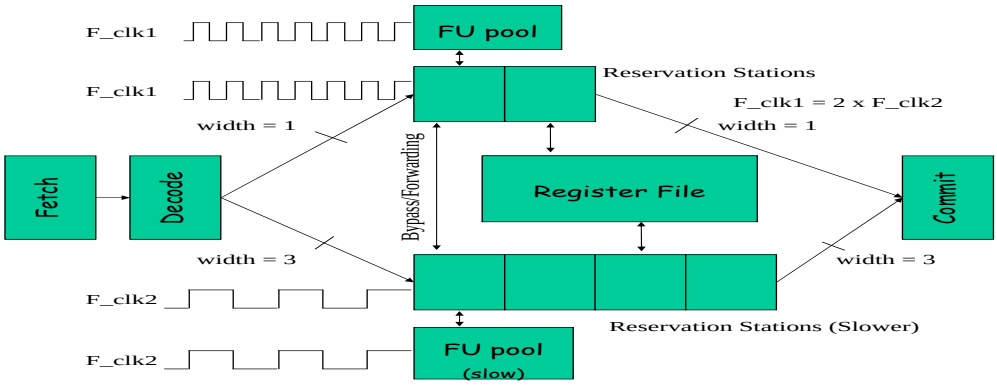


Figure 3.9: Dual Speed Pipelines Architecture.

Figure 3.10: Heterogeneous Multicore Architecture.

Heterogeneity-Aware architecture proposes a multithreaded alternative that lays on the spectrum that extends in between *SMT* and *CMP* processors. Thus, from an *SMT* point of view, the *hdSMT* could be defined as a clustered *SMT* processor comprised of an heterogeneous set of execution pipelines, which execute instruction streams fetched by a multithreaded fetch engine. From a CMP point of view, the hdSMT could be defined as an heterogeneous CMP processor, comprised of heterogeneous cores, in which some resources are shared. Among these shared resources we find the fetch engine, register file, and caches –even the L1 caches. In the following section it is deeply analyzed the *hdSMT* architecture, including an evaluation that confirms its *complexity-effectiveness*.

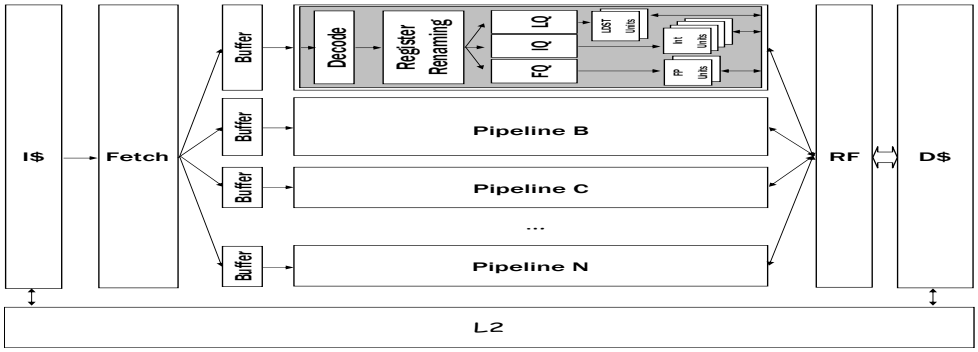


Figure 3.11: The hdSMT Architecture.

3.3 The hdSMT Architecture

The foundations of the *heterogeneously distributed SMT (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 [21], there are multiple possible hardware configurations in between *SMT* and *CMP* processors. As it is augmented the amount of shared resources among the hardware contexts available in the processor, it is covered the distance between a *CMP* processor, which typically only shares the L2 cache, and an *SMT* processor, which shares all the available resources. As Collins et al. indicate in [21], it may be achieved the best of both approaches by clustering some of the main processor structures in an *SMT* processor. However, the *Application Heterogeneity* may turn some of the evenly clustered approaches in [21] into not optimal. The *hdSMT* architecture maximizes fully exploitation of the available hardware budget by partitioning the hardware into heterogeneous clusters. The *Application Heterogeneity* is then matched with this heterogeneously distributed hardware, assigning to each application the cluster that best fits its resources needs.

The *hdSMT* architecture overview is depicted in Fig. 3.11. As in a conventional *SMT* processor, all threads share the caches, register file, and fetch engine. However, the remainder execution pipeline stages and resources are arranged in heterogeneous clusters (or simply *pipelines*). That is, each *pipeline* comprises all the execution 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, that could be shared among more than one thread; that is, each *pipeline* may be multithreaded. Consequently, the maximum amount of threads that can be simultaneously run on an *hdSMT* processor

is not equalt to the amount of *pipelines*, but the sum of *SMT* hardware contexts spread over all the constituent *pipelines*. The size and number of processor 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 the *Application Heterogeneity*. This implies that there are no dependencies[9] 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 workload from the list of ready tasks. At this time, just after being assigned the applications to the pipelines by the OS Job Scheduler and just before starting the execution itself, the Program Counter and the remainder architectural state is updated in each pipeline in the same way as in a *monolithic*[10] *SMT* processor. In order to determine on which specific *pipeline* would be executed each application it is triggered a hardware-based mapping policy (see Section 3.3.1). Whenever an application is assigned to the very same *pipeline* it was in the exactly previous *OS quantum* of execution, no additional changes within the *pipeline* should be made. Otherwise, the *pipeline* is flushed in order to accommodate the new execution thread. The whole subsequent workload's execution is done according to this mapping, without any intermediate thread migration. Notice that the reduced *migration cost*[11] between different *pipelines* provided by the *hdSMT* architecture allows to implement mapping policies which work at lower granularities[12]. The mapping policy implemented in *hdSMT* is described in detail in Section 3.3.1.

The number of hardware contexts and width (i.e., max. number of instructions issuable per cycle) may vary from *pipeline* to *pipeline*. So, an *hdSMT* implementation may be comprised of both narrow single-thread and wide multithreaded pipelines, as well intermediate pipelines. Depending on the workload size, 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

[9] We use exclusively independent threads in our experiments. Multithreaded applications are left for future work.

[10] That is, a conventional non-partitioned processor.

[11] Since both the caches and the register file are shared among all the *pipelines*, migrating an application to a different *pipeline* only involves re-fetching the in-flight instructions.

[12] In an state-of-the-art Operating System like Linux kernel 2.6 the length of the OS execution quantums may vary from a few tens of millions of cycles to several hundreds or even thousands of millions of cycles.

does not reach the number of hardware contexts. This is also applied in the Heterogenous MultiCore architecture [37], 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 *IFetch Policies* not available in conventional and monolithic *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* accepts 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. 3.11). The fetch engine inserts instructions at its own rate while each pipeline extracts 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 characteristics of the set of *pipelines*, this may result in a wider global decode bandwidth since all *pipelines* are fed from their private buffer each cycle.

3.3.1 Mapping policies in hdSMT

The impact of statically partitioning the hardware into homogeneous clusters may be either productive or counterproductive, depending on the resource partitioned. Raasch et al. show in [51] that for storage resources, such as the instruction queue and reorder buffer, statically allocating an equal portion to each thread provides good performance, in part by avoiding starvation. Additionally, the enforced fairness provided by this partitioning obviates sophisticated fetch policies to a large extent. The *SMTs* potential ability to dynamically allocate storage resources across threads does not appear to be of significant benefit. In contrast, an static division of the issue bandwidth has a negative impact on the system throughput. The *SMTs* ability to dynamically multiplex bursty execution streams onto shared function units contributes to its overall throughput.

In the *hdSMT* architecture the hardware is heterogeneously distributed into different clusters. As a consequence, both the storage resources and the issue bandwidth are statically distributed among all the constituent clusters. Since each of this static partitions has a different size, the success in avoiding the negative effects of such a partition depends on the *mapping policy*. Its ability to map high-performing threads to wide pipelines, to better profit both their wider issue bandwidth and higher amount of resources, determines the overall system performance. The *memory behavior* of each running application is used as an indicator of the resource needs of each application. Threads with *good* memory behavior are assigned to bigger hardware partitions or *pipelines*. In this sense, we define

the goodness of an application's memory behavior as the amount of L1 data cache misses; the lesser L1 data cache misses the better the memory behavior of that specific application. To classify each running applications according to their memory behavior without adding excessive overhead, it is used profile information of the number of L1 data cache misses of each application for a similar execution interval. This profile information may come either from a prior off-line application's execution or from a prior OS quantum of execution. Since applications go through different program phases throughout their executions, with very different behaviors, the closer the behavior exhibited during the profile information fed from the OS to the *hdSMT* mapping policy to the real behavior the better results would be obtained. In this PhD dissertation we assume the OS to successfully feed the hdSMT mapping policy with appropriate profilings[13].

By means of this profile information fed by the OS, the active threads are arranged by the number of data cache misses and assigned to the *pipelines*. The pipelines present in the microarchitecture are also arranged, but in this case by the width of the pipeline. Then, threads are mapped to the pipelines starting from the thread with the lower misses count and from the widest pipeline. Recall that the OS is suppossed to select a workload with a maximum execution thread count lower or equal to the sum of hardware contexts, spread over all the constituent pipelines in an *hdSMT* processor.

The proposed hdSMT mapping policy assigns as many threads per pipeline as hardware contexts it has got. If a pipeline does not admit more threads, the mapping policy continues assigning threads to the next pipeline in the list. The only exception to this simple rule is the first thread. Whenever possible, the first thread is mapped alone in the first pipeline. The rationale to this procedure is to prevent the resource competition between the highest-performing thread and other simultaneosly running threads. Since the highest-performing thread is the one which contributes the most to the final processor throughput, isolating it improves the overall processor throughput.

Regarding the interaction between applications, it is assumed that applications with a similar number of L1 data cache misses behave similarly and therefore can share a single pipeline without involving counterproductive contention. Thus, the negative scenario in which applications with a bad memory behavior hinder the forward progress of applications with a better memory behavior is avoided. In this sense, our mapping policy assumes that adjacent applications in the list *T* behave similarly and consequently can share a single *pipeline*.

[13]During *hdSMT* research we employed information from a single profile for each application used, corresponding to a complete execution of the *trace* (see Chapter 2) of instructions, selected for each application according to[55].

In order to match each application with the most appropriate *pipeline*, and so adequately matching the *Application Heterogeneity* with the heterogeneously partitioned hardware, our mapping policy makes the following simple assumption: *"the number of L1 data cache misses of an application is inversely proportional to the required pipeline width"*. The more L1 data cache misses occurred during an application execution interval the more resources will be held by that application while each miss is resolved, hindering other applications from making forward progress using those resources. 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. The full mapping process of the profile-based heuristic policy employed is detailed following in pseudo-algorithmic form:

1. Arrange all active threads, by the number of L1 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.

Our results show that the effectiveness of the mapping policy depends on each specific *hdSMT* microarchitecture it is designed for. Hence, as we will see in Section 3.3.5, a single *hdSMT* mapping policy can not obtain the optimal results for all *hdSMT* microarchitectures. The proposed *hdSMT* mapping policy described in this section, and used in the evaluation of the *hdSMT* architecture itself, is aimed at the *hdSMT* microarchitecture evaluated with the best performance per area ratio (i.e. *2M4+2M2*). For this *hdSMT* microarchitecture the proposed *hdSMT* mapping policy exhibits an average 92% accuracy. Consequently, obtaining an appropriate *hdSMT* mapping policy for each specific *hdSMT* microarchitecture opens new lines of research, left for future work.

3.3.2 Area Cost Model

Several microarchitectures were evaluated during *hdSMT* research. Each of these microarchitectures involve a different hardware budget. Since an straight comparison of the results obtained for microarchitectures with a different amount of resources may be quite unfair, some additional complexity measurement is needed in order to guide this evaluation. However, quantifying complexity is a tricky task and giving a single and comparable measurement is even harder to accomplish. In this research it is followed a quite generalized approach which uses 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 it is used the *Karlsruhe Simultaneous Multithreaded Simulator* [59, 64, 65]. 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, they are removed from the *area cost model* to simplify the results. However, since in *hdSMT* these resources are shared among all *pipelines*, the cost of the additional access logic is taken into account. It is added to the execution core of each *pipeline*, as additional hardware for multiplexing the 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. In fact, some *hdSMT* implementations, while requiring less area, provide more hardware contexts than a monolithic *SMT* processor. Taking into consideration Burns and Gaudiot's work in quantifying the *SMT* layout overhead [16, 17], we have extrapolated single to multipipeline environment area overhead from single to multithreading environment. Thus, we have estimated the additional 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, four different models of *pipeline* are used, named *M8*, *M6*, *M4*, and *M2*. The number in the name of each model gives a hint of the amount of resources assigned to the corresponding *pipeline*. The *SMT* baseline, or *monolithic SMT* since its hardware is not partitioned into clusters, is represented by the *M8 pipeline*. The remainder models represent *pipelines* with reduced resources budget with respect to the baseline. Starting from the *M8* processor model, basically we estimated the hardware budget of the remainder processor models by dividing the original hardware budget by two (i.e, once per *M4* and twice per *M2*). The *M6* processor model was introduced as an intermediate step in between *M8* and *M4* models to allow some further high-performing *hdSMT* microarchitectures.

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*'s new values, it was chosen in each case the number of functional units that kept a performance slowdown below 2%. The resulting amount of processor resources assigned to each *pipeline* is shown in Figure 3.12.(a). Additionally, it is used a private 256-entry *Reorder Buffer (ROB)* per each thread in all configurations, both *SMT* and *hdSMT*.

Except for the *monolithic SMT* baseline, all the configurations evaluated are comprised of a set of *pipelines*. For each of these *pipelines*, the area estimation is obtained from the sum of the area occupied by 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 pipeline* models, only one instruction fetch stage is included in the total area calculus, since it is shared among all the constituent *pipelines*. According to the defined *area cost model*, the area estimation for each of the configurations evaluated is shown in Figure 3.12.(b). All estimations have been made in 0.18 μm, as in [16], to ease our area overhead extrapolations. Notice in Figure 3.12.(b) that the area estimation for the *M6*, *M4*, and *M2 pipelines* includes an instruction fetch stage a 20% bigger than the one included in the baseline (*M8*). Each of these bars may be considered as an *hdSMT* processor comprised of a single *pipeline*, the one measured in each case.

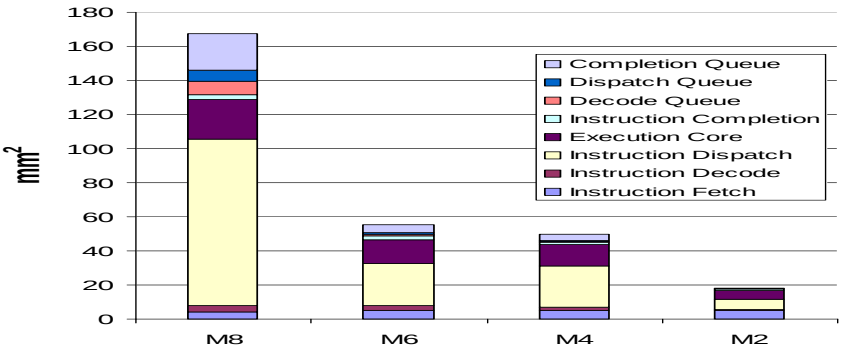
Finally, as shown in Figure 3.12.(a), our *monolithic SMT* baseline (*M8*) is not able to execute more than four threads simultaneously. Although adding additional hardware contexts increases the total area occupied by an *SMT* processor, as evaluated by Burns and Gaudiot in [16], our *area cost model* does not assume any additional area overhead for the baseline in case of executing more than 4 threads simulatenously. As a consequence, the used *area cost model* favors the baseline when 6-thread workloads are evaluated, which require two additional hardware contexts to the *M8* model.

3.3.3 Simulation Setup

Additionally to the reference simulation parameters shown in Chapter 2 (Table 2.4) Figure 3.12.(a) shows the main characteristics of each *pipeline* model used in *hdSMT* experiments. In both *monolithic* and *multipipeline* configurations, the register file is shared among all the threads running on 8-stage execution *pipelines*. Any of the *integer* and *load/store* functional units included in each *pipeline* is connected to each of the *12-read and 6-write ports* of the *integer register file*. FP functional units are connected to each of

	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.

Figure 3.12: Pipeline models.

the *6-read and 3-write ports* of the *FP register file*. Since the amount of functional units included in a *multipipeline* configuration, obtained as the sum of the functional units included in each of the constituent *pipelines*, may be much higher[14] than the baseline's, the access to the read/write ports of the shared register files is multiplexed in these configurations. In order to model the cost of the additional logic, required to handle the multiplexed accesses to the shared register file ports, it is doubled the number of cycles required by any register read/write in multipipeline 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 multipipeline configurations.

In the experiments related to this chapter, it is adopted the *FLUSH* [70] instruction fetch policy for the baseline (M8) case. Built on top of *ICOUNT 2.8* [72], that prioritizes

[14]Wider *pipelines* are prioritized in case of register file port contention in a single cycle. That is, read/write accesses from wider pipelines are served first.

threads according to the number of instructions in the preissue stages, this *SMT* fetch policy predicts an L2 cache miss every time a load spends more cycles in the cache hierarchy than needed to access the L2 cache. It is used a *fixed 30-cycle trigger*, according to the memory simulation parameters shown in Table 2.4. That is, whenever a load instruction last more than 30 cycles to be resolved, an L2 cache miss is predicted. Whenever an L2 cache miss is predicted, the instructions after the L2 missing load are flushed away from the execution pipeline, and the offending thread is stalled until the load is resolved. As a consequence, the resources used by the offending thread are freed and it does not compete for new resources until the load is resolved. This allows the other threads to proceed and use the freed resources to make forward progress, while the stalled thread is waiting for the outstanding cache miss.

For the case of *M6*, *M4* and *M2* pipelines, it is adopted the *L1MCOUNT* fetch policy, a variant of the *DCache Warn* fetch policy [19]. This *SMT* 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 *SMT* 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 the specific characteristics of each *pipeline*, it is allocated a buffer in between the fetch engine and each *pipeline* (see Fig. 3.11). The size of these buffers is 32 entries, for *M6* and *M4* pipeline models, and 16 entries, for *M2* pipeline model.

In our experiments, it is employed the *SPECINT2000 benchmark suite*. From them, we have collected traces of the most representative 300 million instruction segment of each benchmark, following the idea presented in [55]. 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 3.2 and 3.3 show the workloads used, 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. The reason to focus on *SPECINT2000* benchmarks, not including others as *SPECFP2000*, is to delimit the *Application Heterogeneity*. We seek to show that even within an apparently homogeneous, and not highly parallel, suite of applications there is enough *Inter-Application Heterogeneity* to be exploited by an *hdSMT* architecture. In any case, we believe that even in the lack of these *MEM* workloads our results are significative enough to reach interesting conclusions.

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 3.2: 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 3.3: Six threaded workloads.

In each experiment, it is strictly focused on the period of time in which all the initial threads share the processor. The objective in each case is to evaluate the behavior of each microarchitecture with workloads comprised of two, four and six threads. This means that each simulation finishes as soon as any of the threads contained in the evaluated workload finishes executing 300 million instructions.

3.3.4 Microarchitectures and Metrics

In our experiments, several multipipeline microarchitectures are evaluated, both homogeneously and heterogeneously distributed. All these multipipeline microarchitectures are implementations of the *hdSMT* architecture[15], with a shared fetch unit feeding all the constituent *pipelines*. The area estimation for each of the microarchitectures evaluated is shown in Figure 3.13. The name of each microarchitecture, below each area estimation in Figure 3.13, indicates the number and type of *pipeline* models involved in each case. Thus, the *2M4+2M2* microarchitecture is comprised of two pipelines of type *M4* plus two pipelines of type *M2* (see Figure 3.12 for an area estimation of each pipeline type). From left to right, the first microarchitecture (*M8*) in Figure 3.13 represents our *monolithic SMT*

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

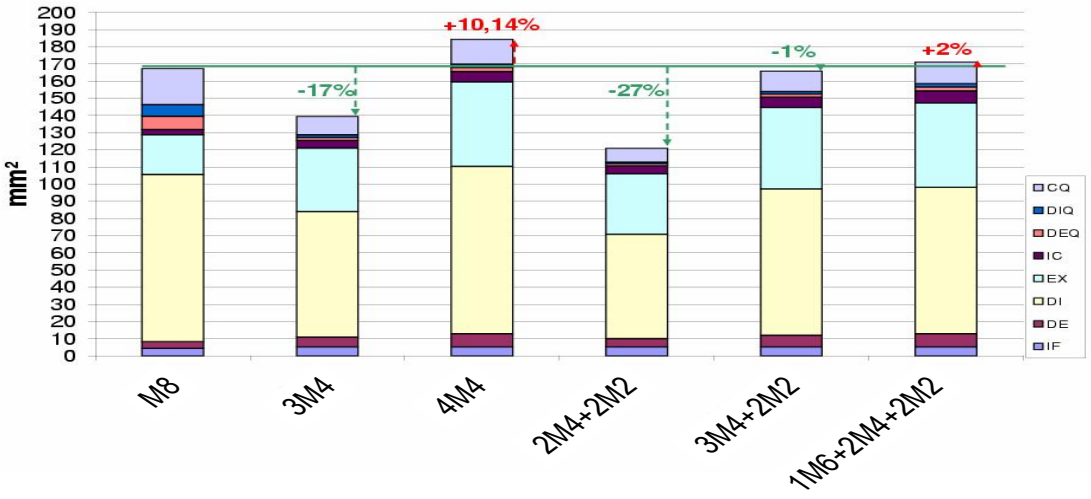


Figure 3.13: Area estimation of evaluated microarchitectures.

baseline. The next two microarchitectures (*3M4* and *4M4*) are homogeneously clustered *hdSMT* microarchitectures; a shared fetch engine feeds multiple pipelines of the same type. Finally, the last three microarchitectures represent the truly *hdSMT* microarchitectures, with multiple pipelines of different types comprising the system. According to Figure 3.13, 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 and workload it is evaluated the performance obtained; measured in *Instructions Per Cycle (IPC)*. 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, we additionally provide results measured in *Performance per Area*, which is obtained 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. Whilst comparing raw performance may lead to unfair comparisons, only justified in case of unlimited resource budgets[16], in a more general case the processor design may obey stricter complexity constraints; both in terms of area, power and thermal disipation.

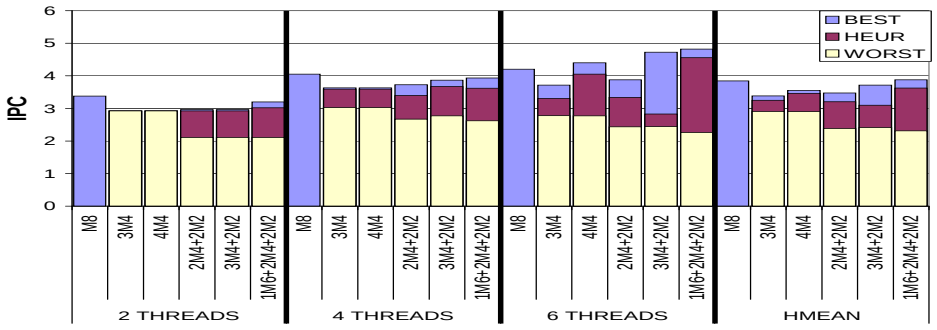
[16]Whenever a microarchitecture is designed with the only purpose of maximum throughput, the complexity involved in the processor's design may be obviated.

3.3.5 Simulation Results

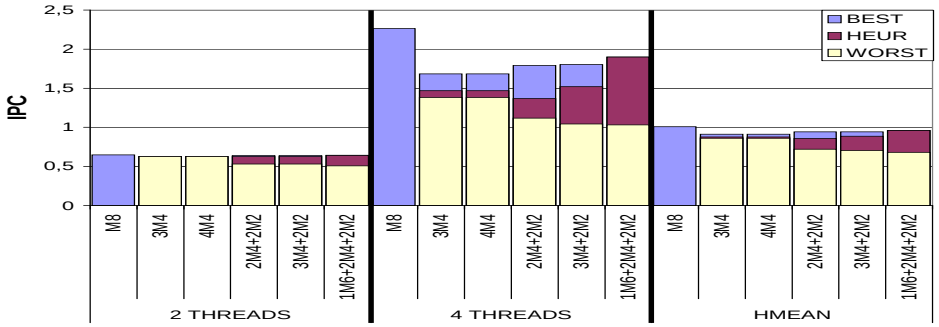
In this section we evaluate and compare *monolithic SMT*, *homogeneously distributed hdSMT*, and *heterogeneously distributed hdSMT processors*. For each workload, three measurements are given. First, the *BEST* result, obtained using an *oracle*[17] thread mapping policy, gives the *maximum performance* of the microarchitecture. Second, the *HEUR* result gives the performance obtained by the microarchitecture using the *heuristic* thread mapping policy presented in Section 3.3.1. 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-thread 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-thread workloads, when all pipelines are of the same sort the three measurements (*BEST*, *HEUR*, *WORST*) coincide.

Figure 3.14 shows the *raw performance* results (measured in *IPC*) for all microarchitectures evaluated. In each case, it is shown the *harmonic mean* of all workloads of a same type and size. These results point out that, although some *hdSMT's* results are quite similar to *SMT baseline* ones, the *hdSMT's* 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) is not able to execute more than *4 threads* (as mentioned in Section 3.3.2). Recall that the maximum amount of threads that an *hdSMT* microarchitecture is able to execute simultaneously is equal to the sum of hardware contexts of each constituent *pipeline*. Hence, according to Figure 3.12.(a), the *2M4+2M2 hdSMT* processor is able to execute up to 6 threads simultaneously, whilst *1M6+2M4+2M2* can handle up to 8 threads simultaneously. Nevertheless, the ability to flush and re-execute instructions of the baseline (M8) 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-thread *ILP* workload scenario (see Figure 3.14.(a)).

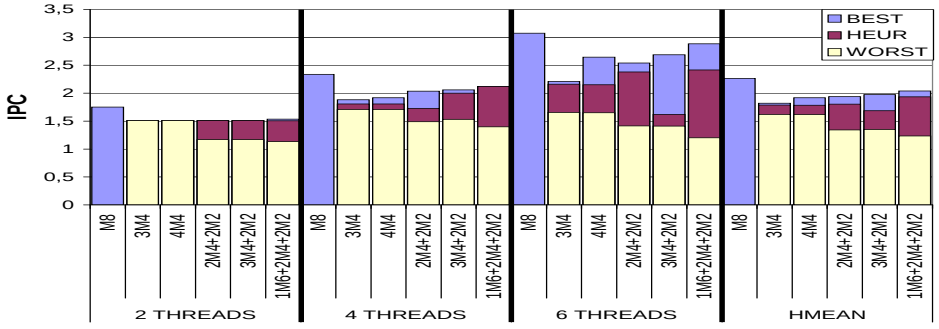
[17] Obtained using brute force, that is simulating all different application-to-pipeline assignments and choosing the one with the highest value.



a) ILP Workloads.

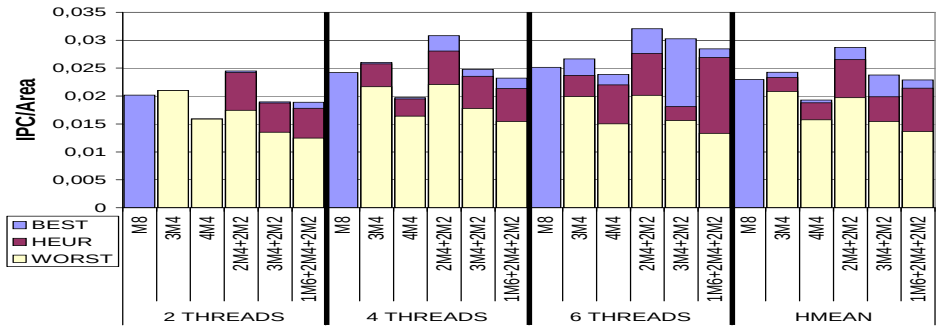


b) MEM Workloads.

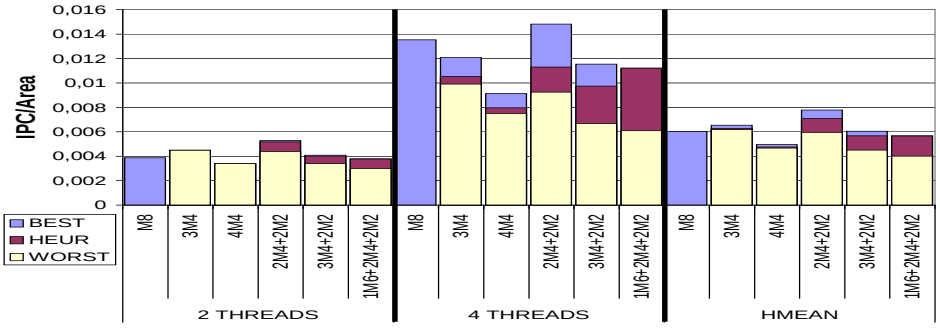


c) MIX Workloads.

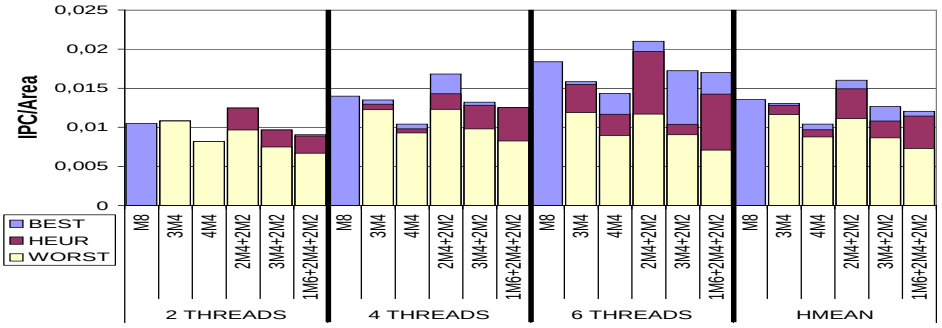
Figure 3.14: Performance comparison.



a) ILP Workloads.



b) MEM Workloads.



c) MIX Workloads.

Figure 3.15: Performance per Area comparison.

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 Figure 3.15 the *Performance per Area* results for all microarchitectures evaluated. Again, it is shown the *harmonic mean* of all workloads of the same type and size. 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. The rationale behind these results, that clearly indicate that *hdSMT* microarchitectures are more *complexity-effective* than a *monolithic SMT*, can be found on the amount of resources needed to execute each application; namely *Inter-Application Heterogeneity* (see *Heterogeneity Analysis* shown in the first half of this chapter). Since processor resources are *statically partitioned* among all the constituent *pipelines*, and the mapping policy employed is *static* (i.e., only reassigns applications to pipelines on a OS context switch granularity) the results highly depend on the mapping policy. Hence, good results would come from the ability to accurately matching the application's hardware requirements and each pipeline's hardware budget during each OS quantum of execution.

Regarding the *homogeneous* (*3M4*, *4M4*) and *heterogeneous* distribution (*2M4+2M2*, *3M4+2M2*, *1M6+2M4+2M2*) of *hdSMT* processors, results in Figures 3.14 and 3.15 point out that *heterogeneous distributions are better*[18] *than homogeneous ones*. Thus, *for each case there is at least one heterogeneous distribution that overcomes*, both in terms of absolute performance and performance per area, *all homogeneous distributions*. However, homogeneous distributions represent an easier scenario in terms of mapping policies. Since all partitions have the same amount of processor resources, the amount of *different* assigments drastically decreases, leading to a more easy-to-assign scenario.

From all previous results it may 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 Figures 3.14 and 3.15. 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 Figures 3.14 and 3.15 it is also noticeable that *the effectiveness of the mapping policy depends on the specific* hdSMT *microarchitecture*. Thus, while the heuristic applied achieves *92%* and *96%* accuracy in

[18] In terms of *complexity-effectiveness*.

2M4+2M2 and *1M6+2M4+2M2* microarchitectures respectively, its accuracy drops to a *88%* in *3M4+2M2* microarchitecture. As a consequence, Figures 3.14 and 3.15 clearly indicate that when designing *hdSMT* microarchitectures it must be payed special attention to the design of the *hdSMT* mapping policy. Although covered in some sense in multithreaded multicore scenario (see Chapter 4), a deep analysis of mapping policies in clustered scenarios is left for future work.

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*.

3.4 Chapter Summary

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 chapter we have deeply analyzed the heterogeneity in software and its reflect on the hardware itself. From this analysis we have settled the foundations of the first contribution of this thesis: the *Heterogeneously Distributed Simultaneous Multithreading (hdSMT) architecture*.

The *hdSMT* architecture is an *SMT* alternative architecture in which the running threads are mapped to a heterogeneosly clustered hardware according to this heterogeneity. The results obtained in the evaluation of this first contribution indicate that the *hdSMT* reduce the waste of resources at reduced budget, obtaining *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 chapter, 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 the program behavior during execution. In this sense, the conclusions obtained from the heuristic mapping policy proposed in this chapter shed some light into the characteristics of future mapping policies over clustered layouts, opening new and interesting research topics for future research.

