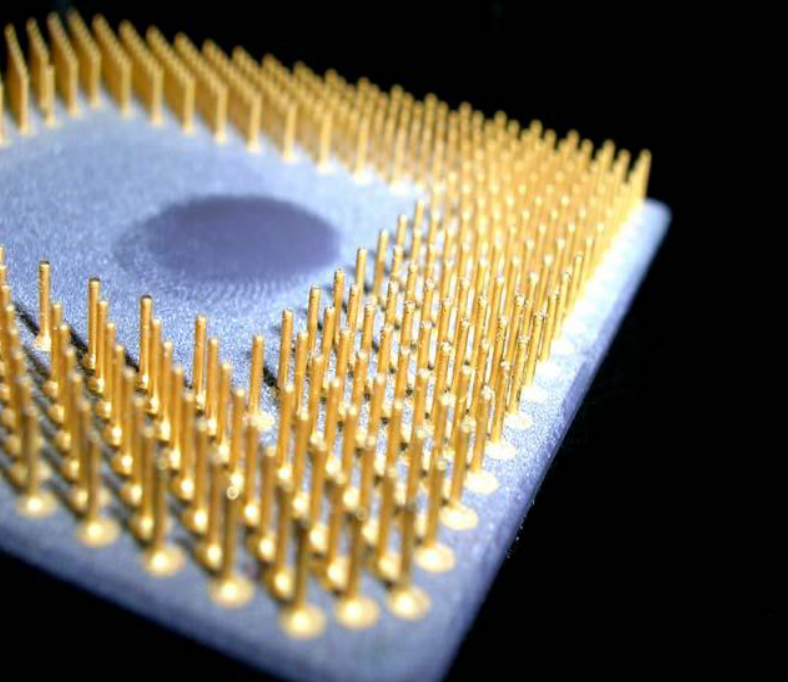


ADVERTIMENT. La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del servei TDX (www.tesisenxarxa.net) ha estat autoritzada pels titulars dels drets de propietat intel·lectual únicament per a usos privats emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA. La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del servicio TDR (www.tesisenred.net) ha sido autorizada por los titulares de los derechos de propiedad intelectual únicamente para usos privados enmarcados en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio TDR. No se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING. On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the TDX (www.tesisenxarxa.net) service has been authorized by the titular of the intellectual property rights only for private uses placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading and availability from a site foreign to the TDX service. Introducing its content in a window or frame foreign to the TDX service is not authorized (framing). This rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author



tesis doctoral

Runahead Threads

Tanausú Ramírez García



UNIVERSITAT POLITECNICA DE CATALUNYA
Departament d'Arquitectura de Computadors

©2010 TANAUSU RAMIREZ GARCIA
ALL RIGHTS RESERVED



UNIVERSITAT POLITÈCNICA DE CATALUNYA
Departament d'Arquitectura de Computadors

Runahead Threads

Tanausú Ramírez García

Advisors:

Mateo Valero

Universitat Politècnica de Catalunya

Alex Pajuelo

Universitat Politècnica de Catalunya

Oliverio J. Santana

Universidad de Las Palmas de Gran Canaria

A THESIS SUBMITTED IN FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY (COMPUTER SCIENCE)
DOCTOR POR LA UPC

A mis padres, por su amor, apoyo y aliento

Acknowledgments

Esta tesis ha sido el resultado de una gran aventura fuera de mi tierra, Gran Canaria. Durante este largo camino, ha habido muchos momentos de alegría y satisfacción personal, así como de malos y duros momentos. Sin embargo, he tenido la suerte de contar con el apoyo de muchas personas, a las cuales les quiero dedicar estas líneas pues sin ellas no habría llegado hasta aquí.

En primer lugar, quiero agradecer a mi director de tesis, Mateo Valero, por su confianza en mí desde el principio. Sin aún conocerme, me invitó a una cena de un congreso nacional, y le bastó un trayecto en su coche desde Lleida a Barcelona para animarme a hacer el doctorado. Mateo me ha brindado su ayuda, sus conocimientos y su apoyo, así como la oportunidad de comenzar mi carrera investigadora en el Departamento de Arquitectura de Computadores (DAC) de la UPC.

Igualmente, quiero dar las gracias a mis otros directores de tesis. A Alex Pajuelo, por su confianza e infalible ayuda tanto en el aspecto profesional como en el personal para que este trabajo saliera adelante. A Oliver Santana, por su dedicación y ayuda prestada en todo momento tanto aquí como desde la distancia. Posiblemente esta tesis no estaría terminada sin vuestra ayuda, simplemente gracias.

Para realizar esta tesis tuve que moverme de Las Palmas a Barcelona, y con ello dejé atrás a mi familia y muchos amigos en Canarias. En este viaje, agradecer especialmente a Carmelo Acosta, amigo y compañero desde el colegio, Javier Verdú y Fran Cazorla compañeros también en la ULPGC. Todos ellos, ya doctores, tomaron previamente el mismo camino que yo, animándome a venir, haciendo mi llegada más fácil y ayudándome en todas las situaciones durante estos años. Gracias Carmelo y Hema por ofrecerme vuestra casa, pero especialmente vuestra compañía en los primeros momentos.

Una vez aquí, también he conocido a nuevas personas estupendas, que además de compañeros de doctorado se han convertido en muy buenos amigos. Entre ellos, quiero mencionar especialmente a Josep María Codina, Ayose Falcón, Ana Bosque, Jaume Abella y Rubén Gran. Me gustaría agradecerles su apoyo, ánimo, amistad y compañía

en las diferentes etapas de esta tesis, pero sobre todo gracias por los momentos de diversión que me ayudaron a superar los obstáculos con alegría.

También agradecer a tantos otros compañeros del DAC que he ido conociendo durante estos años, y con los cuales he podido compartir reuniones de trabajo, sala de becarios, horas de comida, cafés y muchas otras actividades: Adrián Cristal, Marco Galluzi, Alejandro García, Beatriz Otero, Jordi Guitart, Germán Rodríguez, Llorenç Cruz, Ramon Canal, Fermín Sanchez, Javi Alonso, Miquel Moreto... y muchos más que espero que no se molesten por no nombrar aquí. Igualmente dar las gracias al resto de profesores y miembros del departamento que lo conforman y lo hacen uno de los más importantes del mundo en su área de conocimiento.