Chapter 4

Heterogeneity-Awareness in Multithreaded Multicore Processors

Once analyzed the *Heterogeneity-Aware* concept and its application to the single-core scenario we move to the multicore scenario. In particular, we focus on the *Multithreaded Multicore Processors*, a sort of processors that seem to constitute a general trend in industry nowadays. So, state-of-the-art high-performance processors like the *IBM POWER5* and *POWER6*, comprised of two cores with two *SMT* hardware contexts each (i.e., an overall four hardware contexts count), confirm this trend. In these processors, the set of applications selected by the *Operating System* to be simultaneously executed must be assigned to one of the available hardware contexts, distributed among all available cores. We call to this intermediate step the *Thread to Core Assignment (TCA)*.

In this chapter we show the relation between the *Thread to Core Assignment (TCA)* and the underlying *Instruction Fetch (IFetch) Policy*, implemented in each *SMT* core. On the one hand, we show that the performance of a given *TCA* depends on the underlying *IFetch Policy*. On the other hand, the *TCA* determines the performance of the underlying *Instruction Fetch (IFetch) Policy* implemented in each *SMT* core. We include evidences which indicate that a good *TCA* can improve the results of any underlying *IFetch Policy*, yielding speedups of up to 28%.

Given the relevance of *TCA*, we propose an algorithm to manage the *Thread to Core Assignment* in Multicore processors comprised of *SMT* cores. The proposed *TCA Algorithm* boosts system throughput, taking into account the *workload characteristics* and the underlying *SMT IFetch Policy*. It achieves its goal, yielding system throughput improvements up to 21% as compared to the state-of-the-art *TCA* policy in current processors.

4.1 Introduction

Process technology advances have considerably increased the amount of available transistors on a single chip; and this count does not seem to stop increasing in the next years. However, having more available transistors can not always be directly translated into increasing the processor performance. The limitations imposed by the *Instruction Level Parallelsim* (*ILP*) have made *Thread Level Parallelism* (*TLP*) becoming a common strategy to improve processor performance. Multithreaded processors execute multiple applications to better profit the available hardware resources. Since it is difficult to exploit more *ILP* from a single application, Computer Architects have opted to exploit other parallelism sources.

As we saw in Chapter 1, there are multiple multithreaded alternatives, depending on the granularity of the *TLP* exploited. According to the specific *MT* alternative chosed for an specific microarchitecture, the *Operating System (OS)* has a different task when selecting the execution workload for each *OS context switch*. Therefore, the *OS* scheduling process depends on the specific *MT* alternative implemented in the processor. In this chapter we show how the *Heterogeneity-Awareness* concept could assist the *OS* when performing this task, in order to achieve subsequent *complexity-effective* executions.

In an *SMT* processor, the *scheduling process* is comprised of two main steps. Assuming *M* runnable jobs, in the first step the *Operating System (OS) Job Scheduler* selects a set of *N* from these *M* jobs: the workload (*N* is less or equal to the number of hardware contexts of the *SMT*). This first scheduling layer is known as *co-schedule selection* [33]. Once the *OS* composes the workload, the resource allocator of the *SMT* processor decides how to prioritize threads. Usually the resource allocation is carried out by the *IFetch Policy* [20, 24, 70, 72]. This second scheduling layer is known as *resource sharing* [33].

In a *CMP* processor comprised of *SMT* cores, like the *IBM POWER5* [60] and *POWER6* [39], the traditional *SMT scheduling process* requires an additional intermediate step. Once the *OS* selects the applications to schedule together in the processor (*co-schedule selection*) each application must be assigned to one of the execution cores. We call this additional scheduling step the *Thread to Core Assignment (TCA)*. Then, the underlying *IFetch Policy* manages the resource distribution (*resource sharing*) between the applications assigned to the same core. In current *OS* like *Linux 2.6* [14], the *TCA* does not have a significant role when co-scheduling threads. Basically, the decision whether a job has to be scheduled in a given core depends on the fact whether that job was *recently* executed in that core and hence can take profit of the data that could remain in the cache. Nevertheless, thread migrations between execution cores can be triggered by the *OS* for load balancing purposes, losing the data in the cache.

In *SMT* processors, the *IFetch Policy* is usually designed to address a particular situation where the performance of *SMT* degrades significantly. As an example, the *FLUSH* [70] policy avoids the situation where a *memory-bounded* thread clogs the internal resources of the *SMT* causing performance degradation. These policies improve performance in workloads comprised of both *memory-bounded* and *ILP-bounded* threads.

In this chapter, we analyze the new intermediate step, the *TCA*, that we have identified in the *OS* scheduling process of current *CMP* processors comprised of *SMT* cores (*CMP+SMT*). This analysis reveals that the *TCA* heavily affects the performance of the underlying *Instruction Fetch (IFetch) Policy*. Our analysis focuses on the relation between the *TCA* and *IFetch Policy* scheduling layers. That is, we assume a fixed workload after some *co-schedule selection*[1]. The results indicate that a bad *TCA* may negate the performance advantage of a *robust*[2] *IFetch Policy*, like the *FLUSH* [70] and *STALL* [70] policies. That is, if we continue designing multithreaded multicore processors without having into account the *Heterogeneity-Awareness* concept, we would go away from *complexity-effective* processor designs; further as the on-chip transistor count increases.

The importance of the *TCA* lies in the fact that it can prevent the situation in which *SMT* suffers from performance degradation by appropriately assigning the threads to co-schedule in each *SMT* core. Thus, *a good* TCA *reduces the need of a* robust IFetch Policy. Oppositely, *a bad* TCA *may cause a* robust IFetch Policy *to perform poorly*. An illustrative example is shown in Figure 4.1. It depicts the throughput of a 2-core processor with 2 hardware contexts per core, using the *ICOUNT* and *FLUSH* policies, respectively (See Section 4.2 for core details). The applications in the workload (*A,B,C,D*) are assigned to the *SMT* cores (e.g., [A,B] = *A* and *B* assigned to the same core). Notice in Figure 4.1 that while the first *TCA* yields similar results for both policies, the second *TCA* obtains an improvement of 19%. Consequently, a *good TCA* policy is required in order to fully exploit the benefits of the underlying *IFetch Policy* in *CMP+SMT* processors.

As the amount of *SMT* cores within *CMPs* increases, the number of possible *TCAs* exponentially grows. Assuming 2 hardware contexts per *SMT* core, there are 3, 105, and 280 millions of different *TCAs* for 2, 4, and 8-core implementations, respectively. Therefere, the selection of a *good TCA* for each case should not involve an excessive overhead, proportional to the number of cores. Otherwise, this selection scheme would not scale with the ever-growing amount of on-chip cores. This chapter presents the second contribution of this thesis, that can be break down into:

[1] Certainly, there is a relation between *co-schedule selection* and *TCA*, but is out of the scope of this PhD dissertation, and left for future research.

[2] The term *robust* is employed in this chapter to refer IFetch Policies which appropriately handle long-latency loads.

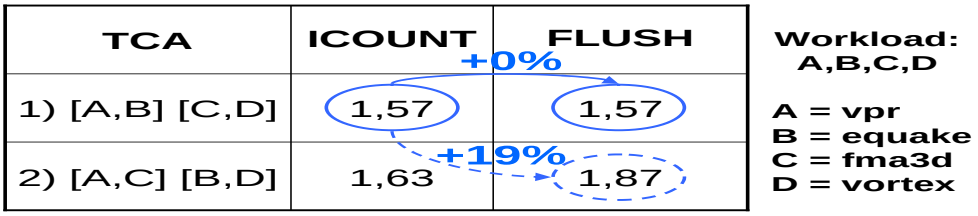


Figure 4.1: TCA Example.

1. *ANALYSIS.-* We analyze the *scheduling process* in *CMP+SMT* processors. We identify the need of a new intermediate step in the scheduling process: *Thread to Core Assignment (TCA)*. We also do the first analysis in the literature[3] of the relation between this *TCA* and the *IFetch Policy*. Our results indicate that a proper *TCA* allows a *naive IFetch Policy*, like *Round Robin* [72], yields similar throughput results to those obtained with a *robust IFetch Policy*, like *FLUSH*. We show results which indicate that a *good TCA* can yield speedups of up to 28%. Therefore, the TCA Algorithm supposses a significative improvement in terms of complexity-effective executions, since *naive* policies generally consume less energy than more *robust* ones.

2. *PROPOSAL.-* We propose the *TCA Algorithm*, which applies the *Heterogeneity-Awareness* concept to the *Multithreaded Multicore Processors*. It selects an appropriate *TCA* for each case, according to the *workload characteristics* and the underlying *IFetch Policy*. Its simple design allows a real implementation without adding excessive overhead, just requiring the number of *committed Instructions Per Cycle (IPC)* during a prior and representative portion of execution. To assist the *TCA Algorithm* with these *IPC* values, we also propose an *IPC prediction mechanism*: the *TCA Calibration*. This simple but effective mechanism predicts the *relative behavior* (measured in *IPC*) of the running applications using an small *sampling phase*. The obtained *IPC* values, whilst not fully accurate, catch the relative behavior of the running applications. Feeding the *TCA Algorithm* with these *IPC* values, we show evidences which indicate that the proposed *TCA Algorithm* obtains assignments 3% close to the optimal assignation for each case, yielding system throughput improvements up to 21%. Besides, the *TCA Algorithm* accuracy scales with the workload size and number of on-chip *SMT* cores.

Simulation Parameters				Benchmarks					
Pipeline depth	11 stages	L1 I-Cache	64KB, 4-way, 8 banks	gzip	a	vortex	j	mesa	s
Queues Entries	64 int, 64 fp, 64 ld/st	L1 D-Cache	32KB, 4-way, 8 banks	vpr	b	bzip2	k	fma3d	t
Execution Units	4 int, 3 fp, 2 ld/st	L1 lat./miss	3/22 cycs.	gcc	c	twolf	l	sixtrack	u
Physical Registers	320 regs.	I-TLB ,D-TLB	512 ent. Full-assoc.	mcf	d	art	m	facerec	v
ROB Size*	256 entries	TLB miss	300 cycs.	crafty	e	swim	n	applu	w
Branch Predictor	perceptron	L2 Cache	4MB, 12-way, 4 banks	perlbmk	f	apsi	o	galgel	x
	(4K local, 256 pers)	L2 latency	15 cycs.	parser	g	wupwise	p	ammp	y
BTB	256 entries,	M. Memory lat.	250 cycs.	eon	h	equake	q	mgrid	z
	4-way associative			gap	i	lucas	r		
RAS*	100 entries								

Type	Workload	Type	Workload	Type	
4W1	b, q, t, j	8W1	d, l, b, g, h, j, a, f	16W1	d, l, b, g, m, n, r, q, i, j, c, f, k, e, a, h
4W2	l, n, o, e	8W2	b, g, m, n, a, h, w, p	16W2	l, l+1, l+2, l+3, g, g+1, g+2, g+3, k, e, a, h, o, p, s, t
4W3	r, i, f, p	8W3	m, n, r, q, f, j, e, h	16W3	b, n, b+1, n+1, b+2, n+2, b+3, n+3, o, p, s, t, w, u, x, z
32W1	d, l, b, g, m, n, r, q, m+1, m+2, b+1, b+2, q+1, q+2, g+1, g+2, i, j, c, f, k, e, a, h, p, s, w, o, h+1, j+1, a+1, f+1				
32W2	l, l+1, b, b+1, m, m+1, n, n+1, g, g+1, g+2, b+2, q, q+1, q+2, r, j, j+1, j+2, h, h+1, a, a+1, f, u, p, p+1, p+2, c, c+1, s, s+1				
32W3	d, b, b+1, b+2, n, n+1, n+2, q, m, m+1, m+2, m+3, l, l+1, l+2, l+3, u, h, h+1, h+2, h+3, j, j+1, f, a, a+1, a+2, p, p+1, w, w+1, f+1				

Table 4.1: Simulation parameters and Workloads. (resources marked with * are replicated per thread)

4.2 Methodology

We simulate *CMP* configurations using a a multibanked L2 Cache shared among all cores. Each core implements *SMT* with two hardware contexts. Each workload is simulated on a *CMP* comprised of $\frac{\text{threads}}{2}$ *SMT* cores (e.g., 8-thread workloads simulated on 4-core CMPs). Additionally, in order to assure a minimal cache share 16-thread and 32-thread workloads (16W & 32W) are simulated using a shared 6MB 6-banked L2 Cache (instead of 4MB 4-banked). Since a complete study of all benchmarks is not feasible due to excessive computational cost we have randomly chosen some of them. The workload size is denoted by the prefix *xW*, where *x* stands for the number of benchmarks involved. Table 4.1 shows the main simulation parameters and the chosed workloads.

Each workload is simulated employing 4 different IFetch Policies: *Round Robin (RR)* [72], *ICOUNT* [72], *STALL* [70] and *FLUSH* [70]; in all cases, simulations are executed for a *fixed interval* of *140 millions of simulation cycles*. In our simulations we assume this simulation interval as a single *OS* quantum of execution. Although an state-of-the-art general-purpose *OS* like *Linux 2.6* does not have a fixed-length *OS quantum* we make this assumption for simplicity reasons. The *Linux kernel 2.6* [14] establishes *OS quantums* with typical lenghts ranging from *0 to 800ms*. In a 4 GHz general-purpose processor,

[3]To the best of our knowledge, there is no prior publication that explicitly identified the need of an intermediate layer in the *OS* scheduling process for *Multithreaded Multicore Processors*.

like the *IBM POWER6* [39] [4], this would be translated into *OS quantums* with a duration of *0 to 3200 million of cycles*. The choice of *140 millions of cycles*[5] as fixed-length of the *OS quantums* in our simulations represent a commitment, since all applications in the workload executes without being interrupted by the *OS*.

Despite the *TCA Algorithm* is a *throughput-oriented* proposal, it must be said that the *fairness* concept can not be directly applyied to *TCA* in *CMP+SMT*, as done in prior *SMTs* for *IFetch Policies*. This is true since the proposed *TCA Algorithm* does not add any hardware/software mechanism to stop/interrupt the application's execution; it just assign threads to *SMT* cores. This is the main reason to not include in this research additional metrics like the *Harmonic Mean* or *Weighted Speedup*.

4.3 Scheduling in Multicored SMT Processors

In a *SMT* processor the *scheduling* of a set of tasks requires decisions at two levels, as shown on the left side of Figure 4.2. First, when the number of available ready tasks *M* is larger than the *T* hardware contexts supported by the *SMT* processor, we need to determine which tasks to co-schedule, that is schedule together. The *OS Job Scheduler* seletcs a set of *N* tasks (where N $\leq$ T) from the *M* ready tasks: the workload. This first scheduling layer is known as *co-schedule selection* [33].

Second, we need to perform the resource distribution among co-scheduled tasks in an *SMT* processor. The *OS* passes the workload to the hardware, which must decide how to distribute the *SMT* processor resources among all applications comprised in the workload. This distribution is aimed at avoiding resource monopolization by the running threads. This second layer is known as *Resource Sharing* [33], as shown on left side of Figure 4.2. There are several proposals in the literature [20, 24, 70, 72] to manage the *Resource Sharing*. These proposals improve the system throughput of an *SMT* processor solving the resource contention among all applications in the workload. In this research we focus on four *IFetch Policies*: *RR* [72], *ICOUNT* [72], *STALL* [70] and *FLUSH* [70]. Far from representing the state-of-the-art of *IFetch Policies*, we use them as an easy-to-explain example, since the proposed *TCA Algorithm* (see Section 4.5) do not degrade the execution's performance of the running threads within each *SMT* core. In fact, as will see in following sections, there is no a single *TCA Algorithm* implementation valid for all IFetch Policies, as there is not a single thread-to-core assignment valid for all cases. Consequently, there would be multiple possible *TCA Algorithm* implementations.

[4] Latest POWER6 implementations reach a frecuency of 4.7 GHz.

[5] According to Chapter 7 in [14], 140M cycles represents a reasonable choice for such an approximation.

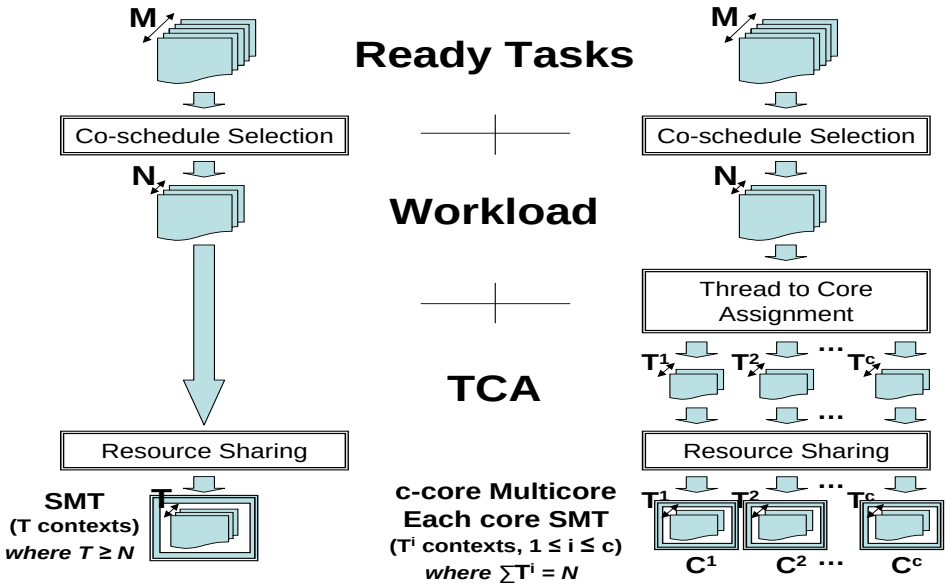
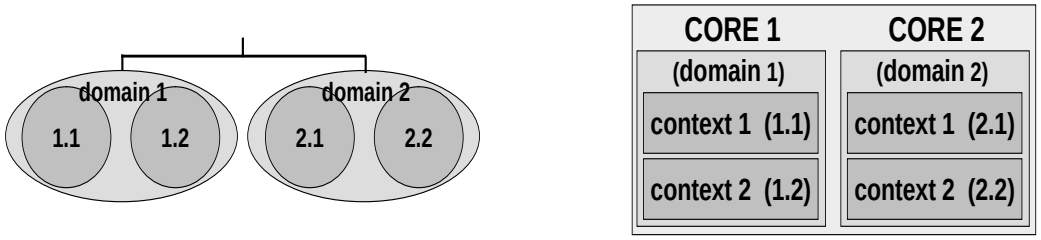


Figure 4.2: Scheduling Layers in SMTs and Multicored SMTs.

In a *CMP+SMT* processor, the T hardware contexts are distributed among all *SMT* cores, as shown on the right side of Figure 4.2. Each execution core works as a different *SMT* processor with its own resource allocation scheme. Consequently, we have to select which of the N applications from the workload to co-schedule in each *SMT* core, where $N = T$. In this way, the N applications from the workload are distributed among the c *SMT* cores. Since the multicore processor resources are statically distributed among all *SMT* cores, the way in which we schedule together tasks in each core determines the performance of the underlying *resource sharing* in each core. Obviously, the more the tasks (N) in the workload the more possible schedulings, or *TCAs*, growing exponentially with the number of tasks. The three layers of the task scheduling in multicore processors comprised of *SMT* cores are shown on the right side of Figure 4.2.

In an state-of-the-art general-purpose *OS* like the *Linux 2.6* [14] the *TCA* does not have a significant role in the *scheduling algorithm*. In fact, it is not explicitly taken into account. *Linux 2.6* considers each hardware context as a different *logical domain*. The *logical domains* are hierarchically organized according to the hardware contexts distribution on the chip. Figure 4.3 depicts an illustrative example for a 2-core *CMP* processor with 2 hardware contexts per core. Each *logical domain* has a different queue[6] of runnable

[6]Latest distributions of the *Linux Kernel 2.6*, like the *2.6.23*, manage these queues in a more sophisti-



(a) Hierarchical domains. (b) OS-HW domain mapping.

Figure 4.3: Linux 2.6 logical domains - Example in a CMP+SMT with 2 SMT cores.

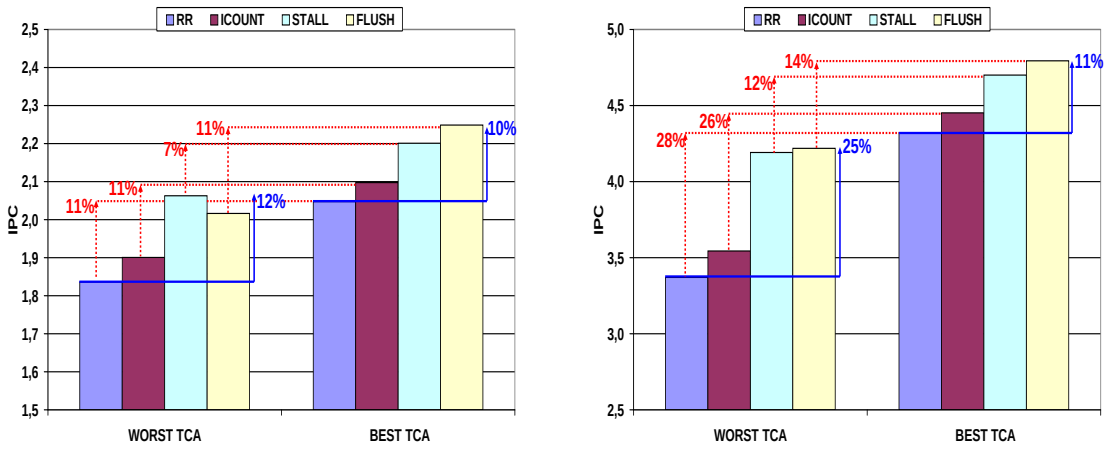
applications, sorted by process priority. In order to keep balanced these queues a *load balancing* process may be triggered, implying thread migrations from one core to another. Besides the process priority, the decision whether a job has to be scheduled in a given core basically depends on the fact whether that job was *recently* executed in that core, to take profit of the data that could remain in the cache. However, the load balancing process performed by the *OS* may involve thread migrations between execution cores, losing the data in the private caches.

4.4 Thread to Core Assignment and the IFetch Policy

In order to analyze the relation between the *TCA* and the underlying *IFetch Policy* we simulate all 4-thread (*4W*) and 8-thread (*8W*) workloads in Table 4.1 on 2 and 4-core *CMP*, respectively. Figure 4.4 breakdowns the results into *WORST* and *BEST TCA*. They correspond to the results obtained using the worst and the best TCA in terms of throughput (i.e., WORST TCA corresponds to the TCA that yields the worst throughput among all possible TCAs). Figure 4.4 shows for each IFetch Policy the average results obtained from the corresponding TCAs for all workloads with the same number of threads.

Figure 4.4 shows some interesting values on top of the graph itself. On the one hand, the percentages on top of each bar in Figure 4.4 indicate the throughput improvement achievable for the corresponding *IFetch Policy* using the *BEST TCA*, as compared to the throughput obtained using the *WORST TCA*. That is, the *relative importance of the TCA* or *TCA Sensitivity*. On the other hand, the percentages on the right side of each group of

cated *tree-based* structure.



(a) 4-thread workloads. **(b) 8-thread workloads.**

Figure 4.4: TCA Sensitivity.

bars indicate the relative importance of the *IFetch Policy* when using similar TCAs. That is, the throughput obtained using the *TCA* that yields the *WORST/BEST TCA* for each *workload* and *IFetch Policy*. Comparing both results it is straightforward that *the* TCA *has similar or even more importance in* CMP+SMT *processors than the* IFetch Policy *in* SMT *processors*. Four main conclusions can be inferred from the average results shown in Figure 4.4:

1. *A good IFetch Policy reduces the negative effect of an inappropriate TCA*. That is, when counterproductive threads are assigned to the same *SMT* core (i.e., inappropriate *TCA*) the goodness of the implemented *IFetch Policy* is of critical importance to obtain high system throughput. As a consequence, the impact of the *TCA* (or *TCA Sensitivity*) is on average lower in presence of good *IFetch Policies*. As a matter of example, in 8-thread workloads (Figure 4.4(b)) the *TCA's relative importance* ranges from *28%* to *12%* for *RR* and *STALL*, respectively.

2. *An appropriate TCA improves the results obtained regarless the underlying IFetch Policy*. The results in Figure 4.4 show that a *good TCA* improves the system throughput by more than 10% even in presence of a *robust IFetch Policy*, like *STALL* and *FLUSH*.

3. *An inappropriate TCA could negate the performance advantange of a better IFetch Policy*. That is, the *TCA* should not be neglected even when implementing *good IFetch Policies*. As an example, in both 4 and 8-thread workloads (Figure 4.4) the results obtained with *RR*, using *BEST TCA*, surpass those obtained using a better *IFetch Policy* like *FLUSH*, using *WORST TCA*. Therefore, *simply investing in good IFetch Policies does not assure the best results*.

4. *There is not a single TCA good for all cases*. As a matter of example Figure 4.5 shows the results yielded by workload *4W2* (see Table 4.1) using each *IFetch Policy* considered and two different *TCAs*. While *TCA 1* yields the *BEST TCA* results for *RR* and *ICOUNT* policies and the *WORST TCA* results for *STALL* and *FLUSH* policies, *TCA 2* yields just the opposite results.

Finding the *BEST TCA* for each case is not a trivial task since the number of possible *TCAs* exponentially grows with the number of *SMT* cores. As a matter of example, there are 105 different *TCAs* for 8-thread workloads using a 4-core *CMP+SMT* processor. Some of them yield the highest throughput and are considered as *BEST TCA*. The remainder *TCAs* may incur in some throughput loss as compared to the *BEST TCA*. Since state-of-the-art OS like the *Linux Kernel 2.6* does not explicitly take into account *TCA*, a *RANDOM TCA* is assumed as *state-of-the-art TCA policy*. As shown following, randomly selecting a *TCA* may incur in significant throughput losses.

Figure 4.6 shows the probability of throughput degradation due to randomly obtaining the *TCA* for each 8-thread workload and IFetch Policy considered. Since current *OS*s does not explicitly take into account *TCA* when assigning threads to cores in Multithreaded Multicore Processors, the probability distribution shown in Figure 4.6 reflects a possible scenario in state-of-the-art processors. Notice in Figure 4.6 that the probability of randomly obtaining the *BEST TCA* (loss lower than 1%) is in average close to 10%. The remainder *TCAs* highly depend on the *IFetch Policy* and the specific *characteristics of each workload*. Thus, while randomly selecting a *TCA* for workload *8W2* incurs in more than 5% of throughput loss with a probability of 71%, using *RR*, this probability drops to 20% using a better *IFetch Policy*, like *FLUSH*. However, the same claim may not be stated for workload *8W3*, where this probability is close to 50% and 75% for *RR* and *FLUSH IFetch Policies*, respectively. Obviously, the specific characteristics of workload *8W3* turn it into a more difficult target for *FLUSH* policy, yielding worse results and raising the probability of obtaining a high throughput loss. Consequently, *it is important to have a mechanism that assures some amount of reliability*, in terms of *TCA* selection for each case.

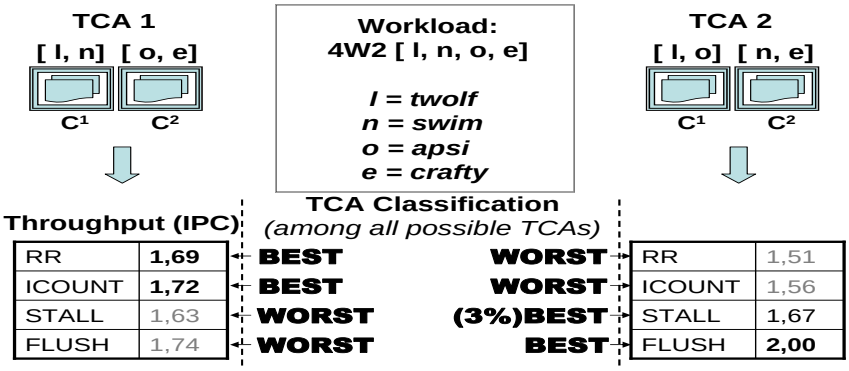


Figure 4.5: Example with different TCAs for a 4-thread workload.

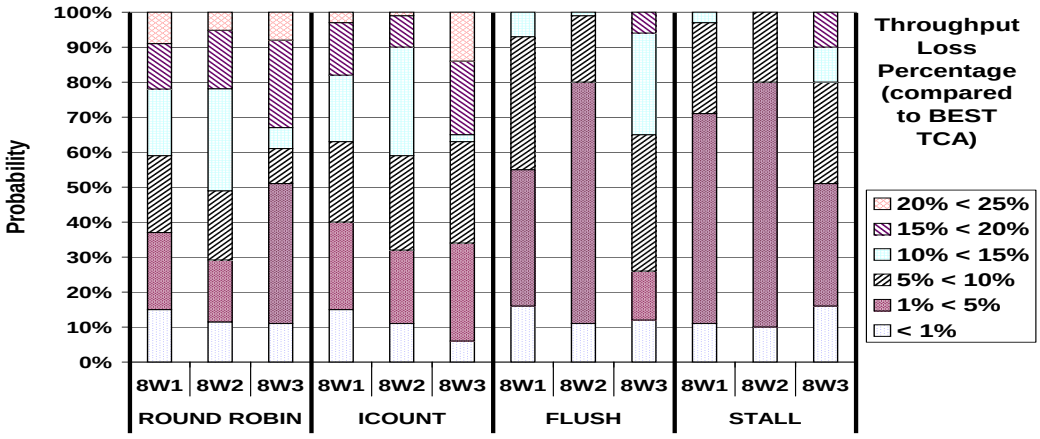


Figure 4.6: Probability of Throughput Loss in 8-thread workloads using Random TCAs.

4.5 Thread to Core Assignment Algorithm

In the following subsections we describe in detail the proposed *TCA Algorithm*, so as the *TCA Calibration* mechanism, aimed at handle the TCA scheduling layer in CMP+SMT processors. A complete evaluation is also included, with results revealing up to 21% of improvement over current state-of-the-art scheduling.

4.5.1 TCA Algorithm Foundations

In order to properly manage the *TCA* in a *CMP* comprised of *SMT* cores we take into account both the *workloads characteristics* and the underlying *SMT IFetch Policy*. The proposed *TCA Algorithm* is designed for homogeneous implementations, with the same *IFetch Policy* implemented in each 2-hardware-context core, like the *IBM POWER5* [60] and *POWER6* [39]. We focus on the *memory behavior* and the *ILP* of each application and how the *IFetch Policy* reacts to these characteristics. Regarding the *memory behavior* we can distingish two types of *IFetch Policies*: (1) *naive*, that perform badly with *memory-bounded (MEM)* applications like *RR* or *ICOUNT*, and (2) *robust*, with good response to *MEM* applications like *STALL* or *FLUSH*. Regarding the *ILP* we must observe how well the *IFetch Policy* boosts a *high-ILP* application performance without critically affect a *low-ILP* application running on the same core. As an indicator of both characteristics we use the *IPC* obtained by each application during a prior and representative portion of execution. The obtention of these *IPCs* is explained in detail in Section 4.5.3. The reason for chosing the IPC as a simple indicator of the applications characteristics is twofold.

On the one hand, the *IPC* of the threads is usually directly proportional to their *memory behavior*. High *IPC* results generally indicate good *memory behaviors*, and vice versa. *MEM* applications can monopolize the available resources in the execution core whether the *IFetch Policy* does not prevent it, as it is the case in *naive IFetch Policies*. Thus, a *MEM* thread wastes some of its assigned resources while waiting for memory, preventing an *ILP* thread, co-assigned in the same execution core, from doing forward progress. The *robust IFetch Policies* solve this problem by stalling (and even flushing) the *MEM* thread whenever it waits for memory. Consequently, in case of the *naive IFetch Policies* it is better to assign threads with similar memory behavior to a single core, and the opposite for robust policies.

On the other hand, the *IPC* of the threads is directly proportional to their *ILP*. High *IPC* results generally indicate high *ILP*. A thread with a high *ILP* tend to eagerly consume all available resources, such as functional units, to make forward progress, since the available paralellism allows to keep all available resources busy. Therefore, scheduling together two high *ILP* applications in a single core increases the resource contention, yielding a reduction in the overall throughput. Assigning these applications to different cores and scheduling them with lower *ILP* applications helps solving this resource contention and improves system throughput. Therefore, it is better to assign together threads with different *IPC* levels, that is high and low *ILP* threads.

Robust policies must detect when a thread is going to wait for memory in order to perform properly. Reacting too early or too late may negatively affect the final through-

Algorithm 4.5.1: TCA(IPC)

1- Arrange threads by IPC.

2- Split sorted thread list into two halves, creating two different lists. We call HIGH-list *to the sublist obtained from the upper half of the original list, with the higher IPC values. We call* LOW-list *to the other sublist, with the lowest IPC values.*

3- For i=0 to $\frac{\textit{Number of Threads}}{8}$ *do*

{
3.1- Assign the last two threads on the LOW-list *to one empty core.*
3.2- Assign the threads on the top and the tail of the HIGH-list *to one empty core.*
3.3- Remove the assigned threads from the lists.