Del mismo modo, destacar que todo este trabajo ha sido financiado por la beca FPU AP2003-3682 del Ministerio de Educación y Ciencia, así como por las líneas de investigación de la Comisión Interministerial de Ciencia Y Tecnología (CICYT) TIN-2004-07739 y TIN2007-60625, y también por la red europea de excelencia High Performance Embedded Architectures and Compilers (HiPEAC).

Personalmente, esta tesis no sólo ha supuesto un reto profesional sino seguramente un gran cambio en mi vida. Gracias a la tesis, he podido conocer otra tierra, otra gente, viajar y realizar muchas otras actividades, como entrar en el mundo de la salsa. Y así, bailando salsa, he podido desconectar de las situaciones de estrés y abatimiento en los momentos más difíciles. Pero lo más importante es que he conocido a muchas personas salseras que valen la pena, Oscar, Susana, Joaquín, Cristina Robles, Eva, Cristina Salinas, Luba, Patrick... y a las cuales quiero agradecerles los muchos momentos de diversión y amistad que han compartido conmigo.

De manera muy especial, quiero dar las gracias de todo corazón a Judit, la persona más maravillosa que he podido conocer, pues su forma de ser transformó nuestro baile en amor. No tengo palabras para agradecerle su cariñosa compañía, por haberme apoyado y alentado repetidas veces, pues gracias a ella encontré las fuerzas necesarias para llegar hasta el final. Y, por supuesto, gracias por todo el amor con el que me rodeas, del cual espero disfrutar toda mi vida.

Finalmente, mis más profundo y sentido agradecimiento a mi familia, que con amorosa paciencia, y entregándome su incondicional ayuda, también me han apoyado en esta etapa de mi vida, soportando la distancia que nos separaba. Mi madre Yolanda, mi padre Antonio, mi hermana Yurena, y mi abuela Tata, a pesar de la distancia siempre pude sentir su cariño. Les quiero mucho.

Abstract

The evolution of processors has been supported by both the rapid technology advance and smart architecture designs. Until recently, processors mainly focused on exploiting instruction-level parallelism (ILP) to improve the performance. Nevertheless, the combination of the practical limits to superscalar processor implementations, the theoretical limits of instruction-level parallelism, and the increment of power constraints has forced different changes in processor designs. In this sense, the computer architecture community turns to new research towards other forms of parallelism.

Fruit of this trend, thread-level parallelism (TLP) has become a common strategy for improving processor performance. Simultaneous Multithreading (SMT) is one of these new paradigms to exploit TLP in recent years. The main feature of SMT processors is to simultaneously execute multiple threads to increase the utilization of the pipeline by sharing many more resources than in other types of processors. However, the memory wall problem is still present in these processors. Memory-intensive threads tend to use processor and memory resources poorly creating resource contention problems. This kind of threads can clog up shared resources due to long-latency memory operations without making progress on a SMT processor, thereby hindering the overall system performance.

In this dissertation, we target these problems with a two-fold objective: to improve the performance of these multithreading processors and to propose techniques to be as efficient as possible inside the current power constraints. To cover these goals, this dissertation firstly proposes Runahead Threads (RaT) to alleviate the memory wall problem, and secondly, provides additional techniques to improve the efficiency of our initial proposal.

Contents

1	Introduction	1
1.1	Multithreaded models	2
1.1.1	Simultaneous Multithreaded Processors	4
1.1.2	SMT trends	5
1.2	Thesis overview	6
1.2.1	The motivation: SMT problems	6
1.2.2	The approach: Runahead Threads	9
1.3	Thesis contributions	11
1.4	List of publications	13
1.5	Thesis organization	14
2	Evaluation Framework	15
2.1	Simulation tools	15
2.1.1	SMTSIM	15
2.1.2	Power tools	20
	Wattch	20
	CACTI	21
2.2	Benchmarks	22
2.3	Evaluation methodology and metrics	25
2.3.1	Performance metrics	25
2.3.2	Power and efficiency metrics	26
3	Runahead Threads	29
3.1	The idea of Runahead Threads	30
3.2	Fundamentals of Runahead Threads	32
3.2.1	Starting a runahead thread	33
3.2.2	Executing a runahead thread	34

3.2.3	Exiting runahead thread	36
3.3	Implementation of Runahead Threads	36
3.3.1	Implementation issues for starting a runahead thread	37
3.3.2	Implementation issues for runahead thread execution	38
	Register validation control	38
	Load and store management	39
	Floating-point resources	40
	Synchronization	40
3.3.3	Implementation issues for exiting from a runahead thread	41
3.3.4	Overall hardware cost and complexity	41
3.4	Evaluation of Runahead Threads	42
3.4.1	Performance of Runahead Threads	43
	Throughput	43
	Per-thread SpeedUp	45
	Hmean	47
3.4.2	Benefits and limitations of Runahead Threads	48
	Prefetching	48
	Resource contention	50
	Instruction overhead	51
3.4.3	Life time of runahead threads	53
3.5	Sensitivity of Runahead Threads to processor parameters	55
3.5.1	Hardware contexts	56
3.5.2	Memory latency	58
3.5.3	L2 cache size	60
3.5.4	Reorder buffer size	61
3.5.5	Shared vs. non-shared ROB	64
3.5.6	Register file size	65
3.6	Design analysis for Runahead Threads implementation	68
3.6.1	Fetch policy setup	68
3.6.2	Different thread priority schemes for Runahead Threads	70
3.6.3	Checkpointing delay	73
3.6.4	Use of the runahead cache	74
3.7	Summary	76

4	Code Semantic-Aware Runahead Threads	79
4.1	Efficiency and Runahead Threads	80
4.2	Overseeing program structures to improve RaT efficiency	82
4.3	Loop control techniques	83
4.4	Subroutine control techniques	89
4.5	Loops and subroutines analysis	97
4.5.1	Loops in runahead threads	97
4.5.2	Subroutines in runahead threads	100
4.5.3	Statistics about the controlled loops and subroutines	101
4.6	Evaluation of code semantic techniques	103
4.6.1	Impact of using the saturated counters	104
4.6.2	Performance evaluation	105
4.6.3	Extra work	107
	Speculative instructions	108
	Power consumption	108
4.7	Summary	110
5	Efficiency-aware Runahead Threads	113
5.1	Runahead distance prediction	114
5.1.1	Useful runahead distance	114
5.1.2	The first approach	116
5.1.3	The second approach	118
5.1.4	Implementation issues related to runahead distance techniques	122
5.2	Evaluation of runahead distance mechanisms	124
5.2.1	Analysis of refinements applied to R2DP technique	125
	Distance difference threshold	125
	Control heuristic to manage the distances with zero value	126
5.2.2	Performance and extra work evaluation	128
5.2.3	Energy efficiency	131
5.2.4	Evaluation of dynamic adaptivity of runahead distance	134
5.3	Analysis of the approaches	136
5.3.1	Runahead distance prediction accuracy	136
5.3.2	Predicted useful runahead distance	137
5.3.3	Length of runahead threads	138

5.3.4	Number of runahead threads	140
5.3.5	Distribution of controlled runahead threads	140
5.4	Summary	143
6	Related Work	145
6.1	Simultaneous Multithreading	145
6.1.1	Instruction fetch policies	146
6.1.2	Dynamic resource allocation techniques	148
6.1.3	MLP-aware policies	150
6.2	Runahead paradigm	151
6.3	Thread-based speculative techniques	153
7	Overall Mechanism Comparison	157
7.1	State-of-the-art SMT mechanisms	157
7.1.1	Performance evaluation	158
I-fetch policies	158
Dynamic allocation techniques	160
MLP-aware policies	162
7.1.2	Extra instruction execution	164
7.2	Efficient runahead thread techniques	167
7.2.1	Performance	168
7.2.2	Energy efficiency	170
7.3	Summary	174
8	Conclusions	177
8.1	Goals and conclusions	177
8.2	Remarks and future directions	181
	List of figures	187
	List of tables	189
	References	197

Chapter 1

Introduction

The incredible progress of computer development up to high-performance modern processors has come both from advances in technology and from innovation in computer design. Computer architecture and processor manufacturing have been able to provide faster processor designs by the available technology. For decades, the shrinking of transistors continuing to follow Moore's law [48] has allowed computer architects to improve processor performance by exploiting bit-level parallelism, instruction-level parallelism and memory hierarchies.

However, several factors have steered computer architects away from continuing to design increasingly more powerful single-thread processors in the last decade. The combination of the non-scalability of conventional out-of-order processors, limited instruction parallelism and power constraints has forced changes in processor designs. As a result, processor architects have focused their efforts on finding new alternatives to effectively utilize the millions of transistors available to improve performance in ways that minimize both design complexity and additional power usage.

In this scenario, *Multithreading* paradigm has become popular to deal with the above problems. Since it is difficult to extract more instruction-level parallelism (ILP) from a single program, architects have opted to execute multiple programs by exposing thread-level parallelism (TLP). Unlike ILP, which exploits implicit parallel instructions within a executed program, TLP is explicitly represented in a higher level of parallelism by the use of multiple threads of execution. The motivation to exploit TLP comes from the idea of using this parallelism among threads as another source to

find independent instructions when insufficient ILP exists. Therefore, processors with multithreading capabilities exploit the concurrency of multiple tasks to optimize the processor's performance by running a larger number of threads.

Since multithreaded processors have rapidly become a common strategy for improving processor performance, the research on this trend plays an important role in future processor designs. This thesis takes a step in the direction of improving the performance of Simultaneous multithreaded processors, one of the promising implementation models of multithreading.

1.1 Multithreaded models

Multithreaded architectures are driven by the realization that single-threaded superscalar processors spend a surprising amount of time doing nothing. On the one hand, conventional processors have many execution units, and a single thread rarely uses all of them at once. On the other hand, when a thread stalls because it needs data that is not stored in the cache, the CPU ends up waiting for memory for hundreds of cycles. In both of these cases, having another thread immediately ready to run can fill up these empty spaces, and can prevent wasting resources.

Hence, several multithreaded models are proposed to make better use of processor resources by executing several threads. Figure 1.1 depicts the different approaches of multithreaded models in function of the strategy to share the expensive execution resources on the processor core. In this figure, we represent that the processor either finds an instruction to execute (filled box) or the corresponding resource slot is unused in that clock cycle (empty box). Each color represents a different thread in execution. From the processor view, a *thread* represents a hardware context of execution with its own instructions and data.

- In a *superscalar architecture*, without multithreading support, only one thread is running at a given time. The lack of ILP limits the ability to issue more instructions per cycle. Superscalar processors often have more functional unit parallelism available than a single thread can effectively use. In addition, a long-latency event (such as a main memory access) can leave the entire processor idle.

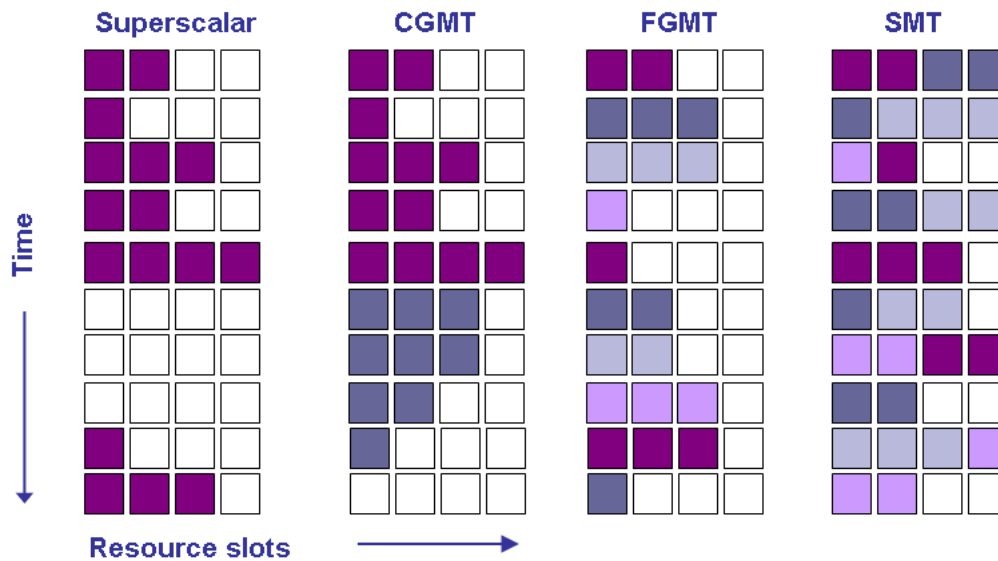


Figure 1.1: Different approaches of multithreaded processor. The horizontal dimension represents the execution resources used in each clock cycle. The vertical dimension represents a sequence of clock cycles.