4- While (Not Empty HIGH-list *and* LOW-list*) do*

{
4.1- Assign the thread on the HIGH-list *top and* LOW-list *tail to one empty core.*
4.2- Remove assigned threads from the lists.

Figure 4.7: TCA Algorithm implementation for FLUSH/STALL policy.

put [70]. Applications with a high rate of memory misses may impose a severe obstacle to a co-scheduled high performing application even in the presence of a robust policy. For example, there is a 29% of performance degradation when co-scheduling *eon* with *equake* (bad memory behavior) as compared to *vortex*; using a FLUSH policy in an SMT core like the one described in Section 4.2. Hence, these *IFetch Policies* may be assisted isolating the threads with the worst *memory behavior* and scheduling them together with the less sensitive thread, that is the following with the worst memory behavior. The number of isolated *MEM* threads depends on the workload size. The more threads in the workload the more possible *MEM* threads present in the workload.

4.5.2 TCA Algorithm

The proposed *TCA Algorithm* manages the *Thread to Core Assignment (TCA)* intermediate layer in the *OS* scheduling process in *Multithreaded Multicore Processors*. Its implementation for *robust IFetch Polices* (i.e., *FLUSH* and *STALL*) is presented in Figure 4.7. The *TCA Algorithm* foundations explained in the prior section, can be easily identified in the implementation shown in Figure 4.7.

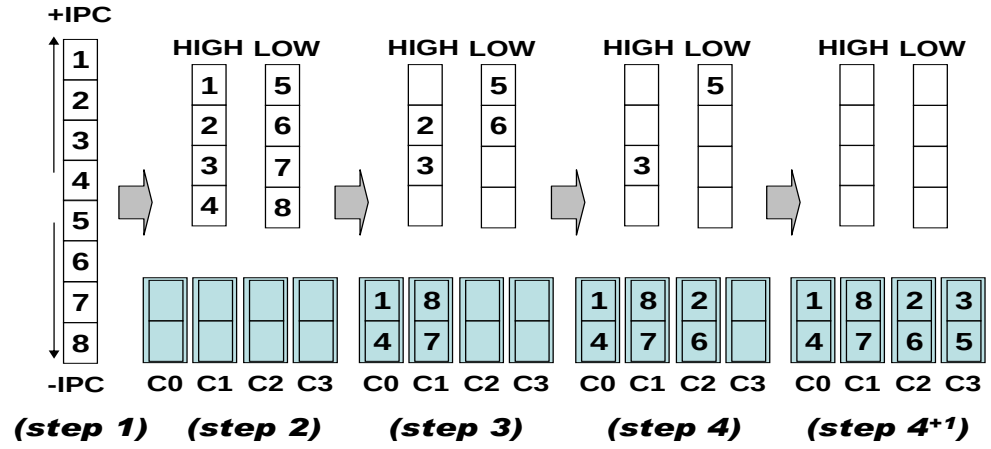


Figure 4.8: TCA Algorithm Example for FLUSH/STALL implementation (8 Threads).

First, in *steps 1* and *2*, the workload applications are classified according to their *memory behavior* and *ILP*. Applications with *good memory behavior* lie in the *HIGH-list* while the remainder applications lie in the *LOW-list*. Within each sublist the applications are arranged according to their *IPC*. Within each sublist, since *ILP* is directly proportional to the *IPC*, applications with more ILP lie in the head of the list.

In the *third step* the threads with the worst *memory behavior* are isolated in order to assist the underlying *IFetch Policy*. That is, these *memory-bounded applications* are assigned together to the same core. In order to balance both *HIGH* and *LOW* thread lists, a pair of *HIGH* threads are also scheduled together for each pair of *LOW* applications isolated. According to *ILP* reasoning above, we choose those threads with the most different *IPC* levels among all threads in the *HIGH-list*. In the particular case of a 2-core *CMP* (4-thread workloads) this step is skipped. Otherwise we would avoid *STALL* and *FLUSH* policies from doing any work, since no mixed pairs of *CPU* and *MEM* applications would be generated. For workloads with *8 or more threads*, this *third step* is repeated according to the *workload size* and *number of cores*, as the probable number of harmful *MEM* threads increases with the *workload size*.

In the *fourth step* the remainder threads are assigned according to both *memory behavior* and *ILP* guidelines. That is, the thread with the *highest IPC* (*HIGH-list*) is paired with the thread with the *lowest IPC* (*LOW-list*). Figure 4.8 shows an example in a 4-core *TCA Algorithm* implementation. The resulting assignment procedure is both *simple* and *scalable*; with an asynthotic $O(NlogN)$ complexity, with N being the application count.

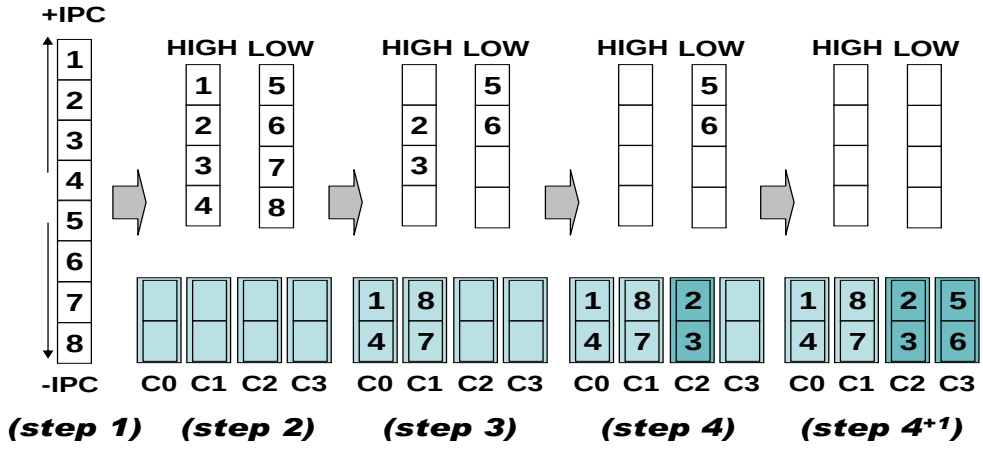


Figure 4.9: TCA Algorithm Example for RR/ICOUNT implementation (8 Threads).

The *main difference* between the *TCA Algorithm* implementation for *FLUSH/STALL (robust)* policies, shown in Figure 4.7, and the corresponding implementation for *RR/ICOUNT (naive)* policies lies in the *fourth step*. While in the *robust* implementation threads with different *memory behaviors* (i.e., from different sublists) are assigned to the same core, *in the* naive *implementation it is done just the oppossite*, according to the *TCA Algorithm's foundations*. That is, the threads assigned to any core come from the same thread-list (i.e., both threads from HIGH-list or both from LOW-list). The example shown in Figure 4.9 illustrates this difference in the *fourth step*. Notice that the difference comes directly from the different response of the underlying IFetch Policy involved in each case. Consequently, future *TCA Algorithm* implementations, involving different IFetch Policies, would require an analysis of the specific characteristics of the corresponding policy, in order to match them with the heterogeneity exhibited by the applications.

There is also another *difference* between the *robust* and *naive TCA Algorithm* implementations, related to the number of co-assigned *MEM* pairs (*step 3*) from the bottom of the *LOW-list*. While in the *robust* implementation this step is repeated according to the number of cores, in the implementation for a *naive* policy only one of these pairs is assigned to the same core. This difference comes from the *bad response of the* RR/ICOUNT *policies to these type of applications*.

Notice that the *TCA Algorithm* does not hamper the execution of any running thread; it does not stop threads but determines which threads should be assigned together to the same *SMT* core. This assignment is done according to the applications' *characteristics* and the underlying *IFetch Policy* implemented in each *SMT* core.

4.5.3 TCA Calibration

The *TCA Algorithm* requires, for each application, an *IPC prediction* during the following *OS* quantum of execution. These *IPC predictions* may come from any prediction mechanism as long as they were representative. However, as the execution flows applications go through different *program phases* [57, 23]. The behavior of an application may significantly change from one *program phase* to another. Therefore, whatever the mechanism employed to supply the *TCA Algorithm* with the requested *IPC predictions* it must be periodically reevaluated, or at least take in care the behavior variability of each application over time.

To assist the *TCA Algorithm*, we have developed an *IPC prediction* mechanism: the *TCA Calibration*. On every context switch, once the *OS* passes the workload to the hardware, an initial *TCA Calibration Phase* is triggered. As shown in Figure 4.10, the *TCA Calibration* simply consists of executing each application in *single thread (ST) mode* for a short amount of time. Since the processor is comprised of 2-hardware-context cores, two evaluation intervals (ST_0 and ST_1) are required, in the worst case, to fully test the whole workload. Although the *IPC predictions* obtained might not be fully accurate they are valid for the *TCA Algorithm* as long as the relative order between applications would be representative. That is, we are not interested in a *sophisticated IPC prediction* mechanism, that yields accurate predictions, but in a *simple* mechanism able to give accurate *relative values*. As long as the relative order is kept accurate the *TCA Algorithm* results would be good.

Using the *ST mode* during an interval of the execution, each time it is required reevaluating the *IPC* values for each application, involves a performance degradation. Obviously, the shorter these intervals the lesser the negative effects. After several experiments, in which we covered different portions of each application execution with an interval length ranging from a few thousands to tens of millions simulation cycles, we adjusted the size of these intervals to *10 millions of cycles*[7]. Adding these single-thread intervals (ST_0 and ST_1) to the *TCA Algorithm*'s overhead itself (denoted as t_a in Figure 4.10) the maximum overhead is 15+t_a%. Due to the *simplicity* of the *TCA Algorithm* the contribution of t_a to the final overhead may be considered as *negligible*.

Notice that the additional cost involved by using the *ST mode* for the *TCA Calibration* is only required when no *IPC* values are available for a new application. Whenever an application has a priorly calculated *IPC* value it may be directly fed to the *TCA Algorithm* to use it, without involving additional overhead. Obviously, as each application is exe-

[7]The research in *auto-adjustable low-overhead* intervals, to minimize the negative impact of the *TCA Calibration* on the system throughput, is left for future research.

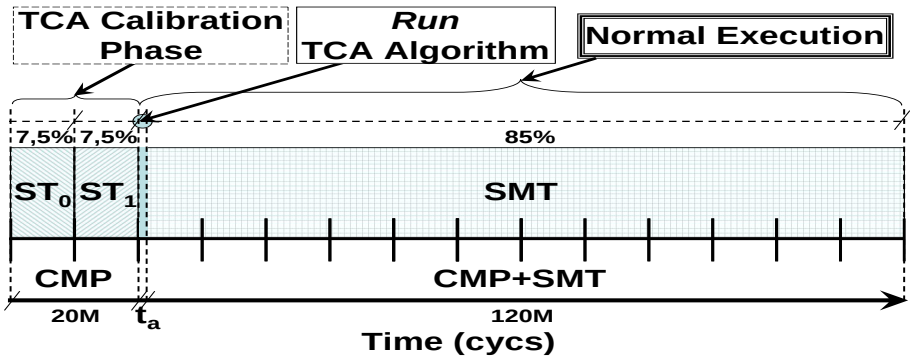


Figure 4.10: TCA Calibration.

cuted it goes through different *program phases*, with very different behavior. To reflect it, we established a *fixed maximum age* for each *IPC* value; after many experiments we settled on *140 millions of cycles*. Whenever an application's priorly calculated *IPC* value is more than *140 millions of cycles* old, it must be recalculated. As a consecuence, since we focus on *CMP* implementations with 2 *SMT* hardware contexts per core, three possible scenarios may arise in *TCA Calibration*:

1. All *or* more than half *the applications need a new* IPC *value*. This is the worst scenario, as shown in Figure 4.10, in which two consecutive *ST-mode* intervals are required for the *TCA Calibration*. Recall that a new *IPC* value is required whenever no prior *IPC* value is available for that application or the *IPC* value available is more than 140 millions of cycles old.

2. *No more than half the applications need a new* IPC *value*. In this case, just one *ST-mode* interval is required during the *TCA Calibration* (i.e., only ST_0 in Figure 4.10 is required).

3. *No application needs a new* IPC *value*. This situation may typically arise whenever quick *OS* context switches occur, as happens in case of exceptions arised in any of the applications. In this case, just the application that experiences the exception is typically removed from the execution workload, being replaced with another ready task. Eventually, no additional overhead is involved since no *TCA Calibration* is needed (i.e., neither ST_1 nor ST_0 in Figure 4.10 are required).

Real *OS quantums of execution* have a highly variable duration. So, in an state-of-the-art general-purpose *OS* like *Linux 2.6* [14], the lenght of the *OS* quantums of execution ranges *from 0 to 3200 millions of cycles*, with typical values lying on the *40M to 500M* interval. We have simulated the *TCA Calibration* performance, using the simulator's monitoring parameters and structures. However, in a real implementation it would be used the processor's *performance counters*, or any other specific monitoring hardware available. An *storage structure* would be used to keep, and read from, each application's prior predictions. In case of *long* OS quantums *(i.e., more than 140M cycles long) the execution may be momentarily interrupted by the hardware for an intermediate* TCA Calibration, possibly requiring an intermediate new *TCA*, depending on the behavior variations of the workload's applications. That is, both a *TCA Calibration Phase* and a *TCA Algorithm* triggering are required after every consecutive 140M of execution cycles.

4.5.4 TCA Algorithm Evaluation

In order to evaluate the performance of the proposed *TCA Algorithm* and *TCA Calibration* mechanisms we applied them to all 4-thread (*4W*) and 8-thread (*8W*) workloads in Table 4.1, simulated in 2 and 4-core *CMP+SMTs* respectively. The average results are shown in Figure 4.11. The *BEST TCA* results shown on the left side of Figure 4.11((a) and (b)) are obtained by simulating all different *TCAs* (i.e., 3 and 105 for 2 and 4-core implementations respectively) for each *workload* and *IFetch Policy*, selecting the ones which yield the highest throughput.

As we mentioned in the prior section, there are three different scenarios regarding *TCA Calibration* overhead; that is the reason to show two groups of results using the *TCA Algorithm* in Figure 4.11. The results shown on the middle of Figure 4.11((a) and (b)) are obtained supplying the *TCA Algorithm* with the *IPC values* obtained from a prior off-line single-threaded 300M-cycle execution of each application on the same execution core; that is, without requiring from any *IPC prediction* mechanism (i.e., the third scenario shown in prior section). The results shown on the right side of Figure 4.11((a) and (b)) involve two consecutive *ST-mode* intervals in the *TCA Calibration*. That is, the worst case (i.e., first scenario in prior section) in terms of *TCA Calibration* overhead.

Figure 4.11 shows that the *TCA Algorithm* yields *results very close to the optimal* for each case, a *3%* in average. Since the *TCA* is not explicitly taken into account by current *OS* for *CMP+SMT* processors, the state-of-the-art *TCA* policy would be represented by a *RANDOM TCA*. Due to the *probabilistic distribution* of the results, shown in Figure 4.6, directly comparing *TCA Algorithm*'s results with a *RANDOM TCA* may be misleading. However, from the results in Figure 4.6 it can be inferred that the probability for a *RAN-*

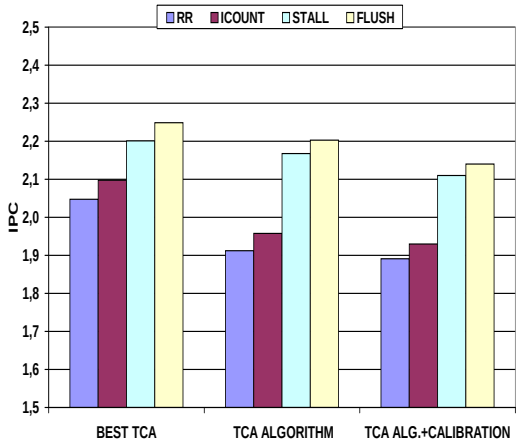
DOM TCA to achieve similar results to the ones yielded by the *TCA Algorithm* (i.e., 3% close to the BEST TCA) are only of *26%* [8], using the *RR* policy. Using a better IFetch Policy, like *FLUSH*, this probability is increased to *33%*. As a matter of example, the *TCA Algorithm* yields a speedup of *21%* in *8W1* using the *RR* policy, as compared to the *WORST TCA*.

Using the *TCA Calibration* mechanism slightly reduces the speedup yielded by the *TCA Algorithm*. As shown on the right side of Figure 4.11, the single-threaded portion of the execution, required by the *TCA Calibration* mechanism, slightly reduces the final throughput. The results in this case are in average *5% close to the optimal* for each case. In this case, the probabilities for a *RANDOM TCA* to achieve similar results to that obtained using the *TCA Algorithm* raise to *41%* [9], using the *RR* policy. Using *FLUSH* this probability is increased to *58%*.

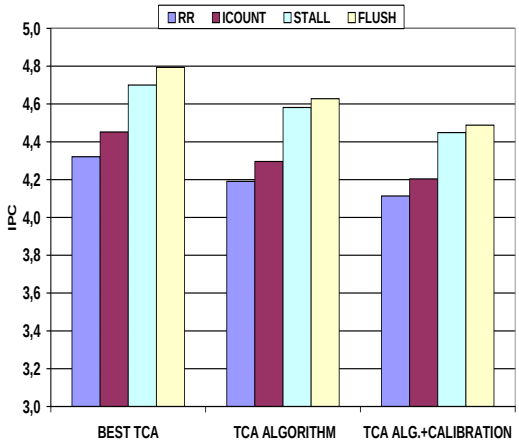
As mentioned in Section 4.5.3, not all context switches would require from the *TCA Calibration Phase*. Thus, only threads that have executed for more than 140M cycles since the last *calibration* would require from a *TCA Calibration*. This fact would reduce the overall use of the single-thread mode, and therefore reduce the final throughput reduction. Nevertheless, considering the minimal overhead involved, the TCA Algorithm supported by the TCA Calibration mechanism offers a quite interesting complexity-effective improvement.

In order to evaluate the *scalability* of the proposed *TCA Algorithm* in forthcoming microprocessor generations, we simulated all the 16 and 32-thread workloads in Table 4.1, simulated in 8 and 16-core *CMP+SMT* implementations, respectively. Due to exponential computational costs, we do not directly compare the *TCA Algorithm* results for 16 and 32-thread workloads with the *BEST TCA*, as done for 4 and 8-thread workloads. Instead, we randomly selected a group of *100* TCAs for each *workload* and *IFetch Policy*. From them, we selected the TCA which yields the highest throughput and called it *BEST of 100*. As done in Figure 4.11, Figure 4.12 shows the average results obtained using the *TCA Algorithm*. The results on the middle group of Figure 4.12 shows figures very close to the optimal for each case, a 3% in average. Therefore, from Figure 4.12 it can be inferred that the *TCA Algorithm* scales to future 8 and 16-core implementations. As shown on the right side of Figure 4.12, the effect of the *TCA Calibration* mechanism on the *TCA Algorithm*'s results is similar to that of 2 and 4-core *CMP+SMT* implementations.

[8] This is the average probability of a 3% throughput loss using random TCA with RR policy.
[9] This is the average probability of a 5% throughput loss using random TCA with RR policy.

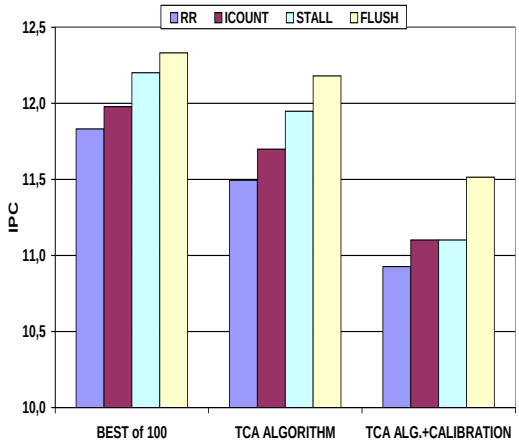


(a) 4-thread workloads.

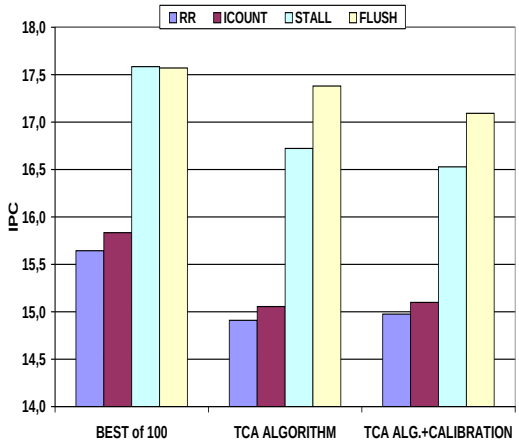


(b) 8-thread workloads.

Figure 4.11: TCA Algorithm results.



(a) 16-thread workloads.



(b) 32-thread workloads.

Figure 4.12: TCA Algorithm results.

4.6 Related Work

In [33] Jain et al. it is explored for the first time the soft realtime scheduling on an *SMT* processor, focusing on the *coschedule selection* (see Section 4.3). They propose new coscheduling variations that consider *resource sharing* and try to utilize *SMT* more effectively by exploiting application symbiosis. In this work we extend this exploration to a new scenario: *CMP+SMT*. In this scenario, we identify the need of a new step in the *scheduling process*: the *TCA*. Similarly to what happens with the *co-scheduling selection* in *SMT* processors, the *TCA* is directly related with the next step of the *scheduling process*, the *resource sharing*.

In [63] and [82] several schedulers and heuristics are proposed to manage the *co-schedule selection* and increase system throughput in *SMT* processors. We focus on the next step of the *scheduling process* for *CMP+SMT* processors. Once the *OS* has selected the workload to be executed in the next *OS* quantum each application in the workload must be assigned to one of the *SMT* cores. The goodness of this assignment determines the final system throughput. These proposals might work in conjuction with the *TCA Algorithm*, selecting easy-to-schedule applications for the *TCA Algorithm* in the underlying system. Nevertheless, more reseach is required to analyze the relation between the *co-schedule selection* and *TCA* scheduling layers (left for future work).

Shin et al. propose an *Adaptative Dynamic Thread Scheduling (ADTS)* [58] to manage the *resource sharing* (see Section 4.3) in *SMT* processors. The *ADTS* improves the system throughput in *SMT* processors by adapting the *IFetch Policy* to the *workload characteristics*. In this work we focus on the prior step of the *scheduling process* for *CMP+SMT* processors: the *TCA*. We do believe that both *ADTS* and the proposed *TCA Algorithm* may benefit each other (left for future work).

Kumar et al. propose in [38] some assignment policies to increase system throughput in *Single-ISA Heterogeneous Multicore* processors. They focus on obtaining the best match between *single-thread heterogeneous cores* and *applications*. Since in these processors each single-thread core has a different amount of resources, the way in which each application in the workload is assigned to one of the constituent cores determines the system throughput. This assignment is typically obtained after an initial sample phase to determine the best application-core match. We focus on a different scenario (i.e., *homogeneous CMP+SMT*) and the assignment is focused on obtaining the best match between *co-scheduled applications* in each *SMT* core. In our case the assignment only requires a representative *IPC* value for each application in the workload. We propose the *TCA Calibration* mechanism to assist the *TCA Algorithm*, providing it with these values with minimal execution overhead.

4.7 Chapter Summary

The *OS scheduling process* in the emerging *Multithreaded Multicore (CMP+SMT) Processors* differs from prior *SMT* and *CMP* processors', requiring a new scheduling layer, that we call *Thread to Core Assignment (TCA)*. In this chapter we have shown the importance of this new scheduling step in the system throughput. On the one hand, *we show that a* good TCA *may yield up to 28% system throughput improvement*. This chapter also analyzes the relation between the *TCA* and the *resource sharing*, generally managed by the *IFetch Policy* implemented in hardware. On the other hand, *we also show that a* bad TCA *can negate the performance advantange of a* good IFetch Policy. As a consequence, better results can be obtained using a *CMP+SMT* implementing *RR* policy, and the appropriate *TCA*, than that of implementing a better *IFetch Policy* like *FLUSH*.

The *TCA* which yields the best results depends on both the underlying *IFetch Policy* and the specific *workload characteristics*. Consequently, *there is not a single* TCA *which yield the best results for all cases*. Moreover, due to the *TCA* result distribution, it gets harder to obtain the optimal *TCA* as the *workload size* increases, since the number of different *TCAs* exponentially grows. According to the current trend, this problem is going to get harder as the amount of replicated cores on the chip increases.

In order to manage the *TCA*, we propose the third contribution of this thesis: the *TCA Algorithm*. This is the first *TCA* policy proposal in the literature. It generates close-to-the-optimal *TCAs* for each case, considering both the *workloads characteristics* and the underlying *IFetch Policy* implemented in the hardware. To do so, the *TCA Algorithm* just requires a representative *IPC* value for each application in the workload. To assist the *TCA Algorithm* with these *IPC values* we also propose an *IPC prediction mechanism*, that we call *TCA Calibration*. Our results show that the proposed *TCA Algorithm* obtains thread-to-core assignments 3% close to the optimal assignation for each case, yielding system throughput improvements up to 21%. Besides, its accuracy *scales* with the *workload size* and *number of on-chip SMT* cores.

Finally, we want to emphasize *simplicity* of the proposed *TCA Algorithm*, a key aspect considered during its development. We do think the proposed *TCA Algorithm*'s design is simple enough to allow a real implementation. Thus, each vendor would develop the corresponding *TCA Algorithm* implementation for each new processor and distribute it with its product, as currently done with the drivers. The *TCA* module could be then added to the *OS*, just requiring an additional Kernel recompilation or dynamic linkage. State-of-the-art processors like the *IBM POWER5* may benefit from the direct application of this contribution.

Chapter 5

Heterogeneity-Aware Multithreaded Multicore Processors

After confirming the benefits of applying the *Heterogeneity-Awareness* concept on the *Multithreaded Multicore Processors*, in this chapter we go further and foresee future *Heterogeneity-Aware Multithreaded Multicore Architectures*.

In the prior chapter we directly applied the *Heterogeneity Aware* concept on state-of-the-art processors, like the *IBM POWER5*, appropriately pairing running applications on an homogeneously distributed processor layout. We showed that the *Heterogeneity Aware* concept may be successfully applied despite of the hardware does not explicitly reflect the *Heterogeneity Aware* concept. In order to unleash the full potential of the *Heterogeneity Aware* concept we must turn the hardware itself *Heterogeneity Aware*.

In this chapter we envision the architecture of future generations of *Heterogeneity-Aware Processors*. In this sense, we propose the *heterogeneous Thread to Core Assignment (hTCA) Framework*, which provides OS-driven complexity-effective executions in the emerging *Multithreaded Multicore (CMP+SMT)* scenario. In *hTCA*, the *IFetch Policy* implemented within each *SMT* core is exposed to the *Operating System (OS)*. The *OS* is then in charge of deciding the best *IFetch Policy* for each *SMT* core according to both the *workload characteristics* and the *user needs*. The results included in the *hTCA* evaluation enclosed reveal an average *95% hTCA accuracy* when selecting the optimal choice to reduce the energy consumption without severely harming the *system throughput*. Our results also show reductions up to *71%* in the additional *energy* required by sophisticated *high-performance SMT IFetch Policies*, implemented within each *SMT* core in a *CMP+SMT processor*; compromising *less than 8%* of the *system throughput*.

5.1 Introduction

As analyzed in the prior chapter, the way in which the running threads are assigned to the constituent *SMT* cores in the *Multithreaded Multicore (CMP+SMT) Processors* processors heavily affects the final system throughput. Thus, an inappropriate *Thread to Core Assignment (TCA)* could negate the performance advantage of a full-fledged *IFetch Policy*. In order to select an appropriate *TCA*, by means of a *TCA Generator*, both the *workload characteristics* and the underlying *IFetch Policy* should be taken into account. Moreover, both the *TCA Generator* and the underlying *IFetch Policy* work in conjunction. Hence, a *naive IFetch Policy*, like the *Round Robin (RR)* policy [72], working in conjunction with a *good TCA Generator* may yield better system throughput results than a more *complex IFetch Policy*, like the *FLUSH* policy, working with a *bad TCA Generator*. In case of optimal *TCAs*, the differences in the system throughput results obtained with different *IFetch Policies* may be significantly reduced. Thus, the results included in the prior chapter points out that the system throughput difference between implementing the *RR* and the *FLUSH* policy may drop to an average 10%, regardless the workload size and number of *SMT* cores.

Among the state-of-the-art *CMP+SMT* processors we find the *IBM POWER5* [60] and *POWER6* [39], in which homogeneous *SMT* cores are replicated along the chip. Each constituent *SMT* core implements, in hardware, its own *Instruction Fetch (IFetch) Policy* [20, 24, 70, 72], which determines the thread(s) to fetch instructions from each cycle. Some proposed *IFetch Policies*, like the *FLUSH* [70] mechanism, explicitly handle load instructions that experience L2 Cache Misses. These instructions represent a severe challenge to be faced up in *SMT* execution cores, since they may block the execution; avoiding all running threads on the same *SMT* core from doing forward progress. However, explicitly handling these instructions generally comes at an additional energy consumption cost. Thus, in order to satisfy a high-throughput demand the *FLUSH* mechanism requires to refetch some amount of instructions, with the consequent additional energy consumption. This additional overhead is sometimes too much high to be paid in a real processor design, eventually implementing less aggressive Instruction Fetch Policies.

In this chapter we start unleashing the full potential of the *Heterogeneity-Aware* concept in *Multithreaded Multicore Processors*. For a processor to fully exploit the heterogeneity in the behavior of the running applications it must reflect this heterogeneity itself. Consequently, the hardware of a *true Heterogeneity-Aware* processor must dynamically adapt to the variations in the applications' behavior, aiming to devote the appropriate portion of the processor resources to each execution thread. Only by doing such dynamic resource alloation, that is being dynamically *Heterogeneity-Aware*, could be reach the

higher ratios of *complexity-effectiveness* in our executions. This must be kept in mind whether we are interested in executing as much instructions as possible involving the lowest energy consumption in the process; a goal of particular interest for future laptop and mobile oriented processor designs.

According to the aim of this chapter, we present a novel *OS*-driven framework aimed at providing *complexity-effective* executions in the emerging *CMP+SMT* processors: the *heterogeneous Thread to Core Assignment (hTCA)*. The *hTCA* is a hardware/software co-designed proposal that lean on the benefits of implementing a good *TCA Generator*, in tune with the user needs. Thus, the *hTCA* user may specify (by means of a user interface included in the OS) the desired *Quality-of-Service (QoS)* according to its needs. This *QoS* indicates, measured in a single percentage, the relative importance of both the *system throughput* and the *power consumption* in the system output. As a matter of example, if the user specifies a *QoS* of 50% the *hTCA* would reduce the system *power consumption* comprimising at the most 50% of the system *throughput*. The *hTCA*, according to the specified *QoS*, dynamically change the *Instruction Fetch (IFetch) Policy* implemented in each *SMT* core and alter the *TCA* produced by the *TCA Generator*, using an *hTCA Algorithm*.

Current commercial products, such as the *Intel SpeedStep Technology* [6] and the *AMD PowerNow!* [2], already provide *complexity-effective* executions. They both provide a user interface in the *OS* which allows reducing the processor working *frecuency* (and even *voltage*). Thus, when the same processor is run at a lower frequency, it generates *less heat* and consumes *less power*. This can conserve battery power in notebooks, extend processor life, and reduce noise generated by variable-speed fans. Unlike *Intel SpeedStep Technology* and the *AMD PowerNow!*, the *hTCA* works at an *architectural level* instead of a *physical level*. The *hTCA* gradually reduces the *architectural functionality* implemented in the processor without affecting the underlying *physical level*. As a consecuence, the *hTCA* may also work in conjunction with both *Intel SpeedStep Technology* and the *AMD PowerNow!*, each one affecting at a different level: (*architectural* or *physical*).

5.2 Methodology

Table 5.1 shows the main simulation parameters so as the workloads chosed. Since a complete study of all benchmarks is not feasible due to excessive simulation time we have randomly chosen some of them. The name of each workload is *xWy*, where *x* and *y* stands for the *number of threads* involved and the *workload identifier* respectively (e.g., *4W2* identifies the second workload with 4 threads). Each workload of size *x* is simulated on a *CMP+SMT* implementation with $\frac{x}{2}$ two-hardware-context *SMT* cores.

Simulation Parameters	
Pipeline depth	11 stages
Queues Entries	64 int, 64 fp, 64 ld/st
Execution Units	4 int, 3 fp, 2 ld/st
Physical Registers	320 regs.
ROB Size*	256 entries
Branch Predictor	perceptron (4K local, 256 perceps.)
BTB	256 entries, 4-way associative
RAS*	100 entries

Simulation Parameters	
L1 I-Cache	64KB, 4-way, 8 banks
L1 D-Cache	32KB, 4-way, 8 banks
L1 lat./miss	3/22 cycs.
I-TLB ,D-TLB	512 ent. Full-associative
TLB miss	300 cycs.
L2 Cache	4MB, 12-way, 4 banks
L2 latency	15 cycs.
Main Memory lat.	250 cycs.