- In a *coarse-grain multithreaded* (CGMT) processor, threads share all execution resources but instructions are issued from a single thread in each cycle. A CGMT processor switches to a different thread when a thread experiences a long-latency event, for example an off-chip cache misses. This allows the processors to hide part of the latency of long-latency operations by switching to another thread. Although this reduces the number of idle clock cycles, the ILP limitations still lead to idle resource slots within each cycle. Furthermore, CGMT processors suffer from the pipeline start-up period of threads each time there is a thread switching.
- In the *Fine-grain multithreading* (FGMT) case, the processor interleaves the execution of multiple threads switching them in alternate cycles. The main difference between coarse-grain and fine-grain multithreading is the granularity at which context switches occur. In a FGMT processor context switches are often done on every clock cycle (round-robin fashion), skipping any threads that are stalled at that time. The primary disadvantage is that it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads. Besides, ILP limitations

still lead to idle slots within individual clock cycles because only one thread issues instructions in a given clock cycle.

- *Simultaneous Multithreaded* (SMT) processors eliminate context switching between multiple threads by allowing instructions from multiple simultaneously active threads to occupy processor's execution slots. SMT processors fetch instructions from different independent threads, and mix them in the processor pipeline reducing the number of idle slots. As a result, an SMT processor not only can switch to a different thread to use the idle issue cycles in a long-latency operation, but also fill unused issue slots in a given cycle with other threads (really exploiting the TLP). The drawbacks come from the imbalances in the resource needs and resource availability over multiple threads.

Although this figure greatly simplifies the real operation of these processors, it does illustrate the different potential performance advantages of multithreading.

1.1.1 Simultaneous Multithreaded Processors

In particular, Simultaneous Multithreading [76][84] is one of the most promising implementations for the exploitation of TLP. An SMT processor allows dynamically varying the interleaving of instructions from multiple threads across shared execution resources. Therefore, SMT processors are able to exploit TLP as well as ILP by filling the pipeline with instructions fetched from multiple threads.

The key premise behind the simultaneous multithreading is that single thread multiple-issue processors already have many of the hardware mechanisms needed to support this multithread approach. SMT processors inherit from superscalar ones the ability to issue multiple instructions each cycle and the hardware support to execute out-of-order from multiple threads. More precisely, superscalar processors have a physical register file that can be used to hold the register sets of independent threads, though it has to deal with larger register files to hold multiple contexts. In addition, register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in the data path without confusing sources and destinations across the threads (assuming separate renaming tables are kept for each thread). Furthermore, multiple instructions from independent threads can be issued without regard to the dependencies among them since the resolution of their dependencies can be handled by the

dynamic scheduling capability. These observations enable straightforward implementations of SMTs on top of an out-of-order processor by basically adding a per-thread renaming table, keeping separate program counters (PCs), and providing the capability for instructions from multiple threads to commit (per-thread retirement).

SMT processors combine the multithreading technique with a wide-issue out-of-order processor to overcome the under utilization of a conventional superscalar processor. Multiple instructions from different threads are mixed within pipeline stages, allowing the extensive resources within these pipe stages in a superscalar processor to be more fully utilized. Thus, SMT achieves better performance throughput than single-threaded superscalar processors for multithreaded (or multiprogramming) workloads, improving the overall efficiency.

1.1.2 SMT trends

Simultaneous multithreading has already had impact in both the academic and commercial communities. Thus, different forms of SMT processors have been presented as research proposals [76][84] and many others have been adopted in real systems. Compaq proposed the Alpha EV8 21464 [25] that consists on a four-threaded eight-issue SMT processor. In this thesis, we take this known design as the base for the SMT model. Multithreading is also mentioned as processor technique for the architecture projects of the Sun UltraSPARC family [39], starting from UltraSPARC IV [66]. *Hyper-Threading* [45], the Intel's brand name for Simultaneous Multithreading, is conceptually similar to the SMT proposals in which some portions of the Pentium 4 pipeline are multithreaded in a partitioned fine-grained fashion. For instance, the Intel Pentium 4 Xeon processor family [38] incorporates this hybrid form of multithreading that enables two logical processors to share some of the execution resources of the processors.

Likewise, as transistor dimensions continue to shrink thanks to process technology, another architectural approach for achieving TLP is represented by Chip Multiprocessors (CMPs) [52]. CMPs support the execution of more than one thread per processor chip by replicating an entire processor core for each thread. Both SMT and CMP architectures lead to orthogonal implementations also known as Chip Multithreading (CMT), having multiple cores able to execute several threads in one chip. In this sense, mixed designs and commercial processors with multithreaded and chip multiprocessors

have appeared. For example, the Intel Core architectures [32] and Sun UltraSPARC T-series also known as Niagara processors. UltraSPARC T1 [67] in 2006 and next UltraSPARC T2 [68] in 2008 are both multicore and multithreaded processors, with up to eight CPU cores, each one able to concurrently handle four or eight threads respectively. There are other current commercial examples, such as the dual-core IBM POWER5 [36] and the succeeding POWER6 [26], processors with support for simultaneous multithreading with two threads per core. Nowadays, new simultaneous multithreaded and chip multicore processors have been announced or are already in production by Intel, AMD, IBM and Sun.

1.2 Thesis overview

1.2.1 The motivation: SMT problems

SMT is going to become more important as this type of technology is making its way into CPU designs by all the major processor manufacturers. Therefore, many future processors will continue implementing some form of SMT since constitute an actual solution to improve the performance/cost ratio of processors. SMT designs exploit the thread-level parallelism by sharing hardware resources among multiple threads. However, the ability to get more performance (exploiting both instruction level parallelism and thread level parallelism) in these multithreaded processors also runs into two key related problems: the memory wall problem and the resource monopolization.

SMT processors can tolerate better the main memory latencies while running a multiprogrammed or multithreaded workload. Long latencies occurring in the execution of single threads are bridged by issuing operations of the remaining threads loaded on the processor. However, SMT processors still suffer from the memory wall problem [83] because long memory latencies continue having importance due to their impact on performance in the execution of a single-thread and in the overall SMT processor performance.

On the one hand, because of the growing memory access latencies (measured in processor cycles), any request from a thread that misses in the caches may eventually take hundred of cycles to satisfy. For example, commercial applications such as transaction processing workloads see 65% processor idle times, and high-performance scientific computations see 95% processor idle times: much of this is due to memory

latencies [47]. Thus, the execution of a thread continues being dominated by memory latencies.

On the other hand, this memory wall creates another critical side problem in the SMT framework: threads with intensive streams of memory accesses (memory-intensive threads) can create important resource contention in SMT processors [74]. As each thread has access to as many hardware resources as it needs, the features of memory intensive threads can unbalance the resource allocation holding more resources than others. A thread with a long-latency load pending will not make forward progress because following instructions can neither issue nor commit while waiting for the memory operation. This causes the thread to hoard a lot of critical resources due to all these pending instructions in the pipeline. As consequence of this situation, the shared resources are clogged, thereby starving other threads of required resources and preventing their forward progress as well. This effect would likely lead to a resource monopolization by a single thread and significantly degrading the performance of the other threads in the processor.

Figure 1.2 illustrates this important problem. This figure depicts two different threads executing their corresponding instructions in different steps. These instructions require certain resource slots which are taken from a resource pool. This resource pool represents whatever shared resource of the processor such as the issue queues, physical registers or reorder buffer. During this multithreaded execution, the thread 1 suffers from a long-latency miss (see 1.2(b)). From this point onwards, the subsequent instructions that already have resources allocated can not commit for this thread. Furthermore, the new introduced dependent instructions also allocate more resources slots. This effect causes the thread 1 to strangle the execution of the thread 2 monopolizing the resources slots (as the 1.2(d) shows) until the long-latency miss is solved.

To handle this long latency problem, different fetch and resource dynamic policies have been proposed to control the use of resources by memory-intensive threads (see Chapter 6). These techniques identify those threads that suffer from long-latency loads, and limit their use of machine resources by stalling or flushing these threads under determined conditions to prevent resource overuse. In these approaches, memory-intensive threads are usually restricted in order to get higher global throughput. Although these policies alleviate the memory-intensive thread monopolization, the performance of fast threads is not reduced at the cost of stalling the execution of memory-intensive threads.

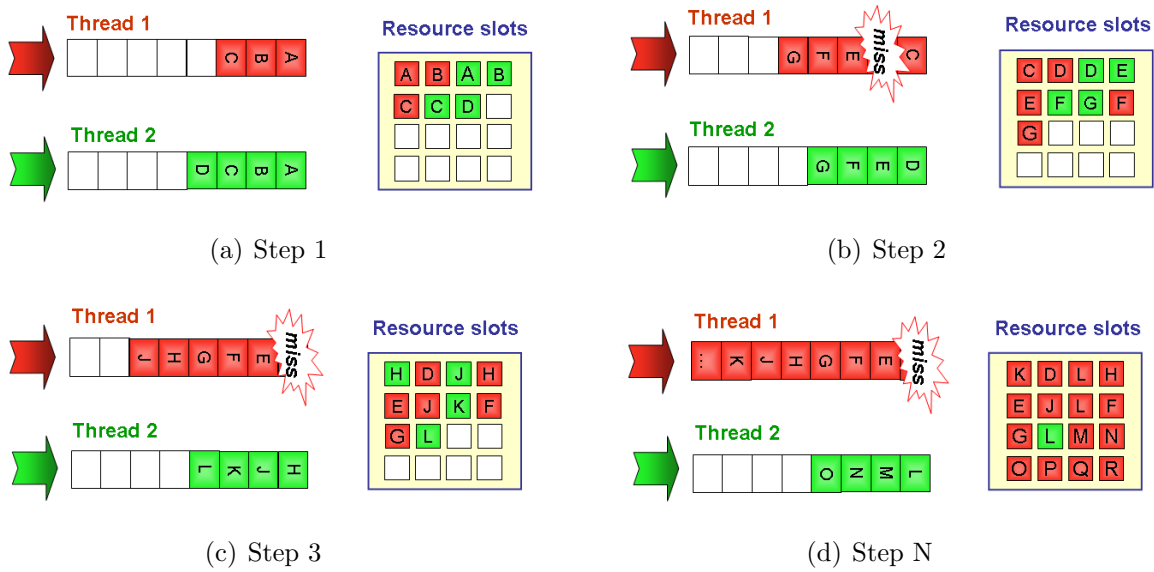


Figure 1.2: Effects of long-latency misses on SMT processors

In addition, these existing techniques do not assume memory intensive threads exhibit other kind of parallelism as is the memory-level parallelism (MLP) [31]. Consequently, these policies are harmful for memory-intensive applications because threads are stopped before they can exploit the available MLP, failing to reach the potential performance achievable. Therefore, significant potential exists to improve the performance of the SMT processors by improving their tolerance to long main memory latencies with regard to memory intensive threads.

Several large instruction window approaches [1][18][19][41][46][65] have been proposed to maintain a high number of instructions in the pipeline to exploit the ILP and MLP as much as possible. The idea of building a large instruction window is worthy of research since these proposals have shown significant single-thread performance. However, they suppose challenges in their implementations due to the design complexity, verification process, and increase in the energy consumption of single-thread processors. For instance, the proposed solutions still require very large buffers, hierarchical structures and non-trivial control logic in the processor core. Hence, these alternatives maybe are even more complex and difficult to implement in multithreaded processors, in which some of large and complex structures to support large instruction windows should be shared and replicated as well. Therefore, large-instruction windows are promising in the single-thread processors, but in the multithreaded scenario, where

current trends point towards simpler cores, they are less suitable. Finally, as power consumption has already become a limiting constraint in the design of high-performance processors, simple power and efficient proposed techniques are especially desirable.