	Number of Threads		
Name	2	4	8
xW1	b, j	b, q, t, j	d, l, b, g, i, j, c, f
xW2	n, e	l, n, p, e	b, g, m, n, a, h, o, p
xW3	d, a	d, s, r, a	m, n, r, q, i, j, e, h
xW4	g, f	g, b, m, f	l, b, g, m, n, r, f, s
xW5	r, p	r, j, f, p	q, b, c, k, e, a, o, t

gzip	a	eon	h	apsi	o	facerec	v
vpr	b	gap	i	wupwise	p	applu	w
gcc	c	vortex	j	equake	q	galgel	x
mcf	d	bzip2	k	lucas	r	ammp	y
crafty	e	twolf	l	mesa	s	mgrid	z
perlbmk	f	art	m	fma3d	t		
parser	g	swim	n	sixtrack	u		

Table 5.1: Simulation parameters and Workloads. (resources marked with * are replicated per thread)

We simulate each workload employing 2 different IFetch Policies: *ICOUNT* [72] and *FLUSH* [70]. Both *ICOUNT* and *FLUSH* are far from representing the state-of-the-art in *SMT IFetch Policies*, but constitute an *easy-to-explain example* of a possible *hTCA* implementation. The analysis and development of *hTCA* implementations involving state-of-the-art *IFetch Policies* [20, 25] are left for future work. All simulations are executed for a fixed interval of 120 millions of simulation cycles.

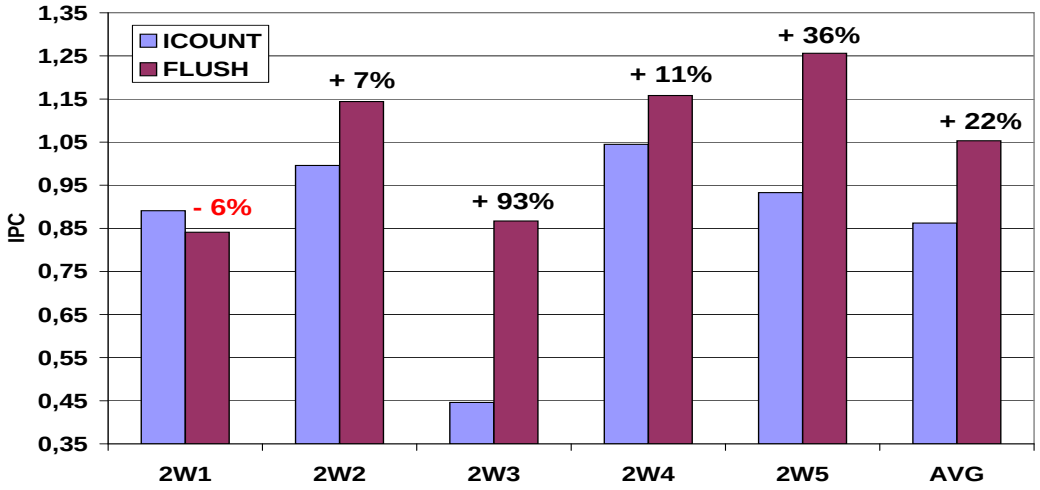
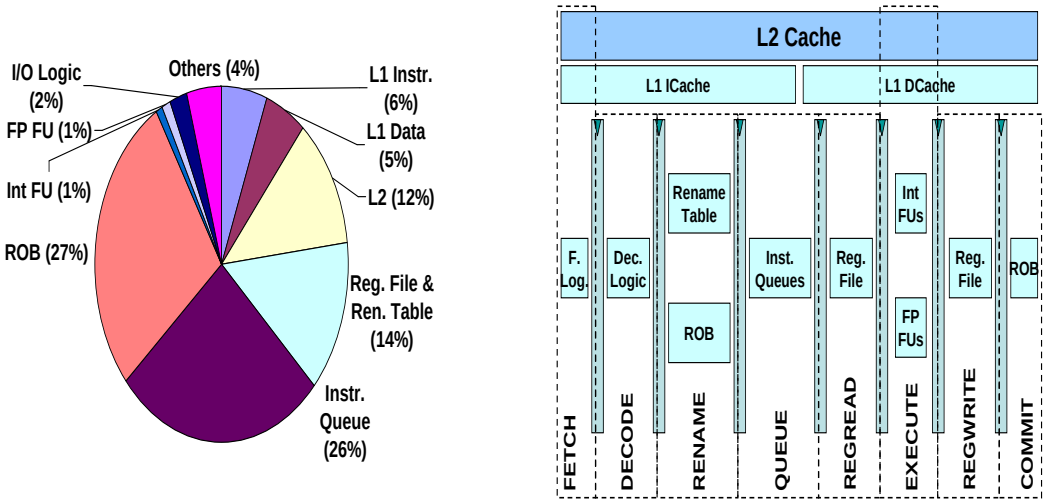


Figure 5.1: Throughput in single-core SMT.

5.3 IFetch Policy in SMT Processors

The *IFetch Policy* represents probably the most important issue in an *SMT* execution pipeline, determining from which thread(s) instructions are fetched every cycle. In order to avoid hardware resource monopolization by any of the running threads, the *IFetch Policy* should explicitly handle long-latency loads. An L2 Cache Miss may block hardware resources, and the whole *SMT* execution pipeline, thus avoiding forward progress by any other running thread. We call *robust/good* to those *IFetch Policies* that explicitly handle long-latency loads, *naive/bad* otherwise. The literature is plenty of *IFetch Policy* proposals [20, 24, 70, 72], some of them, like the *FLUSH* [70], falling into the *robust* category.

The *FLUSH* [70] mechanism avoids any running thread from monopolizing the available hardware resources. Built on top of the *ICOUNT* [72] policy, the *FLUSH* mechanism detects loads that experience L2 Cache Misses (unhandled by the ICOUNT policy) and reacts stalling the offending thread; preventing it from monopolizing more hardware resources. Moreover, the newest instructions (until the blocked load) of the offending thread are flushed away from the execution pipeline. So, by freeing the corresponding hardware resources they are available for the remainder running applications. As shown in Figure 5.1, the *FLUSH* mechanism yields average system throughput improvements of 22% in single-core *SMT* processors, with speedups of up to 93%.



(a) Distribution (% per resource). **(b) Pipeline Stages/Resources Distribution.**

Figure 5.2: Energy Consumption.

The *FLUSH* mechanism represents a *high-power-consumption* alternative, aimed at throughput-oriented scenarios, in which the *system throughput* is the main concern regardless of the power required. Flushing away instructions from the execution pipeline, and having to re-fetch them later on in the execution, implies an *additional energy cost*. This cost depends on the pipeline stage in which was the instruction by the flush time, as described in Section 5.3.1.

5.3.1 Instruction Energy Consumption in SMT Processors

Folegnani et al. analyzed in [26] the energy consumption for each hardware resource in a typical execution pipeline (See Figure 5.2(a)). Assuming that each instruction in a given execution pipeline requires 1 *energy unit*[1] to be committed, and given the resource usage for a typical *SMT* core shown in Figure 5.2, Table 5.2 shows the *Energy Consumption Factor*. Using the *Energy Consumption Factor*, and tracking the pipeline stage in which was each flushed instruction by the flush time, it may be easily estimated the additional energy cost involved by the *FLUSH* mechanism. Thus, Figure 5.3 shows the additional energy consumption employed by the FLUSH mechanism to obtain the system throughput improvements shown in Figure 5.1.

[1]The exact amount of energy depends on the specific microarchitecture characteristics.

	Energy Consumption Factor	
Pipeline stage	**Local**	**Accumulated**
Fetch	0.13	0.13
Decode	0.03	0.16
Rename	0.22	0.38
Queue	0.26	0.64
Reg. Read	0.05	0.69
Execute	0.13	0.82
Reg. Write	0.05	0.87
Commit	0.13	1

Table 5.2: Energy Consumption Factor.

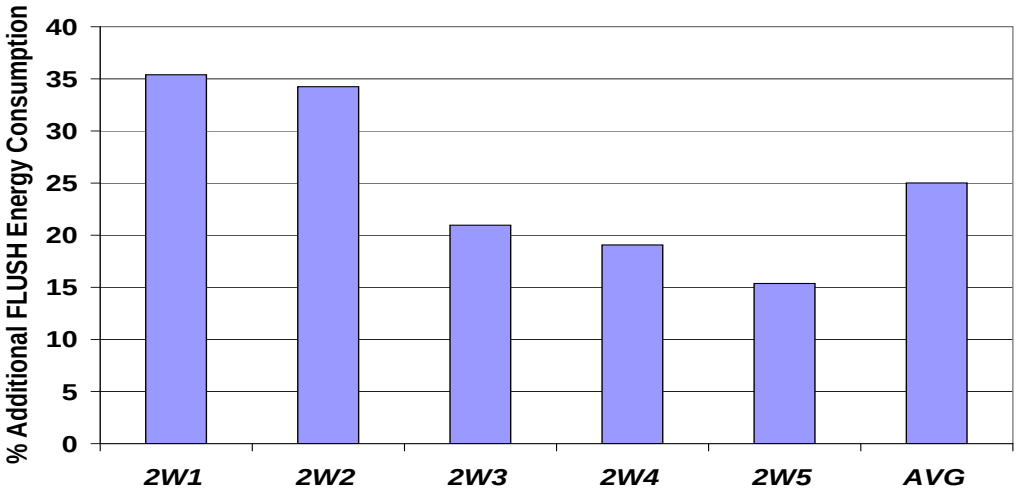


Figure 5.3: Additional Energy Consumption in single-core FLUSH SMT.

5.4 Thread to Core Assignment in SMT On-Chip Multiprocessors

As analyzed in the prior chapter, the *Thread to Core Assignment (TCA)* determines the performance of the underlying *IFetch Policy*, implemented in each *SMT* core, in the emerging *CMP+SMT* processors. By properly pairing to the same *SMT* core applications with *compatible characteristics*, according to each core's *IFetch Policy*, it is possible to smooth the performance differences between different *IFetch Policies*. As shown in Figure 5.4, a *good TCA* (e.i., *BEST TCA* results) reduces the performance differences from implementing *ICOUNT* to a more sophisticated *FLUSH SMT IFetch Policy*. Focusing on the 8-thread workloads this difference goes from a 20% to 7% when moving from *WORST* to *BEST TCA*. As a consecuence, whenever good *TCAs* are assured it is possible to reduce

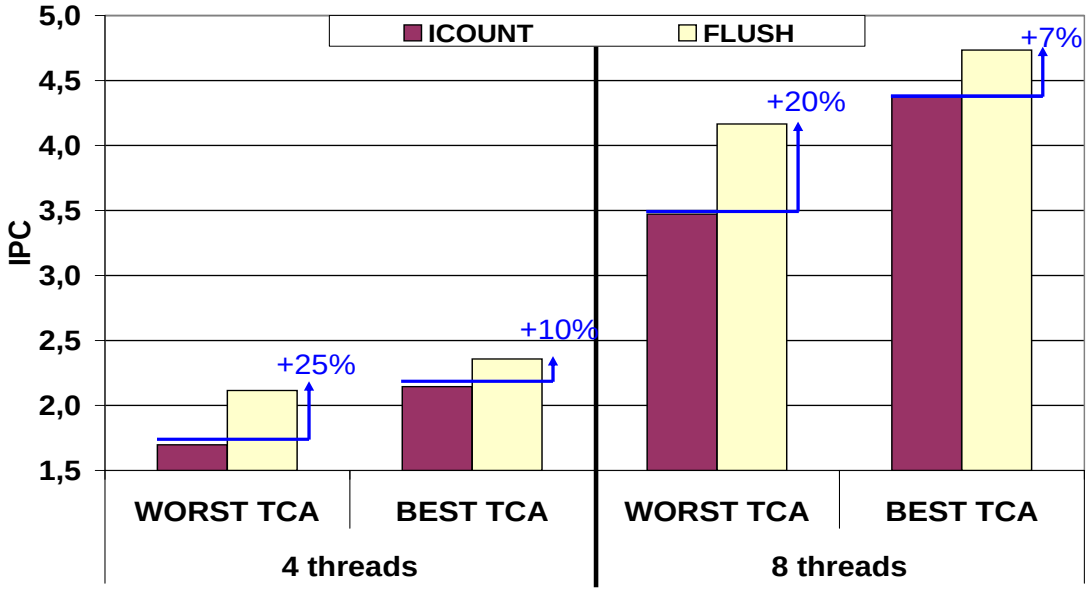


Figure 5.4: TCA Sensitivity for 4 and 8-core CMP+SMTs.

the complexity involved in the *global IFetch Policy*[2] of a *CMP+SMT* without severely compromising the *system throughput*. This *complexity reduction* implies a disminution in the processor *energy consumption*.

The *hTCA Framework* leans on the smoothed difference between IFetch Policies provided by a good *TCA*. Consequently, as part of this framework, the hTCA involves a *TCA Generator* to assure this fact. As we saw in the prior chapter, randomly chosing a *TCA* does not assure reliable results. Therefore, in order to assure *good TCAs* we employ in this chapter the *TCA Algorithm* proposed in the prior chapter (see Section 4.5) as *TCA Generator* for the *hTCA* Framework. Nevertheless, the *hTCA Framework*'s design does not consider any specific implementation for the *TCA Generator*, and could be replaced with alternative TCA Generator implementations as long as they would provide accurate *TCAs*.

[2]The term *global SMT IFetch Policy* refers to the composition of *SMT IFetch Policies* implemented in all the *SMT* cores in a *CMP+SMT* processor. Since each *SMT* core may implement a different *IFetch Policy*, we refer to the *IFetch Policy* of each *SMT* core as *local*.

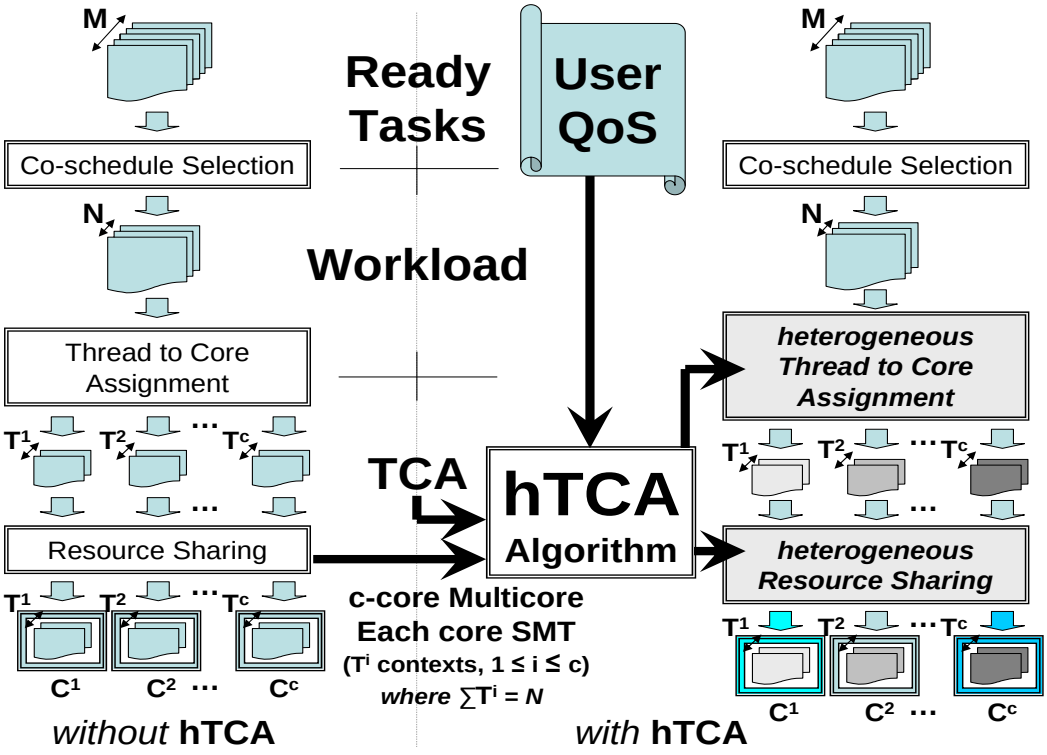


Figure 5.5: Scheduling Layers in CMP+SMTs with and without hTCA.

5.5 The hTCA framework

The *heterogeneous Thread to Core Assignment* (hTCA) Framework provides user-definable *Complexity-Effectiveness* in the emerging *CMP+SMT* processors. By defining a single percentage, that we call *Quality of Service (QoS) percentage*, it may be specified, using the *OS* user interface, the desired relation between the *system throughput* and the *energy consumption* in the *system output*. This relation is provided by the *hTCA* at an *architectural level*, altering both the *global IFetch Policy* and the *TCA* yielded by the *OS TCA Generator*. An *hTCA Algorithm*, implemented as part of the *OS scheduling process*, heterogeneously modifies the two lowest layers in the *OS scheduling process* for *CMP+SMT* processors, as shown in Figure 5.5. Thus, the *hTCA Algorithm*, according to the *QoS percentage* specified by the user, alters both the *TCA*(See Chapter 4) and the *Resource Sharing* (generally implemented by the *IFetch Policy*) [33].

In order to reduce the system's *energy consumption*, according to the user needs (*QoS*), the *hTCA Algorithm* heterogeneously change the *valid IFetch Policy* in each *SMT* core. Thus, an hTCA-like design for a *Multithreaded Multicore Processor* must implement more than one IFetch Policies on each *SMT* core. Each of the implemented policies must be *validated/invalidated* using a simple signal that must be exposed to the OS. By means of a proper blend of both *throughput-aggressive*, but *power-hungry*, and more moderated policies the hTCA Framework may provide the user the ability to chose the desired performance/consumption ratio, according to her needs.

As an example, if we had an *hTCA-processor* in our mobile phone and we were running out the battery, we could reduce this ratio to a 30% to continue using it (although with a some reduction in its performance) and increase the battery life expectance until reaching a recharge point. This sort of adaption to low-energy conditions is quite different from those that works at a *physical level*, reducing *frequency* or *voltage*. In the case of a physical variation (i.e., reducing the clock frequency of the processor) the granularity of the quality of service provided is much coarser. In fact, both *architectural (hTCA)* and *physical* actions are envisioned to work together, using the first for *fine-grain QoS* and the latter for a *coarse-grain QoS*.

To achieve the desired *energy consumption reduction* without severely compromising the *system throughput*, the *hTCA* also alters the *TCA* generated by the *OS TCA Generator*. As we saw in Chapter 4, the *TCA* depends on both the *workload* and *IFetch Policy* characteristics. Since the *hTCA Framework* alters the latter, it is obvious that a new *TCA* must be calculated should we want to maintain an optimal *TCA* in force.

In the following subsections we cover in depth all the *hTCA*'s specific details, both from a hardware and software perspective so as its evaluation for an illustrative implementation using ICOUNT/FLUSH policies.

5.5.1 Hardware/Software co-design

The *hTCA* Framework constitutes a hardware/software co-designed solution in which the *IFetch Policy*, implemented in hardware in each of the constituent *SMT* cores of a *CMP+SMT* processor, is exposed to the *OS*. According to the *QoS* specified by the user, the *OS* alters both the underlying *IFetch Policy* and the *TCA* to fulfill the user demands without severely harming the overall system throughput. This is done at an *architectural level*, altering the functionality provided by the architecture (i.e., going from a *better* to a *worst* IFetch Policy), instead of at a *physical level*, as done by some current solutions such as the *Intel SpeedStep Technology* [6] and the *AMD PowerNow!* [2].

The *system throughput reduction* is limited by the *QoS percentage*. Assuming two available *SMT IFetch Policies*, called *low* and *high-performance*[3], the *hTCA* establishes the number of *SMT* cores *Lx* that should use the *low-performance*[4] *SMT IFetch Policy* using Equation 5.1. Next, the *hTCA Algorithm* decides both the *Lx SMT* cores that will use the *low-performance IFetch Policy* and how should be modified the *TCA*, yielded by the *OS TCA Generator*, in order to maximize the *system throughput*. Finally, the *hTCA* Framework *activates/deactivates* the corresponding *IFetch Policy* in each *SMT* core and assigns threads accordingly. As a consecuence, the *global SMT IFetch Policy* is comprised of *Lx SMT* cores with *local low-performance policy*, while the remainder *SMT* cores use the *high-performance policy*, with the subsequent *energy consumption reduction*.

$$Lx = NumCores - \left\lfloor \frac{QoS \times NumCores}{100} \right\rfloor \tag{5.1}$$

$$with \quad \text{Lx}\in\text{N}, \; 0 \leq \text{Lx} \leq \text{NumCores} \;\; and \;\; 0 \leq \text{QoS} \leq 100$$

5.5.2 The hTCA Algorithm

In order to minimize the *system throughput degradation*, due to using a *low-performance SMT IFetch Policy* in some *SMT* cores, the *TCA* should be modified accordingly. In the *TCA step* of the *scheduling process*, the *OS* assumes that all *SMT* cores implements a *high-performance* policy, using the corresponding *TCA Generator*. Next, the *hTCA* determines the number of *SMT* cores implementing a *low-performance* policy (See Section 5.5.1) and the *hTCA Algorithm* decides which co-assigned threads should be allocated to the *low-performance SMT* cores.

The *hTCA Algorithm* is designed according to both the *SMT IFetch Policies* available in the hardware (implemented within each *SMT* core) and the *TCA Generator* implemented in the *OS*. This means that there is an specific *hTCA Algorithm* implementation for each combination of *SMT IFetch Policies* and *OS TCA Generator*. In this research we focus on an *hTCA* implementation for *ICOUNT*(*low-performance*) and *FLUSH*(*high-performance*) *Fetch Policies*, and the *TCA Algorithm* presented in Chapter 4 as *OS TCA Generator*. Figure 5.6 illustrates the application of the proposed *hTCA* Framework for a 2-core *CMP* processor with 2 hardware contexts per *SMT* core, implementing *ICOUNT/FLUSH IFetch Policy*, like the *IBM POWER5* [60] and *POWER6* [39].

[3]Research with multiple *IFetch Policies* per *SMT* cores is left for future research.

[4]The *high-performance* IFetch Policy is used by default.

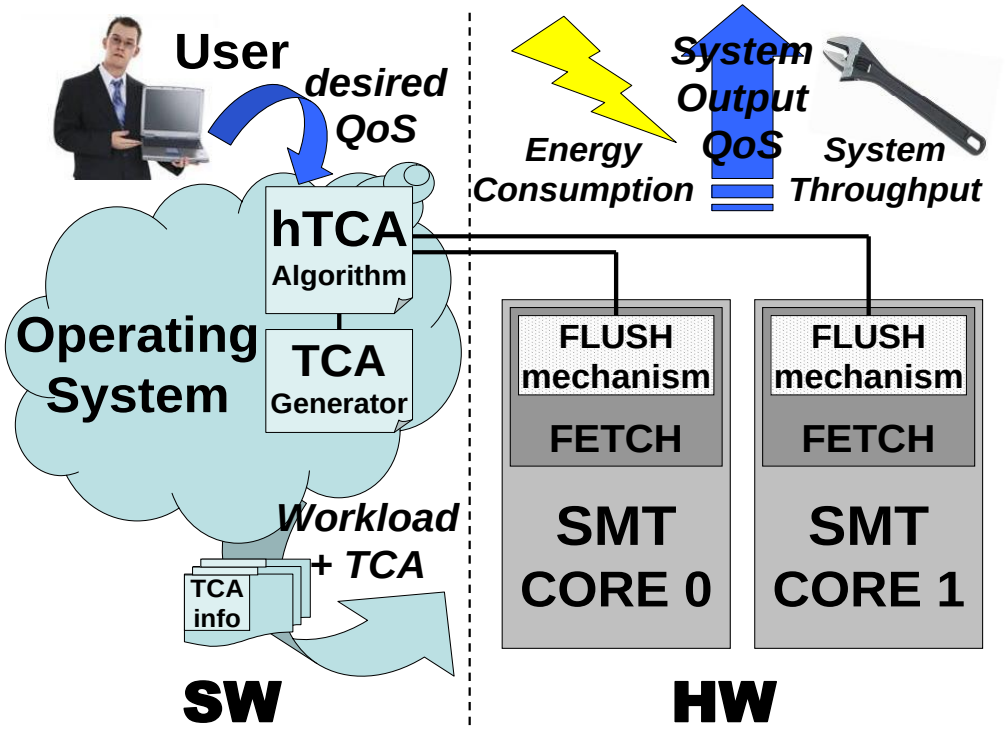


Figure 5.6: hTCA Framework Example for 2-core ICOUNT/FLUSH CMP+SMT.

In a real implementation, each processor's vendor would distribute the corresponding *hTCA Algorithm* implementation for each new processor implementing the *hTCA* Framework, as currently done with drivers. Then, an additional kernel recompilation (or dynamic module linkage) would be enough to update the *OS* with the corresponding *hTCA Algorithm*, in the case of *Linux*, or a driver installation, in the case of *Windows*.

The proposed *hTCA Algorithm* implementation is shown in Figure 5.7. This *hTCA Algorithm* is throughput-oriented and selects the *SMT* cores to *deactivate* (i.e., use the *low-performance IFetch Policy*) *minimizing* the corresponding *performance degradation*. In order to keep simple enough the *OS scheduling process*, the *hTCA Algorithm* reassigns to different *SMT* cores the minimal amount of applications. To do so, the *hTCA Algorithm*, shown in Figure 5.7, reassigns applications according to their characteristics, starting from the ones with the lowest impact on the overall system throughput.

Algorithm 5.5.1: HTCA()

```
1- Split the SMT cores into three core-lists, according to the memory behavior of the assigned applications:
     MEM-cores, ILP-cores and MIX-cores.
2- Arrange the three core-lists by accumulated IPC of assigned applications.
3- If ( Lx = NumCores ) then
{  3.1- Reevaluate the TCA Algorithm using ICOUNT.
4- Else
{  4.1- MIX-pairs = 0
   4.2- For i=0 to Lx do
   {  {  4.2.1- If (NOT-EMPTY(MEM-list) ) then
         {  4.2.1.1- Select the core in the tail of the MEM-list.
         }
         4.2.2- Else If (Not-Empty(ILP-list) ) then
         {  4.2.2.1- Select the core in the tail of the ILP-list.
         }
         4.2.3- Else If (Not-Empty(MIX-list) ) then
         {  4.2.3.1- Select the core in the top of the MIX-list.
            4.2.3.2- MIX-pairs + = 1
         }
      }
      4.2.4- Deactivate the selected core. ↪Low-performance core.
      4.2.5- Remove the selected core from the corresponding core-list.
      4.2.6- If (MIX-pairs = 2) then
      {  {  4.2.6.1- Reassign to the same SMT core the two applications with the highest-IPC values in the last
               two MIX-cores deactivated .
            4.2.6.2- Reassign to the same SMT core the two applications with the lowest-IPC values in the last
               two MIX-cores deactivated .
         }
         4.2.6.3- MIX-pairs = 0
      }
   }
}
```

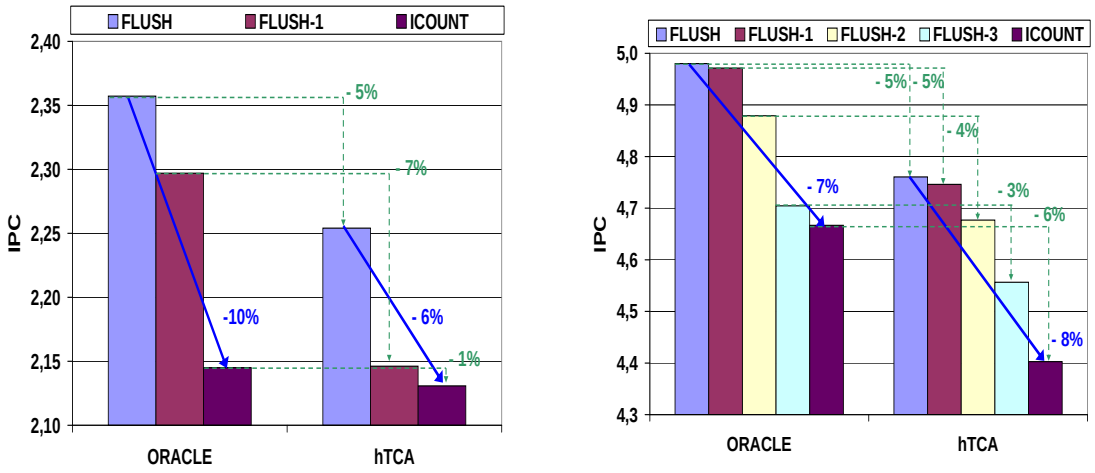
Figure 5.7: hTCA Algorithm implementation for ICOUNT/FLUSH policies.

5.5.3 hTCA evaluation

We first evaluated the *hTCA Framework*'s ability to select the best choice for each level of *complexity-effectiveness* demanded by the user, that is for each *QoS percentage*. Whenever the user defines an specific *complexity-effectiveness* level, using the *QoS percentage* provided by the *OS* interface, the *hTCA Framework* uses a *Core Selector* and a *TCA Generator*, as shown in Figure 5.6, to adapt the execution at an architectural level.

Figure 5.8[5] shows the average system throughput results obtained both using an *Oracle TCA Generator* aided by an *Oracle Low-Performance-Core Selector*(on the left), *ORACLE* from now on, and those yielded using the *TCA Algorithm* (see Section 4.5) as *TCA Generator* aided by the *hTCA Algorithm* as *Core Selector*.

[5]In this section, the term *FLUSH-Lx* stands for a *CMP+SMT* implementation in which all but *Lx* cores implement the *FLUSH* mechanism; *ICOUNT* otherwise.



(a) 4-thread workloads.

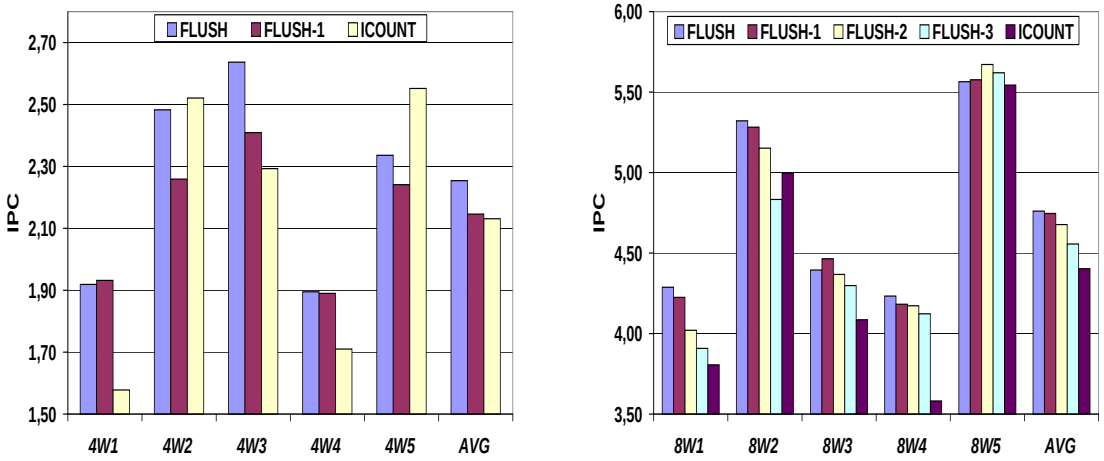
(b) 8-thread workloads.

Figure 5.8: Average System Throughput Comparison.

The *ORACLE* option, using *oracle predictors*[6], yields the *BEST TCA* for each case and selects the next core to deactivate so that the *system throughput degradation* is *minimized*. Obviously, the *ORACLE* option represents an *ideal scenario*. From Figure 5.8 it may be inferred that the *hTCA* Framework succeeds selecting both the *TCA* and *core deactivation sequence*, with an average 95% accuracy.

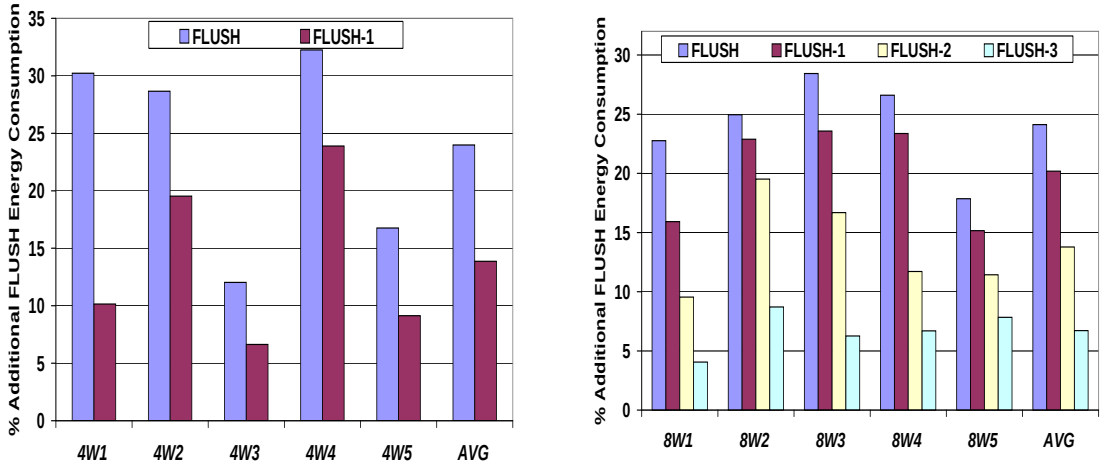
Next, it was evaluated the *hTCA Framework*'s ability to obtain *complexity-effective executions* in *CMP+SMT* processors. The users, by means of a *QoS percentage*, may select the balance of *power conservation* and *performance* that best suits them. Once translated the specified *QoS percentage* (See Section 5.5.1) into a number of cores to *deactivate* (*Lx*), the *hTCA* Framework employs the *hTCA* Algorithm to establish the *core deactivation sequence*, that is the cores that will use the *low-performance SMT IFetch Policy*. Figure 5.9 breaks down the *hTCA* results shown in Figure 5.8. Using the *Enery Consumption Factor* described in Section 5.3.1, Figure 5.10 breaks down the *hTCA Energy Consumption Reduction* obtained as it is augmented the number of *deactivated* cores. From Figures 5.9 and 5.10 it may be inferred that the *hTCA* Framework succeeds yielding *complexity-effective executions*. Thus, it provides reductions in the additional energy re-

[6]Simulated using brute force, that is, simulating all the different possibilities and chosing the ones with the highest values



(a) 4-thread workloads. **(b) 8-thread workloads.**

Figure 5.9: hTCA results.



(a) 4-thread workloads. **(b) 8-thread workloads.**

Figure 5.10: hTCA Energy Consumption Reduction.

quired by the *FLUSH* mechanism of 40% and 71%, compromising less than 5% and 8% of the *system throughput*, respectively for 4-thread and 8-thread workloads. These results also give evidences of the *hTCA* Framework *scalability* when passing from 2 to 4-core *CMP+SMT* implementations.

Finally, a deeper analysis of the results shown in Figure 5.9 reveals that, contrary to what would be expected, the best results are not always yielded by implementing the *high-performance IFetch Policy* in all the constituent *SMT* cores. Thus, both *4W1* and *8W3* experience *system throughput improvements* when deactivating one *SMT* core (e.i., all *SMT* cores implementing the *FLUSH* mechanism but one using *ICOUNT*). Furthermore, some workloads even experience *system throughput improvements* when deactivating two cores (*8W5*); and even when deactivating all the cores (*4W2*, *4W5*). The rationale behind this phenomenon is twofold.

On the one hand, the specific characteristics of the *SMT IFetch Policies* employed may yield, for some workloads, better results using the *low-performance IFetch Policy* than using the *high-performance* one. Notice that this already happens in single-core *SMT* processors, as is the case of *2W1* in Figure 5.1. On the other hand, the relation between the *TCA* and the *IFetch Policy* allows to obtain, for some workloads, *TCAs* using *low-performance IFetch Policies* that improve the results yielded by employing the *high-performance* ones. This phenomenon opens the path for future research on automatic detection of the optimal *hTCA execution mode*, which would yield the *highest system throughput* even *reducing* the *power consumption*.

5.6 Related Work

There are already complexity-effective frameworks implemented in current commercial processors. Thus, both the *Intel SpeedStep Technology (IST)* [6] and the *AMD PowerNow! Technology (APT)* [2] provide a significant reduction in both heat and power consumption, allowing the users to select the balance of power conservation and performance that best suits them. This can conserve battery power in notebooks, extend processor life, and reduce noise generated by variable-speed fans.

The *hTCA* Framework proposed in this chapter provides an additional control over the complexity-effectiveness of the executions in the emerging *CMP+SMT* processors. While both *IST* and *APT* reduce the microprocessor *frecuency* and *voltage*, affecting all running applications, the *hTCA* reduces the *architectural functionality* implemented, changing the valid *IFetch Policy* for a less *power-consuming* one in some of the constituent *SMT* cores. That is, while *IST* and *APT* work at a *physical level* the proposed *hTCA* Framework works at an *architectural level*. As a consecuence, both *IST* and *APT* might be used in conjunction with the *hTCA* Framework to increase the user control over the complexity-effectiveness in the processor, performing different granularities of complexity-effectiveness: *fine-grain*, in case of *hTCA*, and *coarse-grain*, in case of *IST* and *APT*.

Shin et al. propose in [58] an *Adaptative Dynamic Thread Scheduling* (ADTS) to manage the resource sharing in single-core *SMT* processors, adapting the underlying *IFetch Policy* to the workload characteristics. The *hTCA* Framework is designed for *multi-core SMT* processors (*CMP+SMT*) and strives to *reduce* the processor *energy consumption* without severely compromising the *system throughput*; according to the user needs. Both *ADTS* and *hTCA* may work in conjunction since they cover different scenarios; that is *single-core* and *multi-core* respectively.

Kumar et al. propose in [38] some assignment policies to *increase system throughput* in *Single-ISA Heterogeneous Multicore* processors, which focus on obtaining the best match between *single-thread heterogeneous cores* and *applications*. A *global energy consumption reduction* is provided by properly matching each application with the heterogeneous single-threaded core which best fits the application requirements. The *hTCA* Framework focus on a different scenario (e.i., *homogeneous CMP+SMT*) and its explicitly aimed at matching the *system energy consumption* with the *user needs* by *heterogeneously modifying* both the *TCA* and the *IFetch Policy* in *CMP+SMT* processors.

5.7 Chapter Summary

In this chapter we envision the architecture of future generations of *Heterogeneity-Aware Processors*. After analyzing in the prior chapter the benefits of directly applying the *Heterogeneity-Awareness* concept to current *Multithreaded Multicore Processors*, like *IBM POWER5* [60] and *POWER6* [39], in this chapter we start exploring the full potential of future *Heterogeneity-Aware Processors*.

For a processor to be fully *Heterogeneity-Aware* both its *hardware* and *software* (i.e., the applications running on it) must explicitly take into account the inherent heterogeneity in applications execution. To obtain complexity-effective executions, an Heterogeneity-Aware processor dynamically adapts the amount of processor resources devoted to each application so that it yielded *the highest throughput possible involving the lowest energy consumption*.

In this sense, we propose the *heterogeneous Thread to Core Assignment (hTCA) Framework*, which provides OS-driven complexity-effective executions in the emerging *Multithreaded Multicore (CMP+SMT)* scenario. In *hTCA*, the *IFetch Policy* implemented within each *SMT* core is exposed to the *Operating System (OS)*. The *OS* is then in charge of deciding the best *IFetch Policy* for each *SMT* core according to both the *workload characteristics* and the *user needs*. The results included in the *hTCA* evaluation enclosed reveal an average *95% hTCA accuracy* when selecting the optimal choice to reduce the energy

consumption without severely harming the *system throughput*. Our results also show reductions up to *71%* in the additional *energy* required by sophisticated *high-performance SMT IFetch Policies*, implemented within each *SMT* core in a *CMP+SMT processor*; compromising *less than 8%* of the *system throughput*.

We do believe that the *hTCA* implementation presented in this chapter may represent a first step towards future *Heterogeneity-Aware Processors*, able to achieve *complexity-effective* executions in the emerging *many-core era*.

Chapter 6

Further Considerations when Moving to Multicore

When moving from *Multithreaded Singlecore* to *Multithreaded Multicore* some additional challenges may arise. Well-known techniques in *SMTs* may need to be revisited prior to their application to the emerging *CMP+SMT* scenario. In particular, we show that a robust *FLUSH SMT IFetch Policy* may yield worse results than a simple *ICOUNT*. In particular, it suffers a *31%* slowdown when moving from 2 to 4-core *Multithreaded Multicore* scenario. Once analyzed the new challenge, related to the on-chip interconnection network and the *FLUSH* mechanism's static trigger-based design, we present the last contribution of this thesis: the *Multicore FLUSH (MFLUSH)* mechanism.

The *FLUSH* [70] mechanism avoids any running thread from monopolizing the available hardware resources. Built on top of the *ICOUNT* [72] policy, the *FLUSH* mechanism detects loads that experience L2 Cache Misses (unhandled by the ICOUNT policy) and reacts stalling the offending thread; preventing it from monopolizing more hardware resources. Moreover, the newest instructions (until the blocked load) of the offending thread from the offending thread are flushed, freeing the corresponding hardware resources; available for the remainder running applications.

The *MFLUSH* mechanism introduces the *Heterogeneity-Awareness* concept in *IFetch Policies*. It dynamically adapts to the varying conditions, yielding a more *complexity-effective* response to the *heterogeneous* behavior exhibited by the running applications. Yielding results similar to those obtained using an oracle-trigger-based *FLUSH* mechanism, the *MFLUSH* mechanism allows power consumption reductions of up to *20%*, as compared to a traditional *FLUSH* mechanism.

6.1 Introduction

As the transistor count on a single chip augments, Computer Architects strive to find better ways to fully exploit the available hardware budget from an architectural perspective. So, *Uniscalars* gave way to *Superscalars*, and the latter to *SMTs* and *CMPs*. Nowadays, we are witnessing the raise of the *CMP+SMTs* and the advent of the *Many-core Era*, with tens or even hundreds of execution cores along the chip' surface. However, before being able to handle such a great computational power, some basics should be carefully revisited.

On the one hand, conventional *CMP* designs share the second level (L2) cache among all the on-chip cores by means of an interconnection switch. As the number of on-chip cores increases, the pressure on both the L2 cache and the interconnection network is also augmented. As a result, the L2 cache access time turns more *unpredictable*.

On the other hand, the L2 cache access time is used in *SMT* processors to detect L2 cache misses. As shown by Tullsen et al. in [70], L2 cache misses are of key importance in *SMTs*. Thus, a long latency instruction, like an L2 cache miss, in any running thread may stall the whole machine. The *Instruction Fetch (IFetch) Policy* may avoid these harmful situations, determining from which thread(s) instructions are fetched every cycle. Several authors have shown that long latency operations have to be taken into account by the *IFetch Policy* in order to boost *SMT* performance [20, 24, 70, 72]. Some of these *IFetch Policies* track the delay of loads when accessing the outer cache level (the L2 cache in our processor setup) in order to determine whether they miss. Once an L2 cache miss is detected the corresponding thread is stopped/flushed to prevent resource monopolization.

To conclude this PhD dissertation focused on the introduction of the *Heterogeneity-Awareness* concept in the emerging *Multithreaded Multicore Processors*, we revisit a well-known *SMT* technique in this emerging scenario. So, in this last chapter we shed some light on the implications of having multiple *SMT* cores sharing a single L2 cache. We focus our analysis on the application of the *FLUSH* [70] IFetch Policy to the emerging *CMP+SMT* scenario, with multiple *SMT* cores sharing an L2 cache. As we augment the *SMT* core count sharing the same L2 cache both the memory traffic (between each core and L2 cache) and the contention (L2 cache banks and ports) increase. From this analysis, we propose a novel *IFetch Policy* designed to turn *Heterogeneity-Aware* the emerging *CMP+SMT* scenario: the *MFLUSH*. We include a complete evaluation of the *MFLUSH* both in terms of throughput and energy consumption. Our results indicate that the *MFLUSH* succeeds not only in overcoming the specific *CMP+SMT* constraints but also allowing a 20% reduction in the required energy consumption without a significative (less than 3%) system throughput loss.

gzip	a	eon	h	apsi	o	facerec	v
vpr	b	gap	i	wupwise	p	applu	w
gcc	c	vortex	j	equake	q	galgel	x
mcf	d	bzip2	k	lucas	r	ammp	y
crafty	e	twolf	l	mesa	s	mgrid	z
perlbmk	f	art	m	fma3d	t		
parser	g	swim	n	sixtrack	u		

	Number of Threads			
Name	2	4	6	8
xW1	b, j	b, q, t, j	l, b, q, f, t, j	d, l, b, g, i, j, c, f
xW2	n, e	l, n, p, e	g, l, n, p, e, a	b, g, m, n, a, h, o, p
xW3	d, a	d, s, r, a	d, l, s, w, r, a	m, n, r, q, i, j, e, h
xW4	g, f	g, b, m, f	r, g, b, m, h, f	l, b, g, m, n, r, f, s
xW5	r, p	r, j, f, p	h, l, e, r, m, d	q, b, c, k, e, a, o, t

Table 6.1: Workloads used in MFLUSH research.

6.2 Methodology

Since a complete study of all benchmarks is not feasible, due to excessive simulation time, we have randomly chosen some of them comprising 5 workloads for 4 different workload sizes (i.e., 20 workloads). Table 6.1 shows the main simulation parameters and the chosen workloads. The name of each workload is *xWy*, where *x* and *y* stands for the number of threads involved and workload identifier, respectively (e.g., *6W2* identifies the second workload with 6 threads). Each workload size *x* is simulated on a *CMP+SMT* implementation with $\frac{x}{2}$ two-hardware-context SMT cores. All workloads are simulated for a fixed interval of 120 millions of cycles.

6.3 Analysis

We firstly analyze and evaluate the interaction between the shared L2 cache and the *IFetch Policy* implemented within each *SMT* core. We focus on *CMPs* comprised of *SMT* cores, or simply *CMP+SMT*. Each *SMT* core allows two threads running simultaneously and has its private instruction and data cache (see Chapter 2). The first cache level is connected, through an on-chip bus-based interconnection network, to a shared multibanked L2 cache. The Icache and Dcache of each core is connected to all the shared L2 cache banks. Both the memory traffic, between L1 and L2 caches, and contention effects, regarding the use of each shared L2 cache bank, are considered. Regarding resource sharing, two well-known *SMT IFetch Policies* are used in our research: *ICOUNT* and *FLUSH*.

The *ICOUNT* policy [72] prioritizes threads with fewer instructions in the pre-issue stages, and presents good results for threads with high *Instruction Level Parallelism* (*ILP*). However, *SMTs* have difficulties with threads that experience many loads that miss in the L2 cache. When this situation happens, the *ICOUNT* does not realize that a thread can be blocked on an L2 cache miss and will not make forward progress for many cycles. Depending on the amount of instructions dependent of the blocked load, many processor resources may be blocked and the total throughput suffers from a serious slowdown.

The performance of *IFetch Policies* dealing with load miss latency depends on the following two factors: the *Detection Moment* (*DM*) and the *Response Action* (*RA*). The *DM* indicates the moment in which the policy detects a load that fails or is predicted to fail in cache. Possible values range from the fetch of the load until the moment that the load finally fails in the L2 cache. Two characteristics associated with the *DM* are the *reliability* and the *speed*. The higher the *speed* of a method to detect a delinquent load, the lower its *reliability*. On the one hand, if we wait until the load misses in L2 (*Non-Speculative implementation*), we know for certain that it is a delinquent load: *totally reliable* but *too late*. On the other hand, we can predict (*Speculative implementation*) which loads are going to miss by adding a load miss predictor to the front-end. In this case, the *speed* is *higher*, but the *reliability* is *low* due to predictor mispredictions. The *RA* indicates the behavior of the policy once a load is detected or predicted to miss in cache. That is, it defines the measures that the *IFetch Policy* takes for delinquent threads.

In [70] several *RA* are proposed. We focus on the mechanism leading to the best performance, called *FLUSH*. As a result of applying *FLUSH*, the offending thread temporarily does not compete for resources. More importantly, the hardware resources used by this thread are freed, giving the other threads full access to them. Several *DM* are proposed for the *FLUSH* response action.

- *Delay after issue DM:* When this *DM* is used, a load is declared to miss in the L2 cache when it spends more cycles in the cache hierarchy than needed to access the L2 cache, including possible resource conflicts. We will refer to this *FLUSH*'s *DM* as *Speculative* (*FL-SX*), where *X* stands for the delay (cycles) after which the mechanism is triggered.

- *Trigger on miss DM:* In this case we wait until the load miss in the L2 cache to start the corresponding *RA*. We will refer to this *FLUSH*'s *DM* as *Non-Speculative* (*FL-NS*).

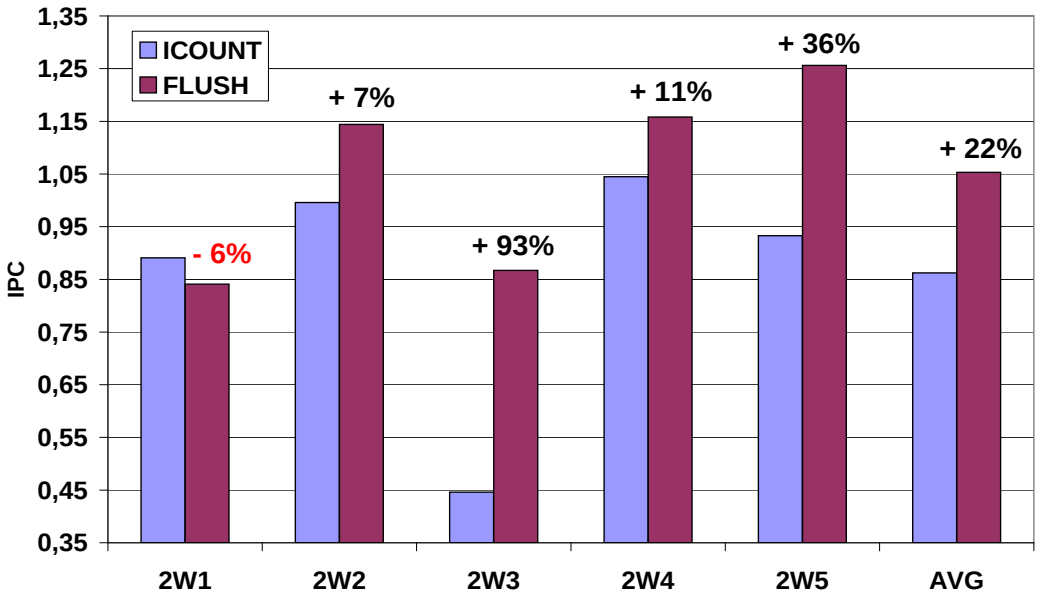


Figure 6.1: Throughput in single-core SMT.

6.3.1 Single-core analysis

According to our simulation parameters (see Table 6.1) we chose *30 cycles* (*FL-S30*) as *FLUSH* trigger, that is the delay waited prior to activate the *FLUSH* mechanism once a load is issued from the corresponding queue.

Our results are consistent with [70]: the *delay-after-issue DM* yields better results than *trigger on miss*, both improving *ICOUNT*. For this experiment, we simulated a single-core SMT configuration. In this uniprocessor, with two hardware contexts, we ran all 2-thread (*2Wy*) workloads in Table 6.1. Figure 6.1 shows the comparison between *ICOUNT* and *Speculative FLUSH (FL-S30)* results. From these results it can be asserted that the *FLUSH* mechanism effectively reduces system throughput losses in workloads containing threads with bad memory behaviors. Thus, the *FLUSH* mechanism yields speedups of up to *93%*, with average speedup of *22%*. However, as described in the following section, these asserts are highly dependent on the amount of replicated *SMT* cores.

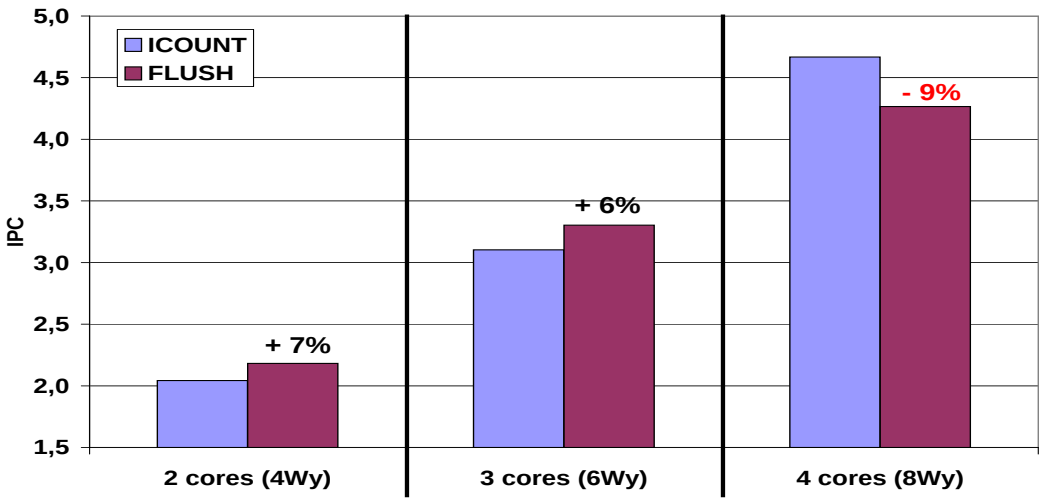


Figure 6.2: Average throughput in multicore CMP+SMT configurations.

6.3.2 Multiple-core analysis

Next, we simulated the remainder workloads in Table 6.1, replicating *SMT* cores with two threads per core. Figure 6.2 shows the average results per each workload size. These results point out that the prior asserts made for the single-core case, regarding the performance of the *FLUSH* mechanism, are not valid for the multicore *CMP+SMT* configurations. In fact, as we increase the amount of replicated *SMT* cores the *22%* average speedup, obtained with the *FLUSH* mechanism in a single-core *SMT* when compared to *ICOUNT*, experiences a progressive reduction. With a 4-core configuration (8 thread workloads - *8Wy*), the *FLUSH* mechanism's performance improvement disappears yielding a *9%* average slowdown.

In order to shed some light into the rationale behind these results, we deeply analyzed the influence of the access time to the shared L2 cache. Figure 6.3 shows the average number of cycles required for each load that hits on the shared L2 cache, since it is issued from the load/store queue until it is finally served. For this measurement we use the *ICOUNT* policy since it does not alter the L2 cache access patern.

Figure 6.3 points out that the probability of suffering from high latencies in L2 cache accesses increases with the amount of SMT cores. As indicated in Table 6.1, each of the 4 banks of the shared L2 cache is single-ported and has an access latency of 15 cycles. That is, two consecutive accesses to the same L2 cache bank cannot be served in less than 15 cycles. Each *SMT* core implements 2 Load/Store Units, shared by the two threads

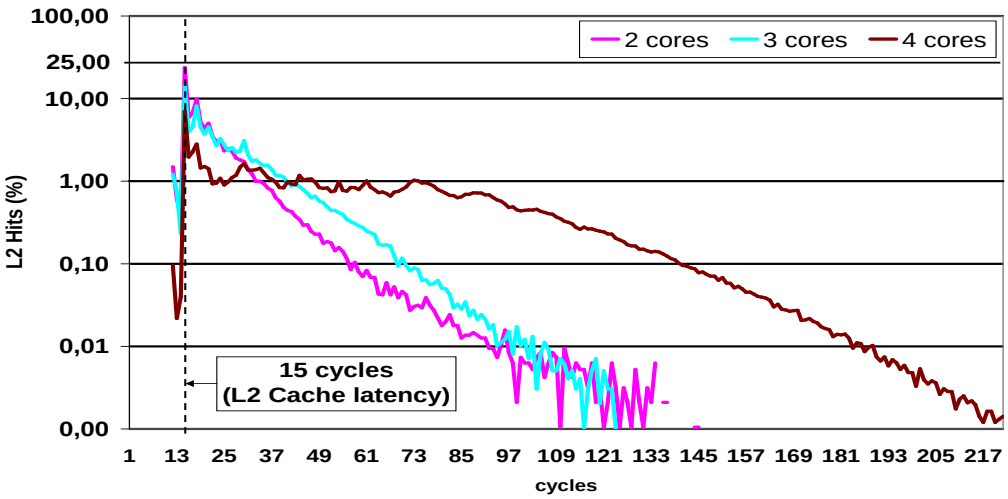


Figure 6.3: Average L2 cache hit time.

running in the core. Within each core it is also implemented a 16-entry MSHR queue that keeps track of the outstanding memory requests. In case of L2 hits, consecutive accesses to the same L2 cache bank may overlap yielding a higher access time. As an example, the fourth consecutive L2 hit to the same L2 cache bank would experience a 45-cycle delay. Each additional SMT core increases in 2 the number of loads that can be issued in a single cycle, with the consequent increment of the pressure on both the interconnection network (L1-L2 bus) and the shared L2 cache.

Figure 6.3 also indicates that the dispersion of the L2 access time also increases with the number of *SMT* cores. Focusing on the average L2 hit time for a 4-core implementation in Figure 6.3, about half the L2 hits are equally distributed in the range of 20-70 cycles. This fact points out that there is no a single threshold, to be used as trigger value for the *FLUSH* mechanism, which provides good results for all cases. This high variability in the L2 cache access time hampers the predictability of the L2 behavior:

- On the one hand, if we set a low threshold value the number of *false misses* increases. That is, the number of long-latency L2 hits predicted as L2 misses. As a result, the performance of the *FLUSH* policy is heavily affected.

- On the other hand, if we set a high threshold value the number of cycles a thread can clog resources increases, leading to performance loss. We comment this issue in the next section.

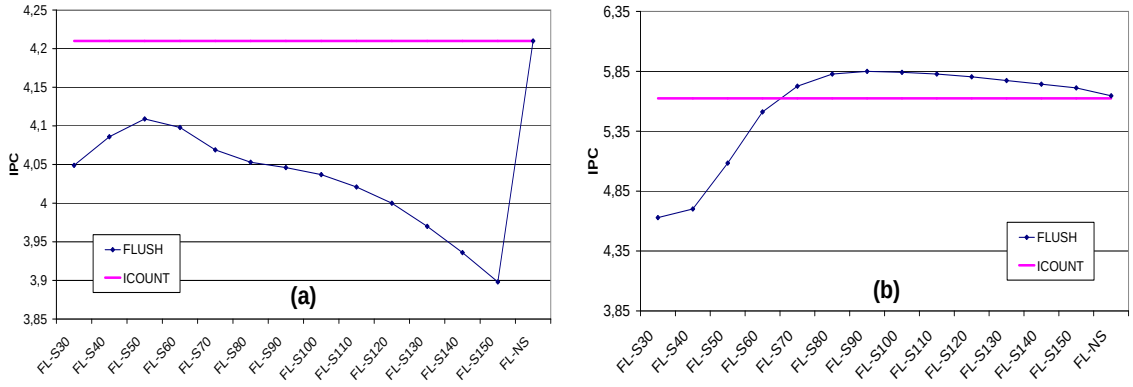


Figure 6.4: Detection Moment Analysis.

To sum up, the performance of the *FLUSH* mechanism exhibits a clear trend to get diminishing returns as we increase the number of *SMT* cores in a *CMP+SMT* scenario. In fact, the *FLUSH* mechanism turns ineffective just by passing from a dual core to a quad core implementation, as depicted in Figure 6.2.

6.3.3 Detection Moment Analysis

The results in Figure 6.3 exhibit higher levels of dispersion as increases the amount of *SMT* cores. In this section we analyze how does this issue affect the choice of the right trigger for the *FLUSH* mechanism. Thus, we ran some additional simulations covering a wider *DM* spectrum. For an explanatory analysis, we chose two representative 8-thread workloads: *(a) 8W3* (see Table 6.1) and *(b)* an 8-thread workload comprised of instances of *bzip2* and *twolf*, where instances of the two applications never share a single core. Figure 6.4 shows the results obtained using different values for the *FLUSH*'s trigger, ranging from *30* to *150* cycles. The *Non-Speculative implementation (FL-NS)* is also included in Figure 6.4.

In Figure 6.4*(a)*, the *trigger* that yields the *highest throughput* is *50 cycles*. However, compared to speculative instances, the *non-speculative FLUSH* implementation yields the highest overall throughput. In Figure 6.4*(b)*, the *best trigger* value is *90 cycles*. These examples illustrate that *there may be different trigger values which best balance the amount of false misses and clog resources*, yielding the highest overall throughput. That is, *the choice of the right value depends on each specific workload*.

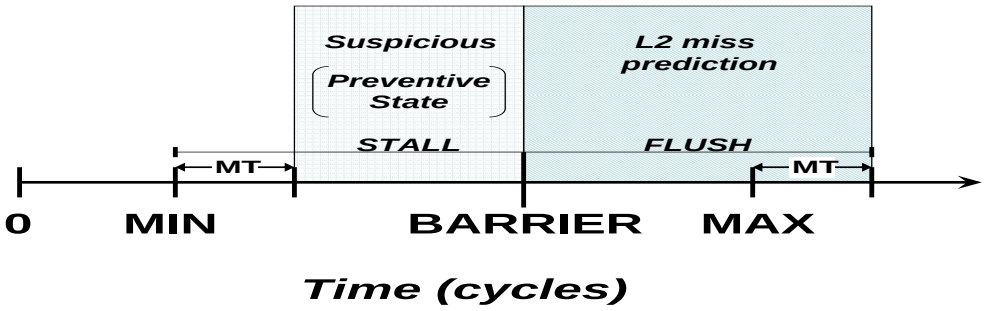


Figure 6.5: MFLUSH Operational Environment.

6.4 The MFLUSH Policy

The *MFLUSH* mechanism adapts the *FLUSH* [70] and *STALL* [70] phylosophy to the emerging *CMP+SMT* scenario. Built on top of *ICOUNT* [71], the *MFLUSH* mechanism *avoids the waste of resources by threads blocked waiting for memory*. Whenever a thread waits for a memory access to be resolved, the *MFLUSH* mechanism predicts its resolution time and reacts accordingly. Since the *CMP+SMT* scenario has less *memory access predictability* than the prior *SMT* scenario, this issue turns into a non-trivial task. The *MFLUSH* is designed to cope with the varying workload behavior and memory traffic conditions of the emerging *CMPs* comprised of *SMT* cores sharing one or multiple L2 Caches. Thus, it adapts its L2 miss predictions to the varying conditions instead of using an heuristic prediction value, as done in *FLUSH*.

The MFLUSH mechanism establishes, according to the specific system characteristis, an *Operational Environment* as shown in Figure 6.5. The *MFLUSH* mechanism predicts for each memory access its resolution time, based on prior accesses. These predictions fall in the *MIN - MAX range* (See Figure 6.5), where *MIN* and *MAX* correspond to the *L1 and L2 cache miss latency*, respectively. As seen in prior sections, the access time of an L2 cache may experience high variability when multiple *SMT* cores share it. The more cores sharing a single L2 cache and interconection bus, the more traffic/memory contention. In order to consider this factor, the *MFLUSH's Operational Environment* includes a *Multicore Traffic (MT) delay*, that is added to both *MIN* and *MAX* values as shown in Figure 6.5. The *MT delay* obeys the following equation:

$$MT = (L1\_L2\_Bus\_delay + L2\_Bank\_Acc\_delay) * (Num\_Cores - 1)(cycles)$$

Due to the high-variability of the L2 cache access time in *CMP+SMT* implementations sharing a single L2 cache, it cannot be used an static value to predict L2 cache misses, as done by the the *FLUSH* mechanism in *SMT* processors. For each L2 cache access, the *MFLUSH* mechanism predicts its resolution time according to the varying conditions of *memory traffic* and *contention*. The mechanism to obtain these *predictions* is described in Section 6.4.1. Based on each prediction, the *MFLUSH* dynamically estimates a *Barrier* value for each memory access. Whenever a memory access lasts more than *Barrier* cycles without being resolved it is considered to miss in the L2 cache. In that case, the *FLUSH* mechanism is triggered (See Figure 6.5), both stalling the offending thread and freeing some of its hardware resources (e.g., rename registers, instruction queue entries, etc). Exactly as in the *FLUSH* mechanism, the offending thread remains idle until the memory access is resolved. During this period of time, the freed resources, originally devoted to the newest instructions of the offending thread, may be used by all other running threads in the same *SMT* core. The *Barrier* estimation obeys the following equation:

$$BARRIER = \frac{L2prediction+MIN}{2} + MT \textit{ (cycles)}$$

In presence of high memory traffic/contention, a *late L2 cache hit* may be as harmful as an *L2 cache miss*. In that case, the *Barrier* value could be too high, involving a possible resource waste. In order to reduce the negative effects of *Late L2 hits*, the *MFLUSH* mechanism considers *suspicious* all L2 cache accesses that *last more than* $MIN + MT$ *execution cycles* to be resolved. As shown in Figure 6.5, the *MFLUSH's Operational Environment* establishes a *Preventive State* for all *suspicious* memory accesses. Thus, any threads with a *suspicious* in-flight memory access is stalled by the *MFLUSH* mechanism, preventing it from obtaining additional hardware resources. However, a thread in the *Preventive State* is still running and can make forward progress with the instructions priorly fetched into the execution pipeline. Whether the *suspicious* memory access is resolved before reaching the *Barrier* the corresponding thread is removed from the *Preventive State*. In that case, the thread is allowed to fetch new instructions into the pipeline. Otherwise, the *suspicious* memory access is predicted as an L2 miss, and the *FLUSH* mechanism is triggered.