1.2.2 The approach: Runahead Threads

The main purpose of this dissertation is to resolve the problems related with the memory wall on SMT processors with a two-fold objective: to improve the performance and to propose efficient techniques inside the current power constraints. To cover all these goals, this dissertation first proposes introducing and evaluating the mechanism of Runahead Threads (RaT) to alleviate the memory wall problem and improve SMT performance, and second, providing additional techniques to improve the power-efficiency of our initial proposal.

Before beginning this work, previous techniques tend to distribute resources to threads that infrequently miss in the caches, i.e. which are compute-intensive. By contrast, we target our approach to improve the performance impact of memory-intensive threads in SMT machines without penalizing the other thread performance. At the same time, we want to provide the latency tolerance provided by a large instruction window without having to design and implement structures to support a large number of in-flight instructions. To accomplish this, we extend the Runahead paradigm for SMT processors by *Runahead Threads* (RaT). Runahead [23][51] is an execution scheme that generates accurate data prefetches by speculatively executing instructions past a long latency cache miss, when the processor would otherwise be stalled. Runahead consists of avoiding the blockage of the instruction window due to long-latency operations, *e.g.* a load that misses in the last level cache. Instead, the processor continues executing instructions speculatively, trying to follow the most likely program path until the load that triggered the runahead mode is resolved.

Through Runahead Threads (RaT), we alleviate the problems related to long-latency loads in SMT processors. First, runahead threads do not clog up resources (and thus they do not starve other threads) on long-latency load misses. A runahead thread allocates and deallocates shared processor resources quickly instead of holding on to them until the long-latency load completes. This allows other threads to make use of the available shared resources. Second, runahead threads exploit further MLP going beyond the processor's instruction window.

The main novelty of this alternative proposal regarding previous SMT techniques is that the memory-intensive threads advance speculatively, instead of being stalled. On the one hand, RaT improves the memory latency tolerance of each individual thread by exploiting the memory-level parallelism available. Each runahead thread does useful processing to improve their performance through prefetching. This provides benefits on a single-threaded application, which is not provided by multithreading. On the other hand, our mechanism also avoids memory-intensive threads clogging up shared resources, transforming them into light speculative threads and allowing all threads to continue executing with the shared resources. Other important advantage is that it is required small changes in the hardware to support RaT in the SMT processor. Thus, RaT mechanism does not require significant hardware cost or complexity.

However, these benefits of RaT come at the cost of executing a large number of instructions speculatively. If runahead thread execution does not provide prefetching benefits, it can result in degrading efficiency by executing a large number of useless instructions. To avoid this problem, this thesis also presents several contributions aimed not only at improving the performance of SMT processors, but also to increase the efficiency of Runahead Threads. In this dissertation, we propose and evaluate different techniques to control runahead thread executions in order to make RaT a more efficient mechanism.

The next proposals are focused on studying how to enhance the effectiveness of RaT. We focus on reducing the speculative work done by runahead threads as much as possible, without degrading their performance benefits. On the one hand, we present a kind of proposals base on analyzing the prefetch opportunities (usefulness) of executed code structures, such loop and subroutines, during the runahead thread executions. These techniques dynamically use some semantic information to detect these particular program structures and to analyze when they are useful or not in order to control the runahead thread execution. By means of this dynamic information, the proposed techniques make a control decision either to avoid or to stall the particular loop or subroutine execution in runahead threads. These decisions make RaT reduce the useless work executed in common program structures and the most important thing, improving its efficiency.

On the other hand, we proposed another novel technique to improve the efficiency of the initial RaT mechanism reducing the number of instructions executed by controlled runahead threads. We introduce a new mechanism which is guided by the useful runa-

head distance, a new concept that indicates how far a thread should run ahead such that speculative runahead execution is useful. Based on the useful distance prediction, our technique makes two actions. First, it predicts whether or not a thread should start runahead execution, (i.e. whether or not runahead execution is useful) to avoid the execution of useless runahead periods. Second, it indicates how long the thread should execute in runahead mode to reduce unnecessary extra work done by useful runahead periods. Limiting the runahead distance of a thread not only avoids unnecessary speculative execution but also allows more shared resources to be efficiently used by non-speculative threads. As result of this proposal, we significantly reduce power consumption of RaT, providing better performance and energy balance than previous proposals in the field.

1.3 Thesis contributions

The contributions of this thesis are summarized in the following main points:

- The mechanism of Runahead Threads. The contribution of Runahead Threads is a different mechanism to overcome the memory wall problem on SMT processors. Although we have been inspired by previous Runahead paradigm, we introduce and evaluate for the first time Runahead Threads for multithreaded scenarios.
- By Runahead Threads, we greatly improve the overall performance of SMT processors exploiting ILP, TLP and MLP. We show that with RaT it is possible to improve the performance of single-thread by the runahead prefetching effect and to avoid the resource monopolization of memory-intensive threads. This contribution represents a completely different approach to tackle the problem of handling long-latency loads on SMT processor.
- Runahead Threads are deeply evaluated. Aside from its different benefits to alleviate the commented SMT problems, we demonstrate that RaT can benefit from smaller register file with even performance improvements. This may allow SMT designs with less expensive hardware. We also show the major drawback of runahead threads. In case there is no prefetching to be performed, the runahead threads execute useless speculative instructions resulting in lower efficiency. To solve this shortcoming, several alternatives for improving the RaT efficiency are proposed with a different point of view.