Triggering the *FLUSH* mechanism has a *cost*, both in terms of *performance* and *power consumption*. A flushed thread is stalled until the offending memory access (load instruction) is resolved, avoiding additional forward progress in the whole thread. Besides, all the newest instructions issued, from the last fetched instruction to the offending memory instruction, are flushed away from the execution pipeline. By the time the offending memory access is resolved, the thread resumes its execution, fetching again in the execution pipeline all flushed instructions. Consequently, all *flushed instructions have a*

higher cost in terms of power consumption. The exact cost depends on the pipeline stage the instruction was by the time it was flushed. Therefore, making an smart use of the *FLUSH* mechanism is critical to obtain both good performance and a moderated power consumption.

6.4.1 MFLUSH Hardware Support

In order to obtain both *fast* and *accurate* dynamic predictions, the *MFLUSH* policy requires some *additional hardware support*, shown in Figure 6.6. Each *SMT* core holds an 8-bit register (*MCReg*) per each L2 cache bank used. The *MCReg* register keeps the latency of the last *L2 cache hit* in the corresponding L2 cache bank. The *MFLUSH* mechanism *assumes the same behavior in consecutive accesses to the same L2 cache bank*. Hence, the *MFLUSH* uses the value in the corresponding *MCReg* register to quickly predict the latency of the next access to the same L2 cache bank.

Figure 6.6 shows an example for a 4-core *CMP* implementation where all cores share a 4-banked L2 cache. Each core is connected to each of the L2 cache banks by means of a shared bus. In case of an L1 cache miss in core 0, the L2 cache bank that should contain the requested data is first determined using the address of the corresponding memory access. The *MFLUSH* mechanism then accesses the corresponding *MCReg* register and uses its content as prediction of the L2 hit latency. As a matter of example, if bank 2 was acceded, the latency prediction would be of *55* cycles, as shown in Figure 6.6. Using this L2 cache hit latency prediction the *MFLUSH* mechanism proceeds with the appropriate response according to the varying memory traffic/contention conditions, as described in Section 6.4.

The *MCReg* registers admit more complex configurations, involving queues (i.e., history length : *MCReg* $= 1$; $queues > 1$) and more complex functions to determine the prediction from all queue entries. However, to keep it simple and fast we use a single *MCReg* register per core and per L2 cache bank. Our results confirm that this choice allows tracking quick memory behavior changes.

6.4.2 MFLUSH Throughput Evaluation

Figure 6.7 shows the system throughput evaluation for *CMP+SMT* implementations with 2, 3, and 4 cores, using 4, 6, and 8-thread workloads respectively. The results in Figure 6.7 include, for each workload, 4 evaluations using different IFetch Policies: *ICOUNT, Speculative FLUSH with 30-cycle trigger (FLUSH-S30), Speculative FLUSH*

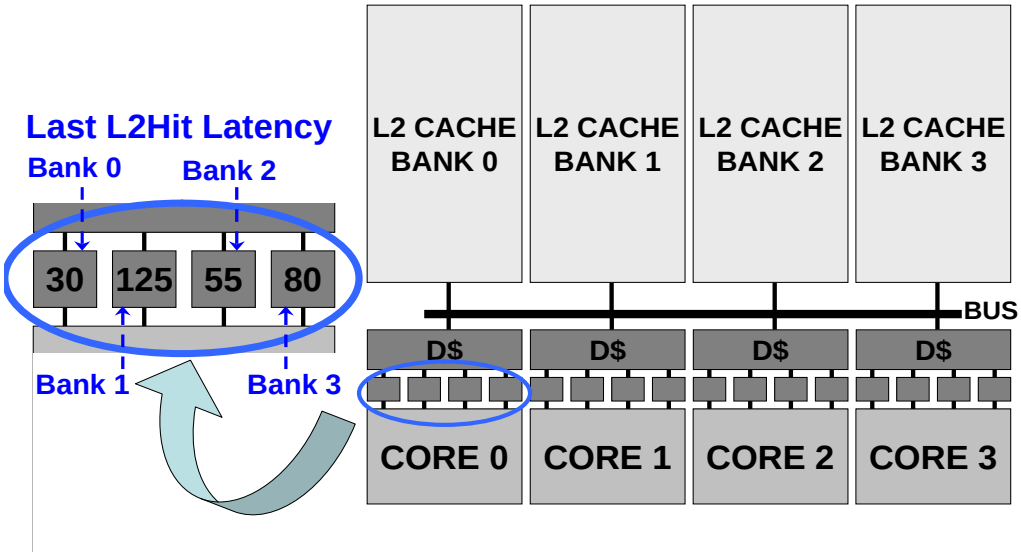


Figure 6.6: MFLUSH hardware support for a 4-core CMP with a 4-banked L2 Cache.

with 100-cycle trigger (FLUSH-S100), and *MFLUSH*. Figure 6.7 shows that in general, the highest results are obtained using *FLUSH-S100*. However, this assert is not true for all considered workloads, as in the case of *4W4*, *6W4*, and *8W1*, in which the *MFLUSH* yields the highest results.

The results in Figure 6.7 also confirm that *a bad trigger choice in Speculative FLUSH*, as happens with *FLUSH-S30* (30 cycles) in most of the cases, *may yield even worse results than the ICOUNT* IFetch Policy. Examples of this situation are *4W1*, *6W1*, and *8W4*. Recall that this trigger choice yields an average *22%* speedup over *ICOUNT* in single-core SMT, as shown in Figure 6.1. Something similar occurs in the *4W3* workload, where the *ICOUNT* IFetch Policy yields *4%* speedup over *MFLUSH*. This isolated fact is due to the specific workload and microarchitecture characteristics.

Focusing on average results, it can be asserted from Figure 6.7 that the *MFLUSH effectively succeeds in giving high throughput results*, *2%* close to the best performing *Speculative FLUSH* option (*FLUSH-S100*). This goal is achieved without requiring additional information regarding neither the trigger value to be used nor the underlying *CMP+SMT* implementation. Recall that *Speculative FLUSH* requires to specify apriori a trigger value (i.e., a 100-cycle trigger for the *FLUSH-S100*).

6.4.3 MFLUSH Power Consumption Evaluation

The *FLUSH* mechanism represents a *high-power-consumption alternative*, aimed at *throughput-oriented scenarios*, in which the *system throughput* is the main concern regardless of the power required. Flushing away instructions from the pipeline, and having to refetch them afterwards, implies an *additional energy cost*. This cost depends on the pipeline stage in which the instruction was by the *flush time*. In order to measure the proposed *MFLUSH* mechanisms *power-efficiency* we use the *Energy Consumption Factor* described in Section 5.4. This factor allows to estimate the *additional energy* required by the *FLUSH* mechanism, *tracking the number of flushed instructions in each pipeline stage and applying the corresponding factor value*. Compared to *FLUSH*, the *MFLUSH* mechanism only adds a read access to a local 8-bit register on L1 cache misses. A write access to that register is only required in case of L2 hits. Due to its reduced cost, the *MFLUSH* hardware support is not added to the *Energy Consumption Factor*.

Nowadays, the power-aware constraints in processor designs are present even for throughput-oriented scenarios. Although there are still scenarios in which obtaining the highest throughput is the main concern, the power constraints impose severe constraints on how this goal is achieved. Consequently, any architectural advance which reduces the energy consumption without hardly compromising the total throughput is of particular interest.

Figure 6.8 shows the *Wasted Energy* implied by each *Speculative FLUSH* implementation (*FLUSH-S30* and *FLUSH-S100*) and *MFLUSH* IFetch Policy. This *Wasted Energy* strictly corresponds to the additional energy required by the *FLUSH* mechanism, which requires refetching flushed instructions once resolved the corresponding memory accesses. The *Wasted Energy* is measured in *energy units* in Figure 6.8, that is the amount of energy required to commit 1 instruction. The results in Figure 6.8 are obtained using the *Energy Consumption Factor* (See Section 5.4) and the number of instructions flushed in each pipeline stage.

The results in Figure 6.8 point out that *FLUSH-S100* wastes in average *10%* more energy than *FLUSH-S30*. Although *FLUSH-S100* involves less total flushes than *FLUSH-S30*, it involves more instructions to be reflushed on each pipeline refill. Waiting more time implies more instructions fetched into the execution pipeline by the time the *FLUSH* mechanism is triggered, and therefore a greater amount of instructions to refetch. Figure 6.8 also confirm that *aggressive flushing comes at an extra energy cost*. In all cases the *MFLUSH* obtains significant energy consumption reductions, reaching *20%* when compared with the best-performing *Speculative FLUSH* choice (*FLUSH-S100*) that, as seen in Section 6.4.2, obtains a marginal *2%* throughput improvement over *MFLUSH*.

Consequently, the *MFLUSH* IFetch Policy *constitutes* not only *a solution to* the *unpredictability* of the L2 cache latency in the emerging *CMP+SMT* scenario but also *provides an important energy consumption saving*.

6.5 Related Work

The *FLUSH* mechanism was proposed by Tullsen et al. in [70] as an improvement for the *ICOUNT* [72] policy in single-core *SMT* processors. The *ICOUNT* policy has difficulties with threads that experience many loads that miss in L2 cache, being unable to realize that a thread can be blocked on an L2 cache miss and do not make forward progress for many cycles. Depending on the amount of instructions dependent on the blocked load, many processor resources may become clogged and the total throughput suffers from a serious slowdown. Several *FLUSH* implementation choices were analyzed in [70], focusing on the simplest and less expensive ones : *Trigger on Delay* or *Speculative FLUSH*. With the rise of the emerging *CMP* comprised of *SMT* cores, like the *IBM POWER5* [60] and *POWER6* [39], it must be faced up a new challenge: the *unpredictability* of the L2 cache hit latency.

The *MFLUSH* mechanism adapts the *FLUSH* and *STALL* phylosophy in prior *SMTs* to the new *CMP+SMT* scenario, obtaining both *dynamic adaptability* to the varying memory traffic/contention conditions and important *energy consumption savings*. This goal is achieved applying the *Heterogeneity-Aware* concept to the *FLUSH* mechanism; since the workload behavior is inherently heterogeneous, so the traffic and memory contention conditions would be. By giving to each execution thread the appropriate portion of the processor resources, adjusting its mechanism's trigger value, we would be able to achieve more *complexity-effective* executions.

Several authors have shown that long latency operations have to be taken into account by the IFetch Policy in order to boost *SMT* performance [20, 24, 70, 72]. In order to apply them to the new *CMP+SMT* scenario a similar analysis, as done in this paper, should be performed. Revisiting prior well-known high-performance proposals when moving to a new application scenario generally requires this type of analyses.

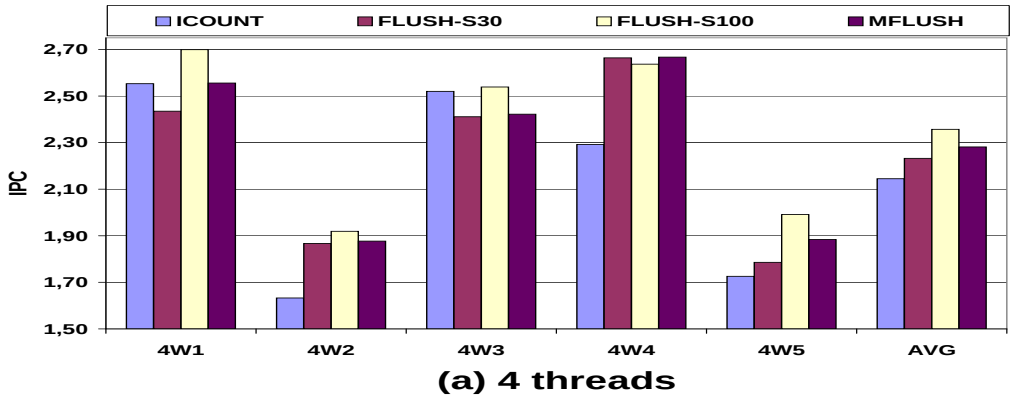
Shin et al. propose an *Adaptative Dynamic Thread Scheduling* (ADTS) [58] to manage the resource sharing in *SMT* processors. The *ADTS* improves the system throughput in *SMT* processors by adapting the underlying IFetch Policy to the workload characteristics. Thus, the *ADTS* changes the IFetch Policy used among *ICOUNT* [72], *BRCOUNT* [72], and *L1DMISSCOUNT* [72], according to the varying *workload characteristics*. In this chapter we propose the *MFLUSH* mechanism, which adapts the *FLUSH* and *STALL* philosophy to the emerging *CMP+SMT* scenario.

6.6 Chapter Summary

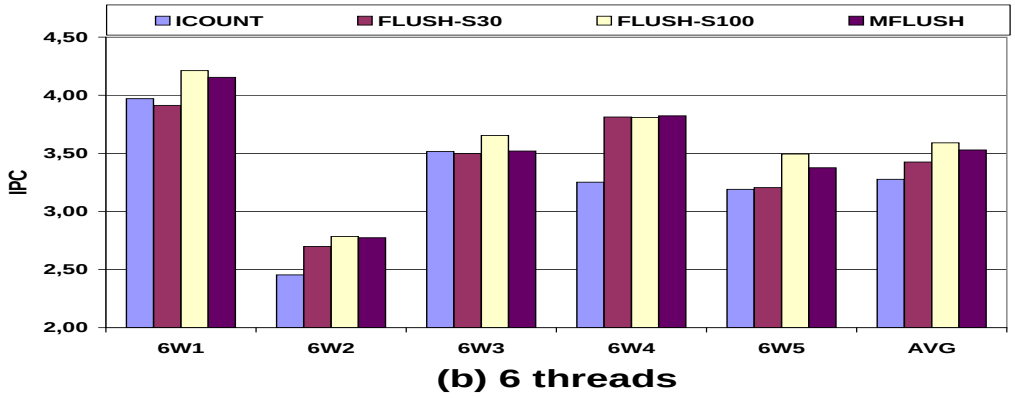
In this chapter we analyze the new challenges to be faced up in future high-degree Multithreaded *CMPs*, with multiple *SMT* execution cores sharing an L2 cache (*CMP+SMT*). In particular we focus on probably the most important *SMT* issue: the *Instruction Fetch (IFetch) Policy*. Considering *ICOUNT* and *FLUSH* IFetch Policies we show results which evidence that *CMP+SMT* may not simply relly on *SMT* IFetch Policies to boost overall throughput. *SMT* IFetch Policies must be revisited when moving to the new *CMP+SMT* scenario.

From the exhaustive analysis included herein, it is proposed a novel IFetch Policy designed to cope with the emerging *CMP+SMT* scenario: the *MFLUSH*. The *MFLUSH* mechanism introduces the *Heterogeneity-Awareness* concept in *IFetch Policies*. It dynamically adapts to the varying memory conditions, yielding a more *complexity-effective* response to the *heterogeneous* behavior exhibited by the running applications. Yielding results similar to those obtained using an oracle-trigger-based *FLUSH* mechanism, the *MFLUSH* mechanism allows power consumption reductions of up to *20%*, as compared to a traditional *FLUSH* mechanism.

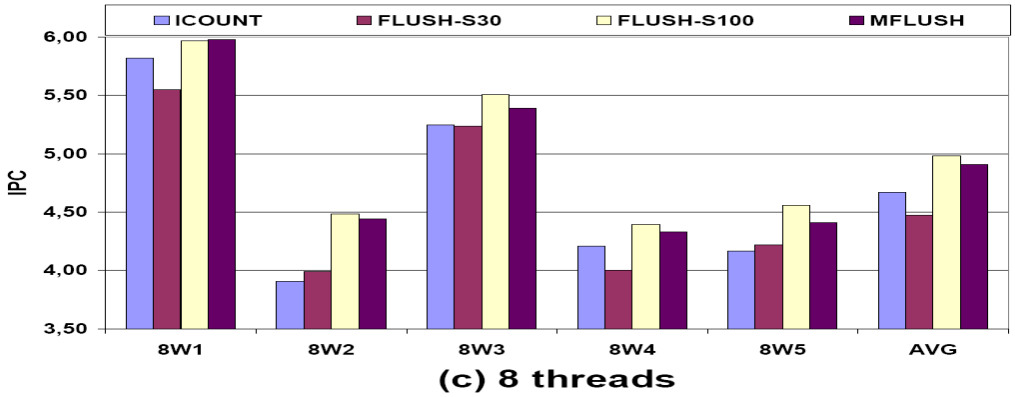
We include a complete evaluation of the *MFLUSH*, both in terms of *throughput* and *energy consumption*. Our results indicate that the *MFLUSH* succeeds not only in overcoming the specific *CMP+SMT* constraints but also allowing a 20% energy consumption reduction without a significative system throughput loss. These results confirm that giving each execution thread the appropriate amount of processor resources, by adjusting the FLUSH mechanism's trigger value so that the amount of refetched instructions would be minimal, we are able to achieve more complexity-effective executions in *Multithreaded Multicore Processors*; that is, being *Heterogeneity-Aware*.



(a) 4 threads



(b) 6 threads



(c) 8 threads

Figure 6.7: Throughput Results.

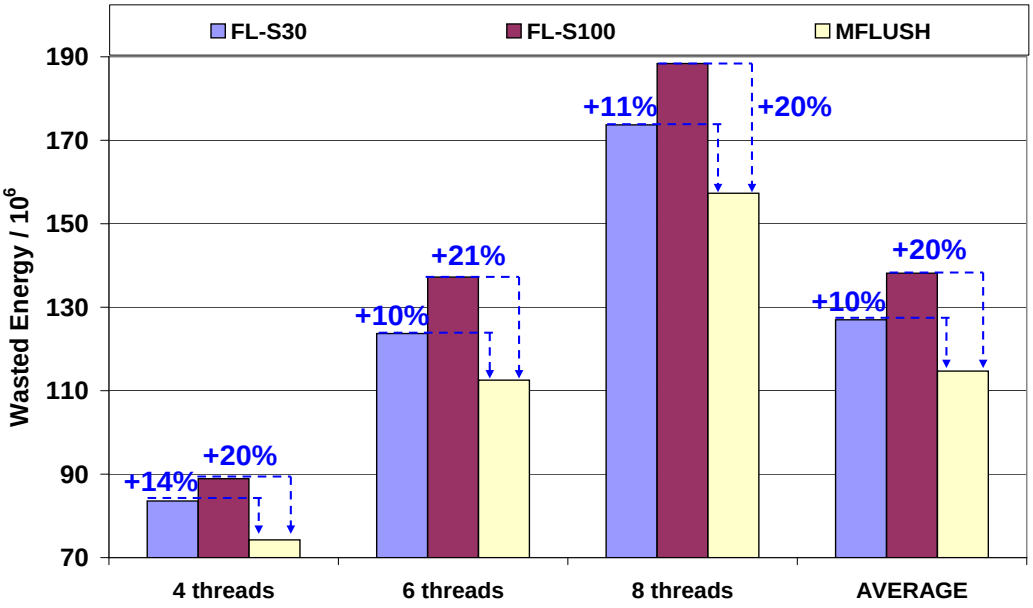


Figure 6.8: FLUSH Wasted Energy.

Chapter 7

Conclusions

This chapter lists the main conclusions of this thesis as well as future directions.

7.1 Thesis conclusions

Due to limitations in the applications' *Instruction Level Paralellism (ILP)*, current trends in Computer Architecture rely on exploiting *Thread Level Paralellism (TLP)*. *Big and complex* processors, like the *Intel Pentium 4* [5], are now being replaced by *smaller and simpler* multithreaded *Processing Elements (PE)*, or cores, replicated along the chip's surface, as in the case of the *IBM POWER5* [60] and *POWER6* [39]. In some cases, as with the *Cell Processor* [27], these *PEs* are not just replicated: they configure an heterogeneous processor layout.

Whenever the hardware is statically partitioned into clusters, as done in *CMPs*, *CMP+ SMTs*, and many other clustered processor implementations, it is crucial to properly *match* the *applications' needs* with the *hardware resources* of each cluster. Despite applications are inherently heterogeneous, that is they have different needs as compared to both other applications and different portions of execution (*program phases*) of the very same application, this *matching process* is straightforward in *homogeneous partitions*. However, when not all the clusters do have the same amount of resources, unproperly matching applications with clusters may involve a serious throughput degradation.

The main contribution of this thesis is the introduction of the *Heterogeneity-Awareness* concept in the design of *Multithreaded Multicore Processors*. A *Heterogeneity-Aware Multithreaded Multicore* processor explicitly takes into account the inherent heterogeneity in the applications' behavior and compares it with the hardware characteristics to per-

form the most appropriate *software-hardware matching*, that is the one that yields the *highest throughput* involving the *lowest energy consumption*. In this thesis it is shown that this *software-hardware matching* is a *key factor* not only for *heterogeneous hardware partitions*, in which this matching appears more evident, but also for *homogeneous hardware partitions*, in which multiple applications are run sharing the resources belonging to a single partition.

- *For heterogeneous hardware distributions*, we propose the *heterogeneously distributed SMT (hdSMT)* architecture, that improves the *complexity-effectiveness* of the processor. Our results confirm that the *hdSMT* approach increases the *IPC/Area ratio* in an average 14%, as compared to a *monolithic SMT* processor. Additionally, our results also reveal that, depending on the characteristics of each workload, reducing the resource contention of some applications we obtain improvements not only in terms of *complexity-effectiveness* (*IPC/Area*) but also in terms of raw throughput (*IPC*), by assigning them to different clusters with private resources. In this sense, memory-bounded workloads experience raw throughput improvements of up to 18%.

- *For homogeneous hardware distributions*, we propose the *Thread to Core Assignment (TCA) Algorithm*. Although apriori less evident, an exhaustive analysis of the homogeneously distributed hardware partitions reveals that the traditional scheduling process performed in multithreaded processors required an additional and intermediate step when moving to the new multithreaded multicore scenario: the *Thread to Core Assignment (TCA)*. We also have studied the relation between this new scheduling step and the resource sharing step, obtaining the following main conclusions:

  I A *good IFetch Policy* reduces the negative effect of an inappropriate *TCA*.

  II An *appropriate TCA* improves the results obtained regardless the underlying *IFetch Policy*.

  III An *inappropriate TCA* could negate the performance advantage of a better *IFetch Policy*.

  IV There is not a single *TCA* good for all cases.

  Due to its simple design, the proposed *TCA Algorithm* represents an easy-to-implement solution for the software-hardware matching in multithreaded multicore scenarios. So, main processor vendors could provide the *TCA Algorithm* implementation for each of their new products just as device drivers are provided nowadays. The results included in this thesis confirm that this algoritm yields average speedups of up to 21%.

Once analyzed the homogeneous hardware distributions, we have also started the way back to the heterogeneous distributions. In this sense we focus on probably the most important issue in the *SMT* field: the *IFetch Policy*. The proposed *heterogeneous Thread to Core Assignment (hTCA) Framework* is an OS-driven Framework for *Complexity-Effectiveness* in *Multithreaded Multicore Processors*. Explicitly taking into account the heterogeneity in the running software, the *hTCA Framework* adapts the hardware to the workload in order to reduce the energy consumption without significantly affecting the system throughput. Our results indicate that the proposed framework achieves a *95% accuracy* when selecting the optimal choice for each case.

When moving from heterogeneous to homogeneous hardware distributions we also jumped from clustered *SMT* processors to *CMP+SMT* processors. When doing this change of scenario we realized that some of the well-known techniques in the prior *SMT* field were not valid for the new *CMP+SMT* scenario. In particular we found that the traditional *CMP* scheme, with a shared L2 cache among all *SMT* replicated cores, while optimal in terms of cache usage, involves additional challenges that must be revisited in this new *CMP+SMT* scenario. As an example, when revisiting some of the most well-known Instruction Fetch Policies in *SMTs* we observed that the *ICOUNT* fetch policy obtained better results than the *FLUSH* fetch policy. Recall that the *FLUSH* policy was built on top of *ICOUNT* to improve it against long-latency instructions. To solve this problem we proposed the *MFLUSH* policy, specifically designed to adapt the *FLUSH/STALL* phylosophy to the new *CMP+SMT* scenario. Our results indicate that the *MFLUSH* not only success in adapting the FLUSH policy to the *CMP+SMT* scenario, avoiding degradation as we increase the amount of *SMT* cores, but also reduces its energy consumtion, with an average 20% energy saving.

In the course of the thesis, throughout all these years of research we have been forced to explore an extremelly wide design space. So, we have covered a very wide range of implementations starting from an heterogeneosly distributed hardware, with a shared fetch engine and multiple heterogeneous execution pipelines (from decode to commit), until multithreaded multicore processor implementations with adaptable IFetch Policy (on/off on demand). This would not have been possible without a very flexible simulation tool that allowed to simulate a great amount of different scenarios and configurations. In this sense, we want to emphasize the importance of the *Multi-Purpose Simulator (MPsim)*, designed and developed specifically for this thesis. The *MPsim's* relevance has growth during the recent years, spreading out of this thesis to become a *key tool* in our reserach group, both in the *DAC* and the *BSC*. Recently, the *University of Las Palmas de Gran Canaria (ULPGC)* has joined to the *MPsim Community*. What is more, the *MPsim* continues evolving to offer more attendance to its users. Some temporal *BSC* workers, as *Domen Novak*, has been employed to specifically develop additional *MPsim modules*, that have

been added to the whole *MPsim Project*. The latest item in the *MPsim Project* involves the integration of the brandnew *COTSon* tool, developed by *HP Labs Barcelona*, into the *MPsim Project*.

As a final reflection exercise, after all the research done in the homogeneous/heterogeneous hardware partition field, I do believe that the future of Computer Architecture will face up *heterogeneously distributed chips*, with a *high concern for complexity-effectiveness*. In this future processor generations the hardware would dynamically adapt to the varying requirements of the running software over time, striving to yield the maximum throughput involving at the same time the lowest energy consumption. This way for example, we would see very-low-power mobile processors with a computational power that exceeds any of the current state-of-the-art high-end processors nowadays. Cell phones with holoprojections, that would allow to physically intarct with 3D projections of people thousands of miles away, would become a reality thanks to this kind of low-power/high-end processors, designed to achieve *complexity-effective* executions using a highly-adaptative heterogeneously distributed hardware that dynamically reacts to the varying conditions of the running software.

7.2 Future work

This thesis opens up several topics from which we emphasize the following:

- *Aggressive hdSMT processors* with advanced fetch units and fully-dynamic migration policies. Currently we are working on a decoupled fetch unit which uses traces of instructions as minimal fetching unit.
- *TCA Algorithm implementations* for *heterogeneous Multithreaded Multiprocessors*, like the *Cell processor*. We are now working on extensions of the *hTCA Framework* in which we gradually introduce an heterogeneous distribution of the processor resources among all the constituent replicated cores.

We are already working on some of these topics and many other more, opened up throughout the research involved by this thesis. The perspectives are promising, with plans of starting at least 3 more thesis from the final state of this thesis. Only time would say how far this thesis would reach and how important/referenced would it be in the future.

Chapter 8

Appendix:
The MPsim Simulation Tool

Computer Architecture has experienced great advances in the last decades. Thus, we have witnessed the raise of *Superscalars*, *Simultaneous Multithreading (SMT)* and *on-chip Multiprocessors (CMP)* among others. All these novel ideas had to be evaluated in order to measure their benefits and potential. To perform this evaluation, computer architects require simulation tools which model the corresponding idea and allow simulating its execution results, employing a set of benchmarks. The accuracy of the model employed is in tune with the research requirements. Thus, while in industry computer architects are highly constrained to an specific product, requiring a highly accurate model, in the academia computer architects generally focus on more long term and less specific research topics. Obviously, the computational cost of the model employed is directly proportional to its accuracy. Consequently, the research in the academia generally employs general-purpose simulation tools, closer to their research interests and computational possibilities.

Among the general-purpose simulation tools typically employed in the academia during the last decade we find Simplescalar [22] and SMTsim [71] simulators. The Simplescalar models a single-core *Superscalar* processor with 5 pipeline stages while the SMTsim models a single-core *Superscalar/SMT* processor with 8 pipeline stages. On top of both simulators, several branch predictors and instruction fetch policies, so as new proposals, may be added. Regarding the Memory Subsystem, both simulators model two cache levels (optionally up to the third cache level), with a single Instruction Cache, Data Cache, ITLB, DTLB, L2 Cache. However, while the Simplescalar has a very simple memory model, in which each memory access is deterministically resolved, the *SMTsim* non-deterministically manages the memory accesses by means of an event queue, which

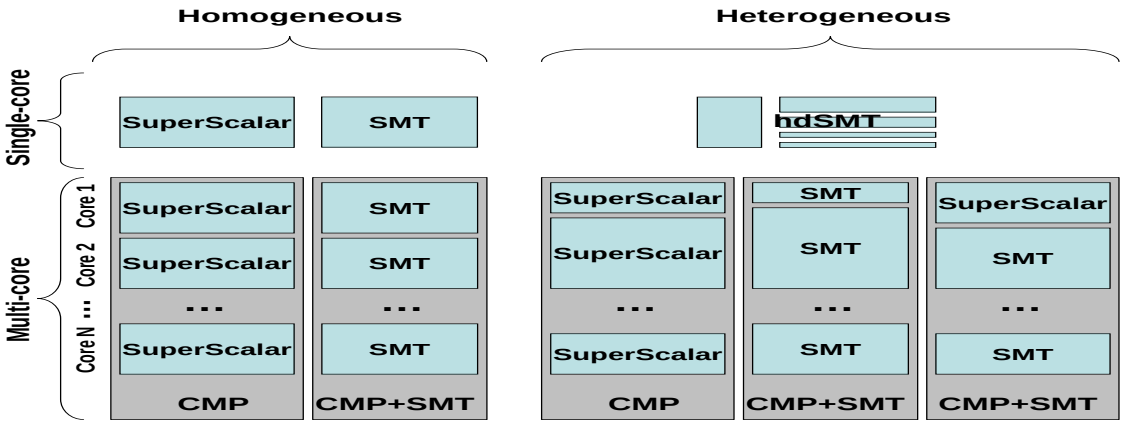


Figure 8.1: MPsim Processor Types.

cronologically stores all memory requests. *Wattch* [15] and *ALPSS* [40] represent extensions to both *Simplescalar* and *SMTsim*, respectively. They add power measurements to the functionality included in both simulators.

The *MPsim* is a highly-flexible simulator based on SMTsim. It allows simulating a wide range of processor types both single core (*Superscalar*, *SMT*) and multi core (*CMP*, *CMP+SMT*), both homogeneous and heterogeneous configurations; so as providing a complete set of simulation alternatives. It is put special emphasis on the simulator flexibility and how it is obtained. The *MPsim's Parameter Interface* allows to easily declare complex system configurations without needing to recompile the simulator's source code. Both core-specific and memory subsystem configuration parameters may be gathered into parameter files, comprising reusable configuration repositories. The simulation results included indicate that high-flexibility may be obtained without hardly compromising the computational cost in a general-purpose simulator.

8.1 MPsim overview

The *MPsim* is a cycle-accurate simulation tool based on the *SMTsim* [72] simulator. Its design focuses on the simulator's *flexibility* and *functionality*, striving at the same time to involve the least computational cost possible. The simulator's *flexibility* does not only refer to the amount of simulation alternatives provided to the user but also to the configuration *easiness* and *adaptability* to future modifications. The *MPsim's Parameter*

Interface ease the declaration of complex simulation configurations. It allows to maintain configuration file repositories that may be reused in different simulations without needing to recompile the simulator's source code.

The *MPsim* allows simulating a wide range of processor types both single core (*Superscalar*, *SMT*) and multicore (*CMP*, *CMP+SMT*). By using the *NUM_CORES* parameter it may be specified the number of cores in the simulated system. All the remainder core-specific parameters will carry the suffix *_Px*, where *x* stands for the core number (e.g., *IFETCH_POLICY_P1 ICOUNT* declares that the core number 1 use the ICOUNT IFetch Policy). These suffixes allow to individually configure each core, making possible *heterogeneous*[1] system configurations. Thus, although each simulated system core is comprised of at least 8 pipeline stages, the specific pipeline depth may be individually declared for each constituent system core. To configure entire systems, both *homogeneous* and *heterogeneous*, each simulated core may be individually declared by using both the command line or configuration files. The *MPsim's Parameter Interface* allows passing text files comprising all core-specific parameters. These configuration files may be reused in multiple declarations as simulation inputs to configure each simulated system core (e.g., *-pf_P1 POWER5* specifies the file POWER5 to configure the core number 1). Figure 8.1 shows the processor types that can be simulated using *MPsim*.

In order to reduce computational costs, the *MPsim* provides a trace-driven[2] front-end. Although *trace-driven*, the *MPsim* 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. The *MPsim* input traces are collected from the most representative 300 million instruction segment of each input benchmark, following the idea presented in [55]. 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. These input traces can be indistinctly read from little-endian/big-endian machines, since the *MPsim* automatically detects the machine characteristics and read data accordingly.