- Code semantic-aware runahead threads. This contribution consists on a code semantic-aware set of techniques to improve RaT efficiency. These techniques perform coarse-grain analysis based on code semantics to oversee the prefetching usefulness of loops and subroutines during runahead executions. In function of this analysis, the processor makes different actions to control the runahead threads. While these alternatives are targeted to particular code patterns, these show good relation between instruction reduction and performance.
- The Runahead Distance Prediction approach. This proposal enhance the efficiency of RaT based on the new concept of useful runahead distance. The useful runahead distance represents the maximum MLP achievable by a particular runahead thread while at the same time reducing the extra useless speculative work. We propose two different generic mechanisms that perform a fine-grain analysis of each runahead thread to collect that useful runahead distance. The novelty is that we predict how long the thread should execute in runahead mode to be efficient. We show as our mechanism is able to eliminate more useless speculative instructions (even in useful runahead periods) that cannot be eliminated by previous techniques. Furthermore, we achieve reducing the power consumption while maintaining the overall SMT performance with RaT.
- Finally, we provide a quantitative survey of RaT mechanisms from this dissertation and the state-of-the-art related techniques. We show that RaT approaches provide significant performance outperforming state-of-the-art techniques. As additional contribution, we provide experiments to compare these related proposals including novel power and energy efficiency results not considered in prior work. We present the first evaluation (to the best of our knowledge) that give power and energy measurements in the Runahead paradigm.

1.4 List of publications

Here, we compile and list all of the publications arise from the research and contributions of this dissertation:

- Tanausú Ramírez, Alex Pajuelo, Oliverio J. Santana and Mateo Valero. *Kilo-instruction Processors, Runahead and Prefetching*. In ACM conference on Computing Frontiers (CF'06), pages 269-278. May 2006, Ischia, Italy.
- Tanausú Ramírez, Alex Pajuelo, Oliverio J. Santana and Mateo Valero. *A First Glance at Runahead Threads*. In HiPEAC Advanced Computer Architecture and Compilation for Embedded Systems (ACACES'07), pages 107-110. July 18th, 2007, L'Aquila, Italy.
- Tanausú Ramírez, Oliverio J. Santana, Alex Pajuelo and Mateo Valero. *Runahead Threads: Reducing Resource Contention in SMT Processors*. In IEEE-ACM Conference, Parallel Architectures and Compilation Techniques (PACT'07), page 423. September 15-19, 2007, Brasov, Romania.
- Tanausú Ramírez, Alex Pajuelo, Oliverio J. Santana y Mateo Valero. *Introducing Runahead Threads for SMT Processors*. In XVIII Jornadas de Paralelismo (JP'07) held in conjunction with CEDI 2007, pages 35-42. 11-14 September, 2007. Zaragoza, Spain.
- Tanausú Ramírez, Alex Pajuelo, Oliverio J. Santana, and Mateo Valero. *Runahead Threads to Improve SMT Performance*. In IEEE High-Performance Computer Architecture Conference, (HPCA'08), pages 149-158. February 18-20, 2008, Salt Lake City, USA.
- Tanausú Ramírez, Alex Pajuelo, Oliverio J. Santana, and Mateo Valero. *Code Semantic-Aware Runahead Threads*. In IEEE 38th International Conference on Parallel Processing, (ICPP'09), pages 437-444. 22-25 September 2009 Vienna, Austria.
- Tanausú Ramírez, Alex Pajuelo, Oliverio J. Santana, Onur Mutlu and Mateo Valero. *Efficient Runahead Threads*. Submitted to 24th ACM/SIGARCH International Conference on Supercomputing (ICS'10). June 2010 Tsukuba, Japan.

1.5 Thesis organization

The structure of this dissertation is as follows:

Chapter 2 is devoted to explain our experimental framework. This includes the simulation tools, the benchmarks, and metrics used in this thesis.

Chapter 3 introduces the mechanism of Runahead Threads, describing its functionality and details for the implementation on SMT processors. We evaluate and analyze in detail different design trade-offs for RaT in the multithreaded environment.

Chapter 4 describes the inefficiency problem of RaT and shows our first proposals to improve the RaT efficiency. We present code-semantic aware techniques that control the extra work on runahead thread executions. We show how by dynamically analyzing the code of programs, we can improve the RaT efficiency performing different actions during runahead threads.

Chapter 5 presents the second of our proposals focused on improving the RaT efficiency. This other approach is based on the concept of useful runahead distance for making RaT more efficient. Based on the runahead distance prediction in order to reduce the speculative instructions, different mechanisms decide when and how long a runahead thread is executed.

Chapter 6 describes prior work that help to understand the scenario of this thesis.

Chapter 7 provides a detailed mechanism comparison exploring all the state-of-the-art SMT techniques as well as results achieved with a deep evaluation between them and our performance and efficient RaT proposals.

Chapter 8 provides conclusions, a summary of the key results, and insights presented in this dissertation. We also propose some directions for future work.

Evaluation Framework

This chapter describes the simulation tools, benchmarks and the methodology that make up the experimental framework of this thesis. Based on this evaluation framework, we evaluate the different proposals, obtain the experiment results and provide the conclusions presented in this dissertation.

2.1 Simulation tools

Computer architects use different simulation tools in which they model their novel ideas and evaluate the simulation results to measure their benefits and potential. To perform the different evaluations of this thesis, we employ a generic SMT processor simulator in conjunction with other complementary tools. They are described in the following sections.

2.1.1 SMTSIM

Our simulation environment is based on a simulator derived from SMTSIM [73]. SMTSIM is an execution-driven simulator which models generic Simultaneous Multithreaded processors and provides high flexibility. This simulator is able to run unmodified Alpha object code (we will describe our benchmarks later in Section 2.2).

We modify SMTSIM to support simulation checkpoints. A simulation checkpoint allows us to store the simulator execution state maintaining the execution-driven feature

while reducing the simulation time considerably. In addition, we improve the memory hierarchy modeling bus, port and bank contention, bandwidth, and main memory timings accurately.

In order to evaluate all the contributions, the different designs of our mechanisms, as well as other fetch and resource scheduling techniques for comparison purposes, are built on top of this simulator. A scheme of the simulator processor model is depicted in Figure 2.1 in which the pipeline and main hardware components of this baseline architecture are shown.