The *MPsim* functionality, provided to the user by means of its flexible *Parameter Interface*, includes a long list of simulation alternatives. Regarding simulation itself, the *MPsim* provides simulation forwarding, numerous simulation statistics and histograms, so as six different simulation finalization modes. Regarding computer architecture alternatives, the *MPsim* provides a set of branch predictors and instruction fetch policies from which select the desired one, thread migration between cores, so as multibanked

[1]The term *heterogeneous* refers to different amount of processor resources, like instruction queue entries and number of registers.

[2]The *execution-driven* functionality is currently being developed.

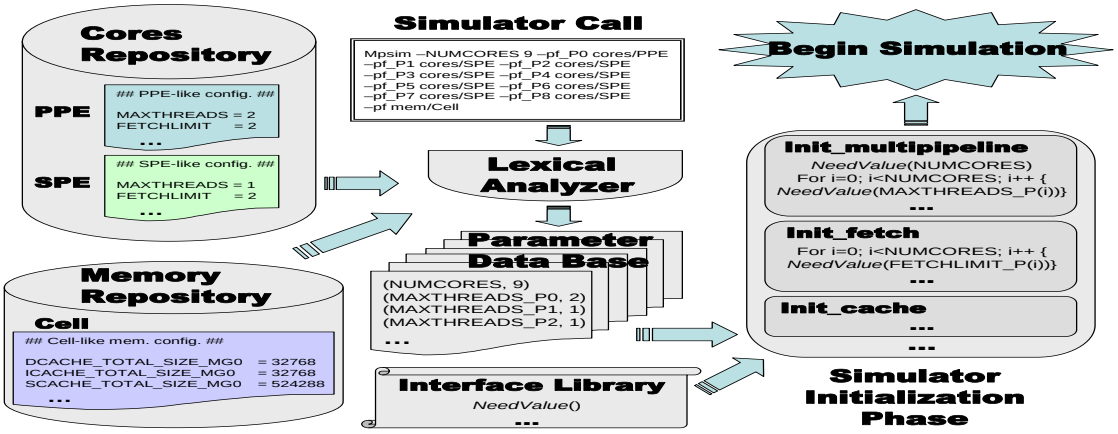


Figure 8.2: Parameter Interface Example for a Cell-like configuration.

multiported caches. All these functionality items may be easily activated/deactivated by the user, according to her needs, using the appropriate parameter for each case (e.g., *STATS_INTERVAL 0* deactivates the intermediate *IPC* statistics). As a matter of example, by means of the *STATS_INTERVAL*, *MAX_NUM_INTERVALS*, *STATS_FORWARDING* and *MAX_NUM_STATS_FILES* parameters it may be obtained intermediate simulation *IPC* statistics (interval *IPC*, *IPC* variability and in-flight L1 misses) in separate dump files.

The *MPsim* also allows some extent of *clustering* when definining the system to be modeled. Thus, the *SHARED_FETCH_UNIT* and *SHARED_REGISTER_FILE* parameters allow sharing a single Fetch Unit and Register File respectively, among all defined system cores. Since a single Fetch Unit may be shared among multiple cores, we indistinctly refer to pipeline/core in the remainder sections. However, recall that the only difference is the value of the *SHARED_FETCH_UNIT* (i.e., pipeline = true, core = false). As a matter of example, in an *hdSMT* [9](See *Chapter 3*) processor (see Figure 8.1) both the Fetch Unit and the Register File are shared among all constituent pipelines.

8.2 Parameter Interface

In order to provide high-flexibility the *MPsim* simulator includes a *lexical analyzer*, yielding a versatile *Parameter Interface*. It scans the simulator call creating pairs of parameter *name* and *value*, which are inserted in an inner *Parameter Data Base*. There is not a fixed parameter declaration order, with the only assumption that every argument which begins with a dash is considered a *parameter name* and the immediate following argument is considered its *value* (e.g., the simulator call *mpsim -arg1 arg2* includes the parameter *arg1* with value *arg2*). Whenever a single *parameter name* is declared more

than once, the value in the *Parameter Data Base* corresponds to the last parameter declaration. The *Parameter Interface Library* includes functions to adquire each parameter from the *Parameter Data Base* to the simulator inner structures. This way, the addition of new functionality benefits from an easy way to adquire configuration parameters.

The special *parameter name parms_file* (or simply *pf*) is reserved to indicate a configuration parameter file, with the *parameter value* indicating the file path. The use of parameter files permits to declare an unlimited number of parameters, allowing more complex simulation configurations. Additionally, by using parameter files, that may also include comments (using #), it is possible to keep *configuration file repositories*. Although the parameter files may include any sort of parameters, the main repositories used are comprised of *cores*, *machines* and *memory subsystems* declarations. In order to ease multicore configurations and repositories maintenance, it may be added the suffix *_Px* to a parameter file name declaration, with *x* identifying a given core. This suffix indicates that all the parameters included in the corresponding file are related to the specified *x* core (e.g., *-pf_P0 file1* declares the file *file1* as input to configure the first core in the simulated system). The *Parameter Interface* then automatically adds this suffix to each parameter name included in the file. Thus, a single core's parameter file may be used to configure multiple cores in a multicore configuration; or in different simulation calls.

Once scanned the whole simulator call, the resulting *Parameter Data Base*, that comprises all declared pairs of *parameter name* and *value*, is used in the subsequent *Simulator Initialization Phase*. During this phase the content of the *Parameter Data Base* is used to initialize the corresponding simulator structures and variables. Any sort of parameters may be requested by the simulator developer by using the *NeedValue* and *GiveValue* functions from the *Parameter Interface Library*. Whenever a parameter is compulsory, and does not admit a deffault value, it is used the *NeedValue*, which automatically stops the initialization phase and prompts an error message in absence of the specified parameter. Otherwise, it is used the *GiveValue* function.

Figure 8.2 illustrates the high-flexibility of the *MPsim Parameter Interface*. In the example, 3 configuration files stored in the simulator's repositories are used to configure a Cell-like processor with a simple simulator call. Given the files *PPE* and *SPE*, that include all core-specific configuration parameters for Cell PPE-like and SPE-like cores respectively, and the file *Cell*, that include all Memory Subsystem related parameters and relations for a Cell-like configuration, the simulator call shown in Figure 8.2 is enough to configure a Cell-like simulation[3].

[3]Although not included in the simulator call for simplicity, it should be also specified the workload to simulate.

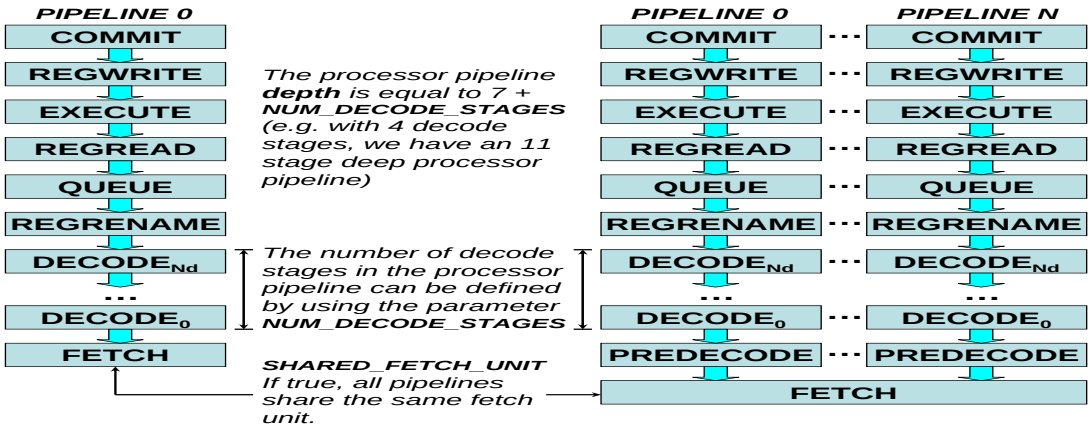


Figure 8.3: MPsim Processor Pipeline Stages.

The *Lexical Analyzer*, included in the *MPsim's Parameter Interface*, scans the whole simulator call shown in Figure 8.2 automatically accessing to the corresponding files in the repositories. The *Lexical Analyzer* uses the suffix information included in the simulator call (i.e., *_Px* in the *-pf_Px* argument, with *x* indicating the specific core) to create the corresponding pairs of *parameter name* and *value* that are inserted into the *Parameter Data Base*. Thus, although there is a single *MAXTHREADS* parameter declaration in PPE and SPE files stored in the cores repository (see Figure 8.2), multiple *MAXTHREADS* pairs are inserted in the *Parameter Data Base*, one per each of the 9 delcared cores. Once the whole simulator call is scanned, including the parameter files, the subsequent *Simulator Initialization Phase* uses the resulting *Parameter Data Base* and the *Interface Library* functions to set up the simulator inner structures and prepare the subsequent simulation. Thus, during the multipipeline environment initialization (i.e., *init_multipipeline*, see Figure 8.2) it is used the function *NeedValue* to initialize the simulator from the information contained in the *Parameter Data Base*, modeling an heterogeneous multi-core processor comprised of 9 cores (i.e., *NUMCORES*), each one containing *MAXTHREADS* hardware contexts (i.e., a dual-thread PPE and 8 single-thread SPEs). After the initialization phase, the simulation begins.

8.3 The Pipeline

The *MPsim* is a cycle-accurate simulator in which each simulated system core is comprised of at least 8 pipeline stages, as shown in Figure 8.3. However, each system core may be defined with a different *pipeline depth*, adding idle pipeline stages in between *Decode* and *RegRename* stages. As a matter of example, to specify an 11-stage execution pipeline in any of the declared cores it is set the parameter *NUM_DECODE_STAGES 4*.

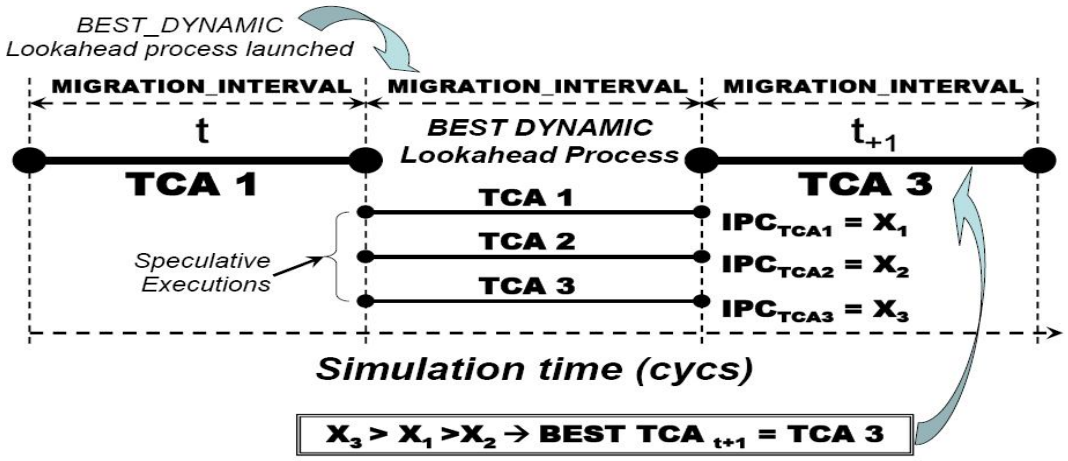


Figure 8.4: MPsim BEST DYNAMIC migration heuristic.

In case of sharing the Fetch Unit among all pipelines (see Section 8.1) a new pipeline stage, called *Predecode*, is automatically added by the *MPsim* to each pipeline. The *Predecode* stage works as a buffer (with user-definable capacity using the *PREDECODE_QUEUE_SIZE* parameter) between the shared Fetch Engine and the decode stage of each constituent pipeline, which may have a different pipeline width. As a matter of example, in a given cycle an 8-wide shared Fetch Engine passes 8 instructions to a 4-wide pipeline; 4 instructions passes to that pipeline decode stage while another 4 instructions are buffered in *Predecode* until the next simulation cycle.

The pipeline resources and implemented policies may be easily declared using the *MPsim Parameter Interface* (see Section 8.2). Each Fetch Unit declared in a simultaneous multithreaded system (i.e., the shared Fetch Unit in an *hdSMT* processor or each Fetch Unit in a *CMP+SMT* processor) may be configured with a different Instruction Fetch Policy, which determines from which thread/s to fetch instructions each cycle. To define the IFetch Policy used by each Fetch Unit we employ the *IFETCH_POLICY_Px* parameter, where *x* corresponds to the processor pipeline number. The user may select any from Round Robin [72], ICOUNT [72], STALL [70], FLUSH [70], and FLUSH_PLUS_PLUS [18]. In a similar way, each Fetch Unit declared in a system may be configured with a different branch predictor, using the *predictor_Px* parameter, where x corresponds to the processor pipeline number. In this case, the user may chose any from GSHARE [44], PERCEPTRON [34] and PERFECT predictors.

8.3.1 Thread Migration

Multicore configurations can be simulated in either *STATIC* or *DYNAMIC* fashion, using the *THREADS_MIGRATION* parameter. *STATIC* simulations assume no thread migrations, from core to core, during the whole simulation. *DYNAMIC* simulations may experience thread migrations according to the specified *MIGRATION_INTERVAL* parameter value (measured in simulated cycles). The assignment of all simulated threads to any of the defined cores is specified by the *FIRST_T2P_ASSIG_POLICY* parameter. It may be chosed from *NRR* (Naive Round Robin) and *CUSTOM* policies, using the *ASSIG_TH_X_P* parameter in the latter case to specify each assignment (e.g., *ASSIG_TH_1_P 0* assigns the thread 1 to the core 0).

In *DYNAMIC* simulations, thread migrations are triggered according to the specified *MIGRATION_HEURISTIC* parameter value. Among the available migration heuristics it can be chosed the *BEST_DYNAMIC*. As shown in Figure 8.4, every simulated interval (i.e., intervals *t* and *t+1*), with a fixed length of *MIGRATION_INTERVAL* cycles, is executed using the *BEST Thread to Core Assignment (TCA)* (See *Chapter 4*). This *BEST TCA* is obtained by means of a *Lookahead process*, that simulates in parallel the next simulation interval using each possible *TCA* (e.g., in the example shown in Figure 8.4 there are only three possible *TCAs*). Once determined the one that yields the highest throughput, the execution selects that *TCA* as the one to be used during the interval *t+1*.

8.4 The Memory Subsystem

The *MPsim* Memory Subsystem inherits the *SMTsim's* foundations, having an event queue to manage all memory requests in a non-deterministic fashion. Whenever a memory request experiences an L1 Cache miss it is inserted a memory request in this event queue, arranged by cronological request time (in simulated cycles). According to the specific system configuration, memory hit/miss and contention, the memory request may have to traverse the L2 Cache, the L1-L2 intercomunication bus, and the L3 Cache, so as accessing to a TLB. If all this fails, an access to main memory (off-chip) is assumed. The memory request queue is regularly accessed by the simulator, triggering each request in the corresponding simulation cycle. As described in Section 8.4.2, the *MPsim* structures this memory event queue into two layers for multicore configurations, implementing an *L2 Access Arbiter*.

Unlike *SMTsim*, with a *fixed* Memory Subsystem definition, the *MPsim* provides the user a *fully-flexible* Memory Subsystem. Thus, it may be configured a Memory Subsystem comprised of any number of memory components (DTLBs, ITLBs, DCaches,

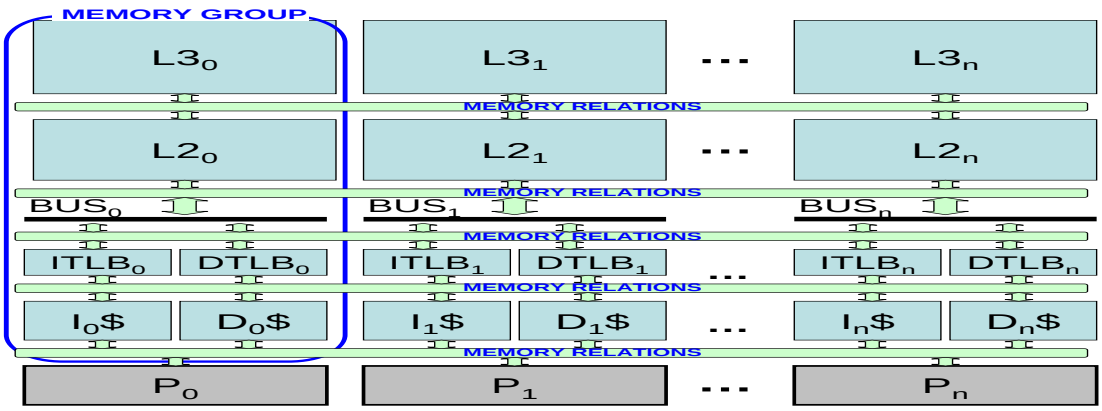


Figure 8.5: MPsim Memory Subsystem.

ICaches, L1-L2 Buses, L2 Caches and L3 Caches) so as relations, between memory components and execution pipelines. The *MPsim's Parameter Interface* allows to specify the desired number of components[4] by using the *NUM_L3_CACHES*, *NUM_L2_CACHES*, *NUM_BUSES*, *NUM_ICACHES*, *NUM_DCACHES*, *NUM_ITLBS*, *NUM_DTLBS* parameters. Once declared, the user may configure each of the components' characteristics individually, using command line parameters or parameter files (e.g., a DTLB is configured with *DTLBPENALTY*, *DPGSIZE* and *DTLB_SIZE* parameters). As a consequence, not all components of the same type must have the same characteristics, allowing *heterogeneous memory configurations*. To ease this configuration, each memory component is associated to a single *Memory Group (MG)*, as shown in Figure 8.5 (e.g., I_0\$, $ITLB_0$, D_0\$, $DTLB_0$, BUS_0, $L2_0$ and $L3_0$ belong to the first *Memory Group*). Thus, when specifying a component's characteristic we add the suffix *_MGx*, where *x* stands for the *Memory Group*, to refer to a particular memory component (e.g., the *DTLB_SIZE_MG0* parameter value specifies the size of the $DTLB_0$, belonging to the first *Memory Group*).

The *MPsim* Memory Subsystem does not assume any implicit relation between any two components[5], allowing the user to explicitly declare the desired relations. The *Memory Groups*, used to univocally refer to each memory component declared in the system, do not imply real memory component relations (i.e., D_0\$ does not necessarily use BUS_0 to communicate with the second level of cache). To specify the desired memory component relations the *MPsim Parameter Interface* provides a simple *Regular Expression Grammar (REG)*, shown in Figure 8.6. This *REG*, implemented as part of the *Lexical An-*

[4]There must be at least 1 declared component of each type except for L3 Caches, which are optional.

[5]Unless a single component of any type was declared (e.g., in a system with a single DCache all cores must access to that DCache). In that case the corresponding relations with other components are implicitly assumed.

REL_X_Y d

where

X = {P_da}
Y = {DC[_di], IC[_dj]}
d = {db,dc}

or

X = {DC_MGdb}
Y = {DTLB[_dk], BUS, L2, L3}
d = {dd,de,df,dg}

or

X = {IC_MGdc}
Y = {ITLB[_dl], BUS, L2, L3}
d = {de, df, dg, dh}

with

1 ≤ da < NUM_CORES
1 ≤ db < NUM_DCACHES
1 ≤ dc < NUM_ICACHES
1 ≤ dd < NUM_DTLBS
1 ≤ de < NUM_ITLBS
1 ≤ df < NUM_BUSES
1 ≤ dg < NUM_L2_CACHES
0 ≤ dh < NUM_L3_CACHES
* 0 ≤ di < Max. Core DCaches
* 0 ≤ dj < Max. Core ICaches
* 0 ≤ dk < Max. D$ DTLBs
* 0 ≤ dl < Max. I$ ITLBs

*** In case of Multi-Relations (N:M)**

Figure 8.6: MPsim Memory Relation Regular Expression Grammar.

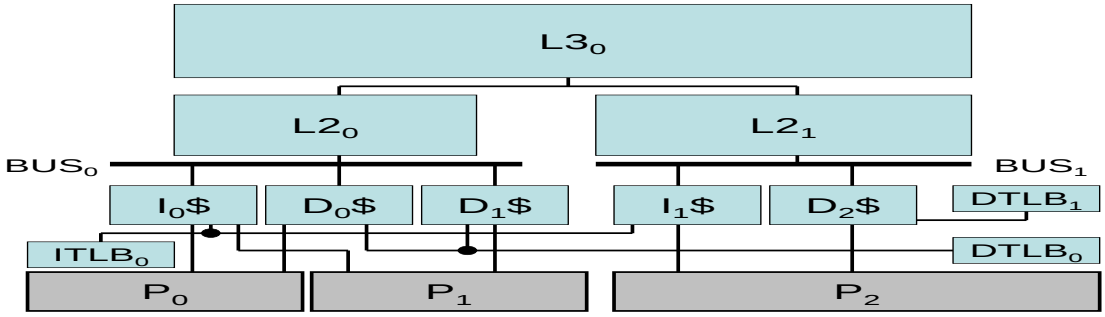


Figure 8.7: MPsim Memory Subsystem Example.

alyzer included in the *MPsim's Parameter Interface* (See Section 8.2), allows to establish a relation between any two memory components. These relations are focused on the first level of cache; the user specifies for each first level cache (i.e., D$ and I$) both the execution pipeline and the remainder memory components that are related with that specific component. The flexibility provided by this simple grammar allows to declare complex memory configurations, including *N:M* relations as is the case of first level caches and TLBs (i.e., a single Data Cache may use more than one DTLB).

As a matter of example, Figure 8.7 shows an example of a Memory Subsystem for a 3-core system. To specify all the constituent memory components shown in Figure 8.7 it should be used the following declaration:

-NUM_DCACHES 3 -NUM_ICACHES 2 -NUM_DTLBS 2 -NUM_ITLBS 1

-NUM_BUSES 2 -NUM_L2_CACHES 2 -NUM_L3_CACHES 1

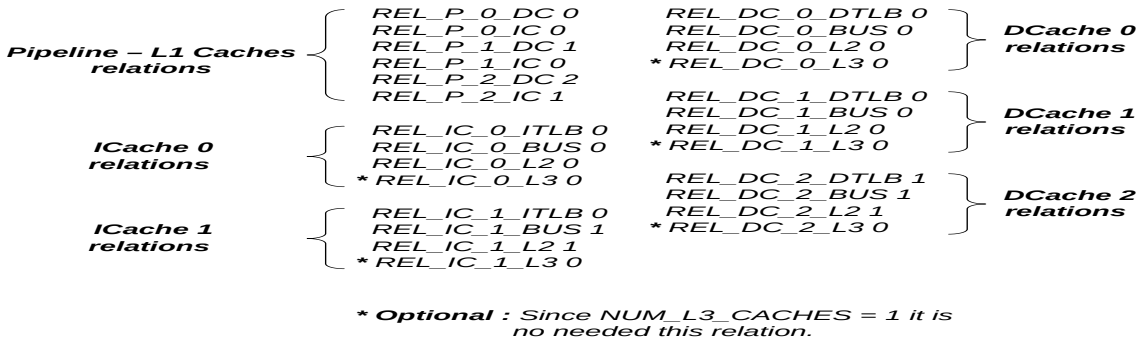


Figure 8.8: MPsim Memory Component Relations Example.

Once declared all the memory components, the relations between them are declared using the memory relation grammar shown in Figure 8.6, as depicted in Figure 8.8. For a Memory Subsystem to be fully declared, every first level cache (ICaches and DCaches) must be related with some pipeline (or multiple pipelines), TLB (or multiple), L1-L2 bus, L2 Cache and optionally with some L3 Cache. Finally, each memory component is configured using its specific parameters (e.g., *-DTLBPENALTY_MG1 300 -DPGSIZE_MG1 13 -DTLB_SIZE_MG1 512* configures the DTLB number 1 with 512 entries, a miss penalization of 300 cycles and a 8Kb virtual page size –2 to 13–). As with pipeline configuration, the *MPsim's Parameter Interface* allows to maintain a *Memory Subsystems & Relations Repository* (*memHierarchies* directory) and use them to declare more complex configurations. As a matter of example, let be *POWER5_MEM* and *POWER5_MEM_rels* the configuration files comprising all memory component configuration parameters and the relations between them, respectively, to configure a *POWER5-like* [13] Memory Subsystem. We would use the following declaration to fully configure a *POWER5-like* Memory Subsystem:

-pf memHierarchies/POWER5_MEM -pf memHierarchies/POWER5_MEM_rels

The complete functionality of the *MPsim* is deepthly explained in [8]. In the following sections we focus on two main issues of the Memory Subsystem's functionality. In Section 8.4.1 we describe the Multibanked and Multiported Cache functionality and the L2 Cache Access Arbiter in Section 8.4.2.

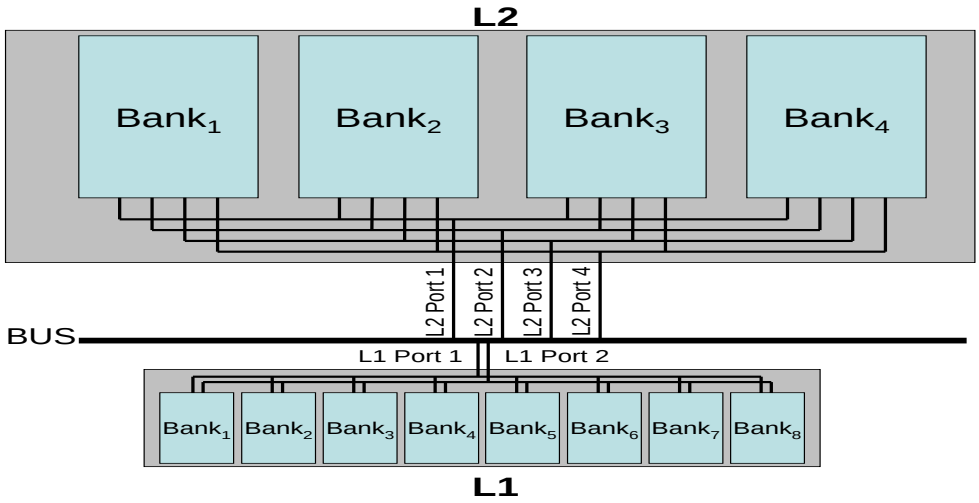


Figure 8.9: 4-bank 4-port L2 Cache and 8-bank 2-port L1 Cache Example.

8.4.1 Multibanked & Multiported Caches

For each cache declared in the Memory Subsystem, it is possible to specify the number of banks in which it will be splitted. The *MPsim'S Parameter Interface* provides this functionality by means of the *ICACHEBANKS_MGx*, *DCACHEBANKS_MGx*, *L2CACHEBANKS_MGx*, and *L3CACHEBANKS_MGx* parameters, where *x* stands for the specific *Memory Group*. Additionally, each cache may be configured with a different number of access ports, using the *NUM_DCACHE_PORTS_MGx*, *NUM_ICACHE_PORTS_MGx*, and *L2CACHEBANKPORT_MGx* parameters, where *x* stands for the specific *Memory Group*. As a matter of example, the following declaration configures a 4-bank 4-port L2 Cache and an 8-bank 2-port DCache, shown in Figure 8.9 :

-NUM_DCACHES 1 -DCACHEBANKS_MG0 8 -NUM_DCACHE_PORTS_MG0 2

-NUM_L2_CACHES 1 -L2CACHEBANKS 4 -L2CACHEBANKPORTS_MG0 4

8.4.2 L2 Cache Access Arbiter

The *MPsim* allows defining multicore system configurations in which many cores may share a single L2 Cache. In order to cope with the L2 Cache contention among all cores the *MPsim* provides an *L2 Cache Access Arbiter*, that can be activated using the *L2_ACC_ARBITER* parameter. The *MPsim's L2 Cache Access Arbiter*, shown in Figure 8.10, manages the access to each L2 Cache bank using a queue per each shared core of

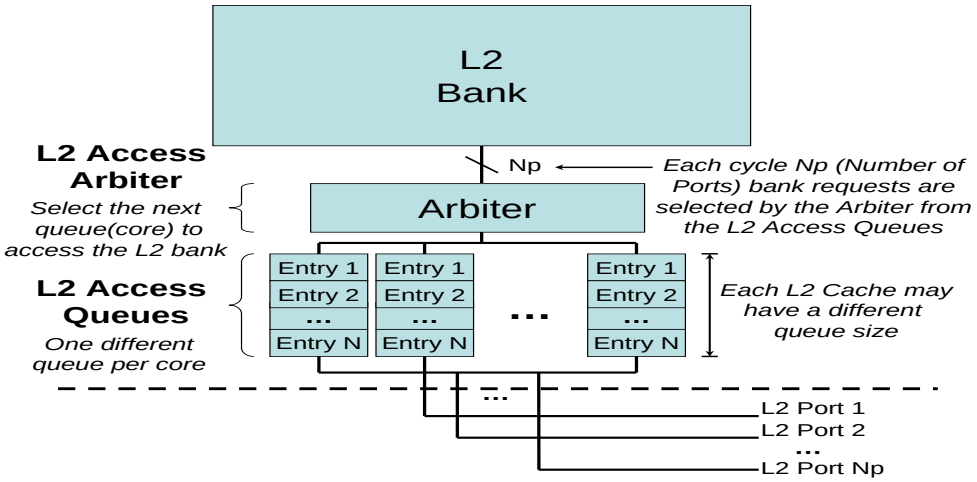


Figure 8.10: L2 Cache Access Arbiter.

each defined L2 Cache. Each of these queues buffer the core's *L2 Cache access requests* until the user-definable *L2 Arbiter* removes it from the corresponding queue and triggers the L2 Cache Bank access; as many requests allowed per simulated cycle as L2 Cache ports defined in the Memory Subsystem declaration. Whenever an L2 Access Queue gets full the corresponding core is temporarily stopped (no forward progress in any pipeline stage) until some queue entry gets empty.

8.5 Power Measurement

The *MPsim's* Power Consumption Measurement[6] is modelled based on *Wattch 1.02* [15]. Before running the simulation, the *MPsim* calculates the energy that each basic component of the processor , according to the system configuration declared, would consume in one cycle if it were fully utilized. Each basic unit is defined as one of four types (array, CAM, functional unit, clock) and different estimation formulas are used for each type. The *Cacti 4.2* [68] is used to provide the optimal specs for each array and CAM unit based on the required total size, block size and associativity. The *MPsim* keeps track of the number of times each unit is accessed during the simulation and calculates the energy consumed accordingly. The total energy consumed by each component is shown at the end of the simulation, along with average power per cycle.

[6]The current version only models the power consumption within each execution core. Interconnections and memory related consumption will be included in subsequent *MPsim* versions.

8.6 Computational Cost

Although high-flexibility constitutes a very important characteristic for a general-purpose simulator it may not be achieved regardless its computational cost. Due to finite computational resources, computer architects require simulation tools that are able to yield results in a limited amount of time, according to reseach deadlines. It must be kept in mind that the results obtained from such a simulation tool generally constitute a first step in a multistep evaluation process. Due to their limited accuracy, general-purpose simulators are normally used to both identify architectural challenges and obtain general trends. In this sense, the flexibility offered by the selected simulation tool is of crucial importance. The tool's ability to allow simulating multiple architectural alternatives with low effort helps to both accelerate and improve this first evaluation step. Once identified the architectural challenge and evaluated a possible solution, more accurate results may be obtained employing either a more specific (and complex) simulator or FPGAs [78].

Although focused on high-flexibility, the *MPsim* design strives to involve the least computational cost possible. The idea is to provide a flexible and easy-to-use simulation tool, that allows the user to simulate a wide range of simulation alternatives with low effort, able to yield simulation results in a reasonable time. Although these goals seem a priori to conflict with each other, it may be found a satisfactory balance. Following are enumerated some of the main design decisions employed in the *MPsim* development:

1. *Parameter Interface*. Providing high-flexibility should not interfere with the simulation itself. The parameter interface should be adaptable to accommodate future simulation improvements but it should not interfere with the inner simulation structures.

2. *Initialization Phase*. The configuration parameters acquisition, performed by means of a flexible and easy-to-use *Parameter Interface*, should be immediately followed by an *Initialization Phase*. During this preparatory phase it should be anticipated all the work possible according to the simulation configuration. Thus, while some simulator modules could be fully deactivated, without compromosing neither memory nor processing in the subsequent simulation, others could be devoted enough memory to get rid of time consuming dynamic memory allocation/deallocation. According to the specific simulation configuration and the available resources, the *Initialization Phase* may considerable reduce the subsequent simulation computational cost.

3. *Avoid unnecessary work*. Instead of requiring function calls to determine whether a module is activated or not during the simulation, each module may include *macros*.

Without compromising neither the code legibility nor modularity, a macro including a conditional branch to the corresponding function call may reduce the additional cost for deactivated modules; adding only an extra conditional branch for activated ones. Furthermore, since modules are activated/deactivated only durign the *Initialization Phase*, these branches are easily predictable by the branch predictor implemented on the hardware executing the simulator itself.

In order to give an idea of the computational cost involved by the *MPsim* simulation tool, following are included some simulation results. For this set of experiments we use the *SPEC2000 Benchmark Suite* (See Section 2.2). We collected workloads comprised of 1, 2, 4 and 8 benchmarks, shown in Table 8.1. Following we gather some comments regarding the simulation parameters shown in Table 8.1:

1. The 22-cycle L1 misspenalty comes from the sum of L1 latency (3) plus the L2 latency (15) plus the L1-L2 bus trasnfer (2) plus the DFill Delay (2).

2. Both Instruction Cache (I-Cache) and Data Cache (D-Cache) follow an implementation of the *Least Recently Used (LRU)* replacement policy and *Write Back* write policy. Since current *MPsim* version does not allow multithreaded workloads, no memory coherence protocols are present.

3. Since the Memory Wall problem seems to still be problematic in the short and medium term, a conservative 250-cycle main memory latency is used.

4. It is used a private TLB for both instructions (Instruction TLB, I-TLB) and data (Data TLB, D-TLB). Whenever a TLB miss arises, it must be accessed to the main memory to resolve the new page address and bring it back to the corresponding TLB. Consequently, it must be paid the main memory latency (of 250 cycles) plus the TLB resolution itself (50 cycles).

The name of each workload[7] is *xWy*, where *x* stands for the number of threads involved and *y* stands for the workload identifier (e.g., *4W2* identifies the second workload with 4 threads). Each workload with size *x* is simulated on a *CMP+SMT* implementation with shared L2 Cache and $\frac{x}{2}$ two-hardware-context *SMT* cores implementing *ICOUNT* [72] policy; both single-thread and dual-thread workloads are simulated on a single-core implementation. Both core-specific and memory subsystem configuration parameters are shown in Table 8.1.

[7]Except for single-thread workloads, represented by the name of the corresponding benchmark.

Simulation Parameters			
Pipeline depth	11 stages	L1 I-Cache	64KB, 4-way, 8 banks
Queues Entries	64 int, 64 fp, 64 ld/st	L1 D-Cache	32KB, 4-way, 8 banks
Execution Units	4 int, 3 fp, 2 ld/st	L1 lat./miss	3/22 cycs.
Physical Registers	320 regs.	I-TLB ,D-TLB	512 ent. Full-assoc.
ROB Size*	256 entries	TLB miss	300 cycs.
Branch Predictor	perceptron (4K local, 256 pers)	L2 Cache L2 latency	4MB, 12-way, 4 banks 15 cycs.
BTB	256 entries, 4-way associative	M. Memory lat.	250 cycs.
RAS*	100 entries		

	Number of Threads		
Name	2	4	8
xW1	b, j	b, q, t, j	d, l, b, g, i, j, c, f
xW2	n, e	l, n, p, e	b, g, m, n, a, h, o, p
xW3	d, a	d, s, r, a	m, n, r, q, i, j, e, h
xW4	g, f	g, b, m, f	l, b, g, m, n, r, f, s
xW5	r, p	r, j, f, p	q, b, c, k, e, a, o, t
xW6	b, j	b, q, t, j	d, l, b, g, i, j, c, f
xW7	n, e	l, n, p, e	b, g, m, n, a, h, o, p
xW8	d, a	d, s, r, a	m, n, r, q, i, j, e, h
xW9	g, f	g, b, m, f	l, b, g, m, n, r, f, s
xW10	r, p	r, j, f, p	q, b, c, k, e, a, o, t

gzip	a	swim	n
vpr	b	apsi	o
gcc	c	wupwise	p
mcf	d	equake	q
crafty	e	lucas	r
perlbmk	f	mesa	s
parser	g	fma3d	t
eon	h	sixtrack	u
gap	i	facerec	v
vortex	j	applu	w
bzip2	k	galgel	x
twolf	l	ammp	y
art	m	mgrid	z

Table 8.1: Workloads used for evaluating the computational cost. Resources with * follow a private per-thread implementation.

All workloads were simulated on a Dual-Core 2 Intel Xeon processor with 2,333GHz, 1.333MHz FSB, and 4MB cache running Linux 2.6.15. Figures 8.11, 8.12, 8.13, and 8.14, show the time required to simulate each workload until any of the comprising benchmarks finish simulating 300 million instructions. Except for the *181.mcf*, with a pathological bad memory behavior due to nested memory references, all single-thread workloads are fully simulated in about twelve minutes time, which constitutes a reasonably low computational cost. As could be a priori expected, doubling the number of benchmarks in the workload (i.e., dual-thread workloads *–2Wy–*) doubles the required simulation time, as shown in Figure 8.12. Adding more dual-thread *SMT* cores, and consequently simultaenously simulating[8] more benchmarks, increases the required simulation time as shown in Figures 8.13 and 8.14.

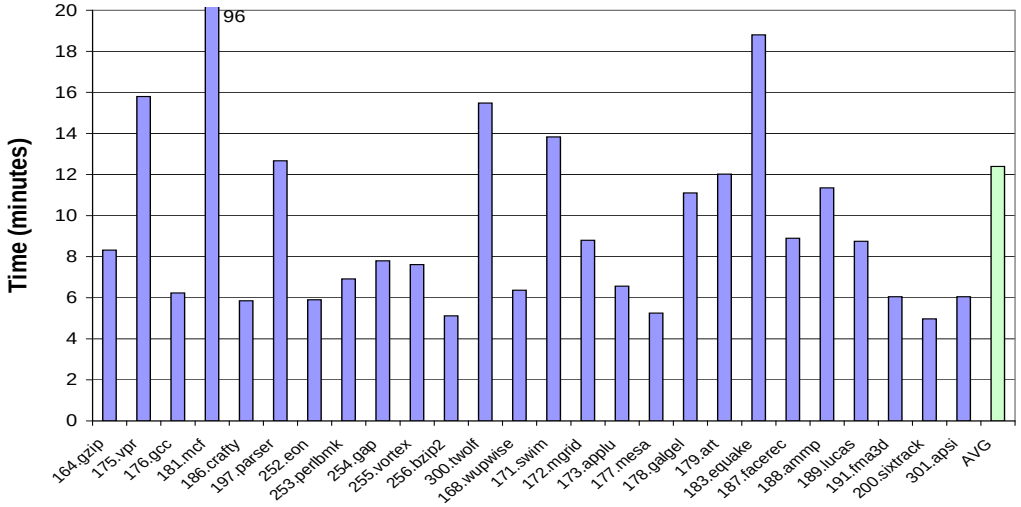


Figure 8.11: Single-Core Single-Thread Simulation Cost.

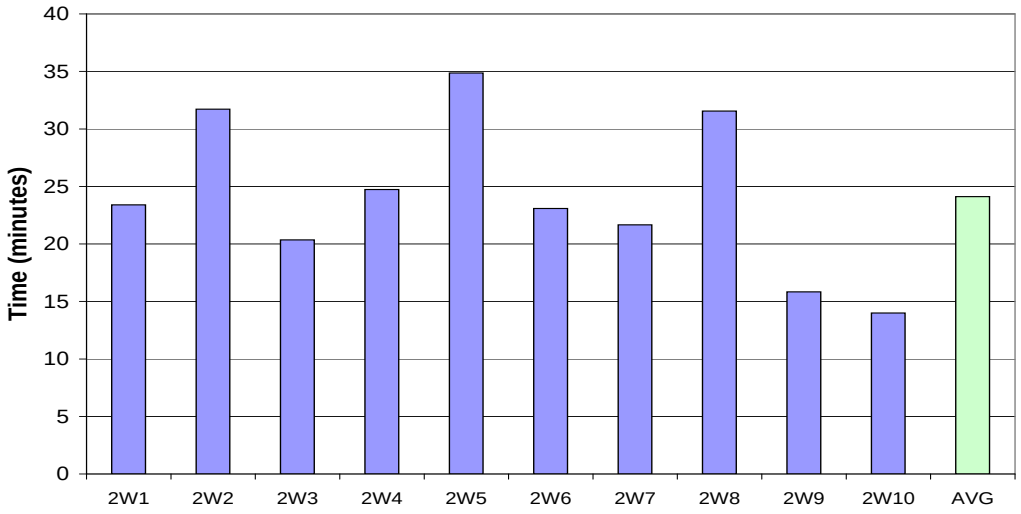


Figure 8.12: Single-Core Dual-Thread Simulation Cost.

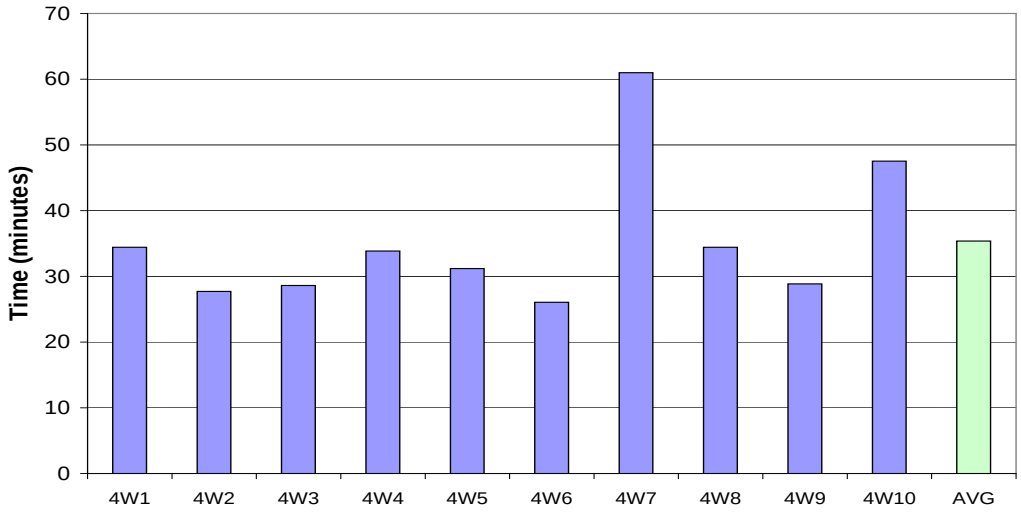


Figure 8.13: Dual-Core Dual-Thread Simulation Cost.

8.7 Conclusions & Future Work

The *MPsim* is a highly-flexible general-purpose simulation tool. It constitutes a cycle-accurate multi-purpose simulation tool which allows simulating a wide range of processor types. Both *single core* (*Superscalar*, *SMT*) and *multi core* (*CMP*, *CMP+SMT*), so as *homogeneous* and *heterogeneous* configurations, are available and may be employed by means of its flexible *Parameter Interface*.

The *MPsim* has been developed, and keeps on evolving, to constitute a simulation tool to assist computer architecture research in a wide range of scenarios. The programming phylosophy employed in the development of the *MPsim* favors *high-flexibility*, without critically compromising computational cost, so as new ideas could be easily added to the simulation tool's functionality. The simulation results included confirm that high-flexibility may be provided in a general-purpose simulator without hardly compromising its computational cost.

[8]Since the *MPsim* is designed using sequential code, as done by the *SMTsim*, increasing the size of the workload linearly increases the simulation cost. A parallel programmed version of the *MPsim* simulator is currently being developed.

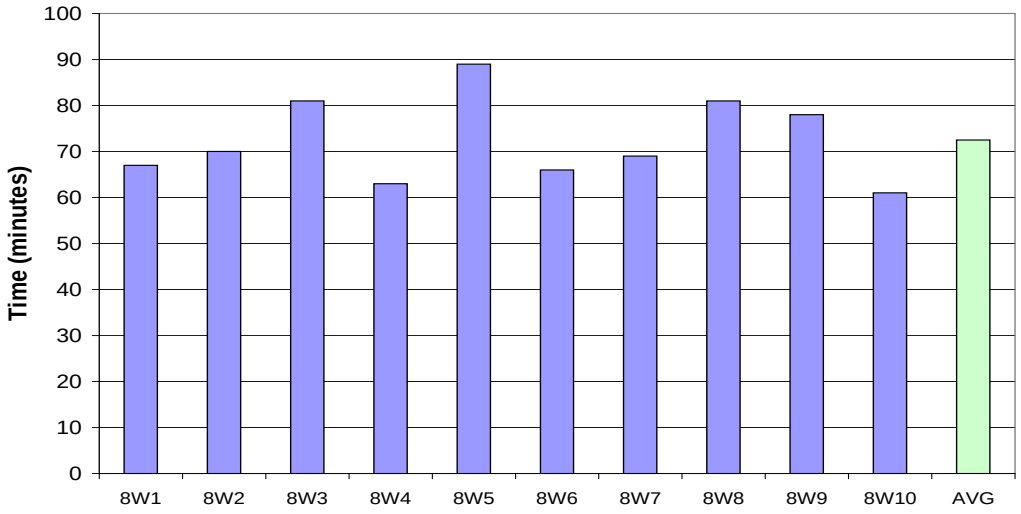


Figure 8.14: Quad-Core Dual-Thread Simulation Cost.

The *MPsim* simulator is already being used as simulation tool by *DAC* and *BSC* members. The different researchs in which is getting involved have made it evolve, yielding additional functionality to the one described herein. Among others, this additional functionality added to the *MPsim* simulator includes *multi Instruction Set Architecture (multiISA)* simulation (ALPHA & PowerPC). The *MPsim community* is growing step by step, using the *grup-mpsim@ac.upc.edu* distribution list as meeting point.

8.7.1 Further Considerations and Acknowledgements

The *MPsim* simulator is the product of a combinated effort that started long before the beginning of this PhD dissertation. The original idea of developing such a simulation tool comes from *Daniel Ortega*, who also envisioned and developed the main functions of the library upon which it is built the *MPsim's Parameter Interface*. Some of the *MPsim's single-thread* functionality comes from *Ayose Falcon* and *Oliverio Santana* (fetch engine & branch prediction). In the *SMT* field, *Francisco J. Cazorla*'s contributions (IFetch Policies & some of the Simulation Finalization Methodologies) to *MPsim* were of a crucial importance. The *Power Measurement* functionality has its origins in the work done by *Domen Novak* at *BSC*. Others researchers as *Jeroen Vermoulen*, *Miquel Moreto* and *Jose C. Ruiz* have contributed to increase the *MPsim's* robustness.

With all these contributions, the task of developping a highly-flexible simulation tool, that would include all this stuff in a computationally and reasonable way, has comprised a great challenge. It firstly was developped the *MPsim's Parameter Interface* using the functions developped by *Daniel Ortega*, and adding some more to complete the interface library. Then, the whole simulator had to be redesigned to create a centralized *Parameter Data Base* from which all simulator parameters would be obtained[9] With such a *Parameter Data Base* available, each of the simulator components (execution pipeline stages, memory, etc) was added a initialization phase, overall orchestrating the *Simulator Initialization Phase*. Using the *MPsim's Parameter Interface* as a general input for all new simulation functionality, multicore staff was added (fully configurable multicore configurations). To support covering unusual processor layouts, the whole memory subsystem was revisited so that the user could specify how many components does he want to use in each configuration, so as the specific parameters for each of these components. A regular expression grammar was added to the *MPsim's Parameter Interface* to allow high-flexibility when specifying the connections between each of these components. Other simulation functionality such as dynamic thread migration (between execution cores), L2 cache access arbiter, shared fetch engine or pipeline depth specification were added, among others, also using the *MPsim's Parameter Interface* to adquire the required simulation parameters.

Regarding ongoing and future work, *Miquel Moreto* is responsible for the *multiISA MPsim* implementation which seems a very promising simulation functionality to cover the incoming *Heterogeneous Processor Generations* with multiple *ISAs* on a single chip, like the *Cell* Processor (*PowerPC* and *SIMD,Altivec*).

Thank you very much indeed to *Daniel Ortega*, *Ayose Falcon*, *Oliverio Santana*, *Francisco J. Cazorla*, *Domen Novak*, *Jeroen Vermoulen*, *Jose C. Ruiz* and *Miquel Moreto* for all their contributions to the *MPsim* simulation tool.

[9] So far it was done using headers.c and having to recompile the whole simulator. The amount of variable parameters that could be provided using the simulator input arguments was quite limited.

Publications

International Conferences

- *A Complexity-Effective Simultaneous Multithreading Architecture*. Carmelo Acosta, Ayose Falcon, Alex Ramirez and Mateo Valero. *International Conference On Parallel Processing (ICPP-05)*, Oslo (Norway). June 2005.
- *Core to Memory Interconnection Implications for Forthcoming On-Chip Multiprocessors*. Carmelo Acosta, Francisco J. Cazorla, Alex Ramirez and Mateo Valero. *1st Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*. Phoenix, USA. February 2007.
- *MFLUSH: Handling Long-latency loads in SMT On-Chip Multiprocessors*. Carmelo Acosta, Francisco J. Cazorla, Alex Ramirez and Mateo Valero. *International Conference On Parallel Processing (ICPP-08)*, Portland (USA). September 2008.

National Conferences

- *Heterogeneity-Aware Architectures*. Carmelo Acosta, Ayose Falcon, Alex Ramirez and Mateo Valero. *XV Jornadas de Paralelismo*. Almera (Spain). September 2004.
- *hdSMT: An Heterogeneity-Aware Simultaneous Multithreading Architecture*. Carmelo Acosta, Ayose Falcn, Alex Ramirez and Mateo Valero. *XVI Jornadas de Paralelismo*. Granada (Spain). September 2005.

Others

- *Complexity-Effectiveness in Multithreading Architectures*. Carmelo Acosta, Ayose Falcon, Alex Ramirez and Mateo Valero. *ACACES 2005*, L'Aquila (Italy). August 2005.

Submitted Papers

- *Thread to Core Assignment in SMT On-Chip Multiprocessors*. Carmelo Acosta, Francisco J. Cazorla, Alex Ramirez and Mateo Valero. *Submitted to SBAC - PAD 2009*.
- *hTCA: An OS-driven Framework for Complexity-Effectiveness in SMT On-Chip Multiprocessors*. Carmelo Acosta, Francisco J. Cazorla, Alex Ramirez and Mateo Valero. *Pending due to dependences with unpublished paper*.

Technical Reports

- *A First Glance at a Heterogeneity-Aware Simultaneous Multithreading Architecture*. Carmelo Acosta, Ayose Falcon, Alex Ramrez and Mateo Valero. *UPC-DAC-2004-23*, June 2004.
- *Maximizing Multithreaded Multicore Architectures through Thread Migrations*. Carmelo Acosta, Ayose Falcon, Alex Ramrez and Mateo Valero. *UPC-DAC-RR-CAP-2009-2*, January 2009.
- *The MPsim Simulation Tool*. Carmelo Acosta, Francisco J. Cazorla, Alex Ramrez and Mateo Valero. *UPC-DAC-RR-CAP-2009-15*, March 2009.

List of Figures

List of Tables

References

[1] http://researchweb.watson.ibm.com/journal/rd/446/borkenhagen.txt.

[2] http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_10220_10221%5E964,00.html.

[3] http://www.intel.com/design/intarch/pentiumiii/pentiumiii.htm.

[4] UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007.

[5] Desktop Performance and Optimization for Intel Pentium 4 Processor – Intel White Paper.

[6] Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor. *White Paper* (2004).

[7] UltraSPARC T1 Supplement. *Draft D2.0, 17 Mar* (2006).

[8] ACOSTA, C., CAZORLA, F., RAMIREZ, A., AND VALERO, M. The MPsim complete handbook. Tech. Rep. UPC-DAC2009, 2009.

[9] ACOSTA, C., FALCÓN, A., RAMÍREZ, A., AND VALERO, M. A Complexity-Effective Simultaneous Multithreading Architecture. In *Proc. of ICPP-35* (2005).

[10] AGARWAL, A., LIM, B. H., KRANZ, D., AND KUBIATOWICZ, J. APRIL: A processor architecture for multiprocessing. Tech. Rep. MIT/LCS/TM-450, 1991.

[11] ALPERT, D. Will microprocessor become simpler? Microprocessor Report, Nov. 2003.

[12] ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. The Tera computer system, 1990.

[13] B. SINHAROY, R. N. KALLA, J. M. T. R. J. E., AND JOYNER, J. B. POWER5 system microarchitecture. *IBM Journal of Research and Development. 49(4/5):505–52* (2005).

[14] BOVET, D. P., AND CESATI, M. Understanding the Linux Kernel - Third Edition.

[15] BROOKS, D., TIWARI, V., AND MARTONOSI, M. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA* (2000), pp. 83–94.

[16] BURNS, J., AND GAUDIOT, J. Quantifying the SMT Layout Overhead - Does SMT Pull its Weight? In *Proceedings of the 6th International Conference on High Performance Computer Architecture* (2000), pp. 109–120.

[17] BURNS, J., AND GAUDIOT, J. SMT Layout Overhead and Scalability. *IEEE Transactions on Parallel and Distributed Systems 13*, 2

(Feb. 2002), 142 – 155.

[18] CAZORLA, F. J., FERNÁNDEZ, E., RAMIREZ, A., AND VALERO, M. Improving Memory latency aware fetch policies for SMT processors. In *Proc. of ISHPC-V* (2003).

[19] CAZORLA, F. J., FERNÁNDEZ, E., RAMIREZ, A., AND VALERO, M. DCache Warn: An I-Fetch policy to increase SMT efficiency. In *Proc. of IPDPS-18* (2004), pp. 24–34.

[20] CAZORLA, F. J., FERNÁNDEZ, E., RAMIREZ, A., AND VALERO, M. Dynamically Controlled Resource Allocation in SMT Processors. In *Proceedings of 37th Annual ACM/IEEE International Symposium on Microarchitecture* (2004).

[21] COLLINS, J. D., AND TULLSEN, D. M. Clustered multithreaded architectures – Pursuing both IPC and cycle time. In *Proc. of IPDPS-18* (2004).

[22] DESIKAN, R., ERNST, D., GUTHAUS, M., KIM, N., LARSON, E., MUDGE, T., MURUKATHAMPOONDI, H., SANKARALINGAM, K., YODER, B., AUSTIN, T., AND BURGER, D. SimpleScalar Version 4.0 Release. *held in conjunction with MICRO-34* (Dec. 2001).

[23] DHODAPKAR, A., AND SMITH, J. Comparing Program Phase Detection Techniques. In *Proceedings of 36th Annual ACM/IEEE International Symposium on Microarchitecture* (2003).

[24] EL-MOURSY, A., AND ALBONESI, D. H. Front-End Policies for Improved Issue Efficiency in SMT Processors. In *Proceedings of the 9th International Conference on High Performance Computer Architecture* (2003), pp. 65 – 76.

[25] EYERMAN, S., AND EECKHOUT, L. A Memory-Level Parallelism Aware Fetch Policy for SMT Processors. In *Proceedings of the 13th International Conference on High Performance Computer Architecture* (2007).

[26] FOLEGNANI, D., AND GONZALEZ, A. Energy-Effective Issue Logic. In *Proceedings of 28th Annual International Symposium on Computer Architecture* (2001).

[27] GSCHWIND, M., HOFSTEE, H. P., FLACHS, B., HOPKINS, M., WATANABE, Y., AND YAMAZAKI, T. Synergistic processing in cells multicore architecture. *IEEE Micro 26*, 2 (Mar. 2006), 10–24.

[28] HALSTEAD, R., AND FUJITA, T. MASA: A multithreaded processor architecture for parallel symbolic computing, May 1988.

[29] HAMMOND, L., NAYFEH, B. A., AND OLUKOTUN, K. Single-chip multiprocessor. In *IEEE Computer Special Issue on Billion-*

Transistor Processors (1997).

[30] HIRATA, H., KIMURA, K., NAGAMINE, S., MOCHIZUKI, Y., NISHIMURA, A., NAKASE, Y., AND NISHIZAWA, T. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (May 1992), pp. 136 – 145.

[31] HWU, W. W., AND PATT, Y. N. HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality. In *25 Years ISCA: Retrospectives and Reprints* (1998), pp. 300–308.

[32] J. VERA, F. J. CAZORLA, A. PAJUELO, O. J. SANTANA, E. FERNANDEZ, AND M. VALERO. A novel evaluation methodology to obtain fair measurements in multithreaded architectures. *Proc. of the 2nd. Workshop on Modeling, Benchmarking and Simulation* (2006).

[33] JAIN, R., HUGHES, C. J., AND ADVE, S. V. Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. In *Proc. of 23th Intl. Real-Time Systems Symposium* (2002).

[34] JIMNEZ, D., AND LIN, C. Dynamic Branch Prediction with Perceptrons. In *Proceedings of the 7th International Conference on High Performance Computer Architecture* (2001), pp. 197–206.

[35] KANELLOS, M. Moore's law to roll on for another decade. *CNET News.com* (Feb. 2003).

[36] KUEHN, J. T., AND SMITH, B. J. The horizon supercomputing system: Architecture and software, Nov. 1988.

[37] KUMAR, R., FARKAS, K., JOUPPI, N. P., RANGANATHAN, P., AND TULLSEN, D. M. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of 36th Annual ACM/IEEE International Symposium on Microarchitecture* (2003).

[38] KUMAR, R., TULLSEN, D. M., RANGANATHAN, P., JOUPPI, N. P., AND FARKAS, K. I. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of 31st Annual International Symposium on Computer Architecture* (2004).

[39] LE, H., STARKE, W., FIELDS, J., O'CONNELL, F., NGUYEN, D., RONCHETTI, B., SAUER, W., SCHWARZ, E., AND VADEN, M. IBM POWER6 microarchitecture. *IBM J. Res. Dev. 51*, 6 (2007), 639–662.

[40] LEE, S., AND GANDIOT, J.-L. ALPSS: architectural level power simulator for simultaneous multithreading, version 1.0. Tech. Rep. CENG-02- 04, 2002.

[41] LEVY, M. Multithreaded technologies disclosed at MPF. Micropro-

cessor Report, Nov. 2003.

[42] LIMOUSIN, C., SEBOT, J., VARTANIAN, A., AND DRACH-TEMAM, N. Improving 3d geometry transformations on a simultaneous multithreaded simd processor. In *Proceedings of the 15th International Conference on Supercomputing* (May 2001).

[43] M. J. SERRANO AND R. WOOD AND M. NEMIROVSKY. A Study on Multistreamed Superscalar Processors. Tech. Rep. 93-05, University of California Santa Barbara, 1993.

[44] MCFARLING, S. Combining Branch Predictors. Tech. Rep. WRL-TN-36, 1993.

[45] MOORE, G. E. Cramming more components onto integrated circuits. *Electronics Magazine*.

[46] MOORE, G. E. Excerpts from A Conversation with Gordon Moore: Moores Law. *Intel Corporation - White Paper*.

[47] MULVIHILL, D., AND ALLEN, M. Evaluating branch predictors on an SMT processor. Tech. Rep. CS 752, University of WisconsinMadison, 2002.

[48] OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K. The case for a single-chip multiprocessor. In *Proc. of ASPLOS-7* (1996).

[49] PHAM, D., ASANO, S., BOLLIGER, M., DAY, M. N., HOFSTEE, H. P., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI, Y., RILEY, M., SHIPPY, D., STASIAK, D., SUZUOKI, M., WANG, M., WARNOCK, J., WEITZEL, S., WENDEL, D., YAMAZAKI, T., AND YAZAWA, K. The Design and Implementation of a First-Generation CELL Processor. In *In Intl. Solid-State Circuits Conference Digest of Technical Papers* (2005).

[50] PYREDDY, R., AND TYSON, G. Evaluating design tradeoffs in dual speed pipelines. In *Proc. of WCED-2* (2001).

[51] RAASCH, S. E., AND REINHARDT, S. K. The Impact of Resource Partitioning on SMT Processors. In *Proc. of PACT-12* (2003).

[52] RAMIREZ, A., SANTANA, O., LARRIBA-PEY, J., AND VALERO, M. Fetching Instruction Streams. In *Proceedings of 35th Annual ACM/IEEE International Symposium on Microarchitecture* (2002).

[53] RAMSAY, M., FEUCHT, C., AND LIPASTI, M. H. Exploring efficient smt branch predictor design. In *Proceedings of the 2003 Workshop on Complexity Effective Design* (2003).

[54] SEZNEC, A., FELIX, S., KRISHNAN, V., AND SAZEIDES, Y. Design trade-offs on the EV8 branch predictor. In *Proceedings of 29th*

Annual International Symposium on Computer Architecture (2002).

[55] SHERWOOD, T., PERELMAN, E., AND CALDER, B. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Proc. of PACT-10* (2001).

[56] SHERWOOD, T., PERELMAN, E., HAMERLY, G., SAIR, S., AND CALDER, B. Discovering and Exploiting Program Phases. *IEEE Micro 23*, 6 (Nov. 2003), 84–93.

[57] SHERWOOD, T., SAIR, S., AND CALDER, B. Phase Tracking and Prediction. In *Proceedings of 30th Annual International Symposium on Computer Architecture* (2003).

[58] SHIN, C., LEE, S.-W., AND GAUDIOT, J.-L. Dynamic scheduling issues in SMT architectures. In *Proc. of IPDPS-17* (2003).

[59] SIGMUND, U., STEINHAUS, M., AND UNGERER, T. On Performance, Transistor Count and Chip Space Assessment of Multimedia-enhanced Simultaneous Multithreaded Processors. In *Proc. of MTEAC-4* (2000).

[60] SINHAROY, B., KALLA, R. N., TENDLER, J. M., EICKEMEYER, R. J., AND JOYNER, J. B. POWER5 System microarchitecture. *IBM J. Res. Dev. 49*, 4/5 (2005), 505–521.

[61] SMITH, B. Architecture and applications of the hep multiprocessor computer system. In *Proceedings of the Fourth Symposium on Real Time Signal Processing* (1981), pp. 241 – 249.

[62] SMITH, J. E., AND PLESZKUN, A. R. Implementation of Precise Interrupts in Pipelined Processors. In *ISCA* (1985), pp. 36–44.

[63] SNAVELY, A., TULLSEN, D., AND VOELKER, G. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In *SIGMETRICS Conf. Measurement and Modeling of Comput. Syst.* (2001).

[64] STEINHAUS, M., KOLLA, R., LARRIBA-PEY, J. L., UNGERER, T., AND VALERO, M. Transistor Count and Chip-Space Estimation of SimpleScalar-based Microprocessor Models. In *Proc. of WCED-2* (2001).

[65] STEINHAUS, M., KOLLA, R., LARRIBA-PEY, J. L., UNGERER, T., AND VALERO, M. Transistor Count and Chip-Space Estimation of Simulated Microprocessors. In *T. R. UPC-DAC-2001-16, UPC* (2001).

[66] STORINO, S., AIPPERSPACH, A., BORKENHAGEN, J., EICKEMEYER, R., KUNKEL, S., LEVENSTEIN, S., AND UHLMANN, G. A commercial multithreaded risc processor, Feb. 1998.

[67] SWANSON, S., MCDOWELL, L., SWIFT, M., EGGERS, S., , AND LEVY, H. An evaluation of speculative instruction execution on simultaneous multithreaded processors. *ACM Transactions on Computer Systems 21*, 3 (2003), 314 340.

[68] TARJAN, D., THOZIYOOR, S., AND JOUPPI, N. Cacti 4.0. Technical Report HPL-2006-86, Hewlett-Packard. Tech. rep.

[69] TULLOCH, P. Discussing the Many-Core Future, 2007.

[70] TULLSEN, D. M., AND BROWN, J. A. Handling Long-latency loads in a Simultaneous Multithreaded Processor. In *Proceedings of 34th Annual ACM/IEEE International Symposium on Microarchitecture* (2001), pp. 318 – 327.

[71] TULLSEN, D. M., EGGERS, S., AND LEVY, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of 22nd Annual International Symposium on Computer Architecture* (1995).

[72] TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of 23rd Annual International Symposium on Computer Architecture* (1996).

[73] UNGERER, T., ROBIC, B., AND SILC, J. Multithreaded processors. *The Computer Journal 45*, 3 (Nov. 2002), 320341.

[74] UNGERER, T., ROBIC, B., AND SILC, J. A survey of processors with explicit multithreading. *ACM Computing Surveys 35*, 1 (Mar. 2003), 2963.

[75] USAMI, K., AND HOROWITZ, M. Clustered voltage scaling techniques for low-power design. In *Proc. of the Intl. Symposium on Low Power Electronics and Design* (1995).

[76] WALL, D. W. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (1991), pp. 176 – 188.

[77] WECHSLER, O. Inside Intel Core Microarchitecture - Setting New Standards for Energy-efficient Performance . *White Paper* (2006).

[78] WOLF, W. FPGA-Based System Design. Prentice Hall, 2004.

[79] YAMAMOTO, W., AND NEMIROVSKY, M. Increasing superscalar performance through multistreaming. In *Proceedings of the 1st International Conference on High Performance Computer Architecture* (1995), p. 49 58.

[80] YAMAMOTO, W., AND NEMIROVSKY, M. Increasing superscalar performance through multistreaming. In *Proc. of PACT* (1995).

[81] YEH, T., AND PATT, Y. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of 20th Annual International Symposium on Computer Architecture* (1993), pp. 257 – 266.

[82] ZAKI, O., MCCORMICK, M., AND LEDLIE, J. Adaptively Scheduling Processes on a Simultaneous Multithreading Processor.