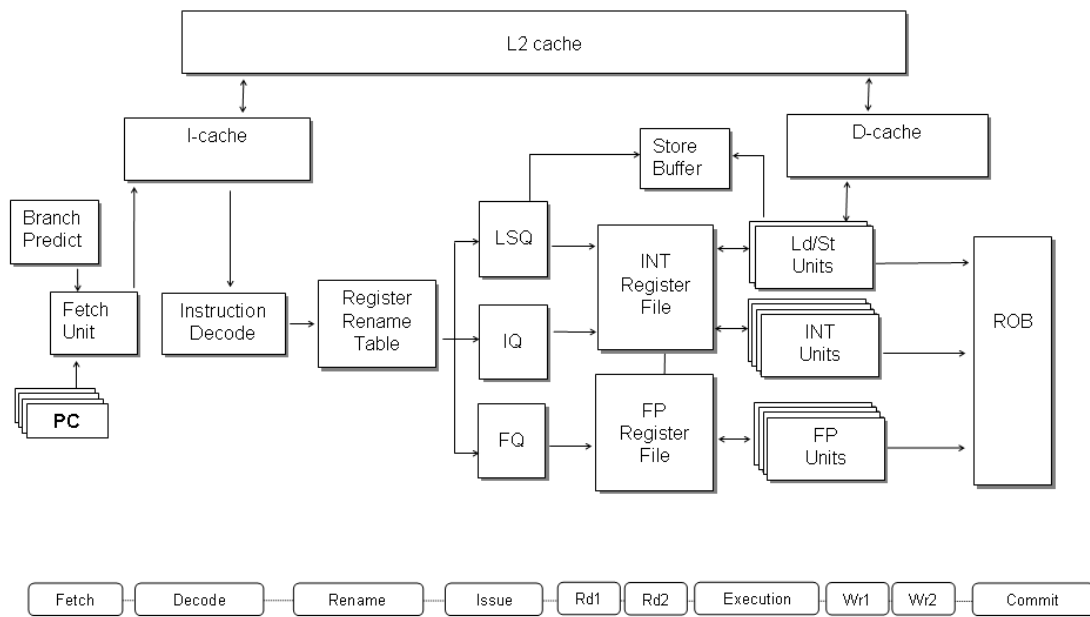


Figure 2.1: Block diagram of SMT processor architecture

The larger multiported SMT register file requires a longer access time. An actual solution to avoid increasing the processor cycle time and to maintain the processor frequency is to extend register access across the pipe stages. So, we consider to extend the stages in which the register file is accessed (both read and write operations) with the corresponding additional bypasses. Now, read and write stages take two cycles each one. As result, this SMT architecture is composed of ten stages as it is shown in Figure 2.1. For instance, the pipeline of Intel Core micro-architecture is 14 stages long [82].

The processor fetches instructions from the instruction cache in program order. We use an underlying baseline fetch policy which determines from which of the available

threads instructions are fetched. We choose ICOUNT (2,8) policy [75] which in each cycle, it selects two different threads (according to a fetch priority) and fetches up to eight instructions from each thread. By previous analysis [75], the ICOUNT (2,8) scheme performs 10% better than fetching from one thread at a time (1,8) and 5% better than fetching four instructions from each of two threads (2,4). We also perform an evaluation of the ICOUNT setup policy in Section 3.6 according to the new context of this dissertation. To support fetching from two different threads in each cycle, the instruction cache (Icache) is usually replicated to get several instruction blocks. In addition, both return address stack (RAS) and branch history register (BHR) need to be also replicated for each thread to avoid the interleaving of thread interferences in the branch predictions.

Next, instructions are decoded and renamed in order to track data dependencies. When an instruction is renamed, an entry is allocated in the corresponding issue queue (IQ) until all its operands are ready. Each instruction also allocates one entry in the reorder buffer (ROB) and a physical register for mapping a destination architectural register (if it is required). While the rename table can be partitioned across threads, all physical registers are viewed as a common pool in which SMT threads share the logic that manages them (port, free list, etc...). As soon as an instruction has all its operands ready, it is issued: it reads its operands, executes in the corresponding functional unit, and writes its results. Finally, that instruction is committed in program order.

The main configuration parameters of the simulated SMT processor architecture are summarized in Table 2.1. For this architectural model, we configure an eight-way superscalar organization in which up to four hardware contexts can be executed. A key factor for configuring the processor model is the resource organization. We define a resource configuration, taking into account which of resources are shared between threads and which are exclusive for each thread. In statically partitioned multithreaded designs, each context has a fixed pool or part of a resource assigned. This leads to a lack of flexibility away from the “simultaneous multithreaded” idea of sharing resources. Academic SMT models use dynamic resource sharing, which allows threads to allocate the different idle shared resources to improve the processor utilization. In our SMT model, we use a full resource sharing organization to benefit from the dynamic multithreaded advantages. Threads coexist in the different pipeline stages, sharing the following processor resources:

Processor core	
Processor depth	10 stages
Fetch width	8 instructions (up to 2 threads per cycle)
Decode and commit width	8 instructions
Reorder buffer	shared ROB (64 entries per context)
INT/FP registers	48 physical regs. per context
INT/FP/LS issue queues	64 / 64 / 64 entries
Functional units	4 INT / 4 FP / 2 LdSt
Perceptron Branch predictor	Perceptron (256) 4096 x 14 bit local and 40 bit global history
RAS	32 entries per context
Memory subsystem	
I-cache	64 KB, 4-way, 1 cycle latency
D-cache	64 KB, 4-way, 3 cycles latency
L2 Cache	1 MB, 8-way, 20 cycles latency
Caches line size	64 bytes
MSHRs	16 per context
Main memory latency	minimum of 300 cycles

Table 2.1: SMT processor baseline configuration. In all the experiments of this thesis, these parameters remain unchanged except when explicitly stated otherwise.

1. Issue Queues (IQs): The processor considers every instruction (independently of the thread) equal from the execution point of view until they are committed. All threads insert their instructions in the corresponding IQ after the rename stage in function of the type of instruction. IQ are unified buffers that have generally high and somewhat unpredictable utilization.
2. Execution Units: All executions units are shared and fully pipelined. In the out-of-order pipeline (back-end), instruction dependency chains determine execution resource utilization more than any arbitration schemes. This parameter is set by the number of available total functional units per cycle in our model.

3. Physical registers: the physical register file (RF) is shared by all contexts. There is a register file for integer operations and another for floating-point ones. The quantity of available physical registers for renaming is critical in a scenario with several hardware contexts with their own architectural registers. The latter depends on the particular instruction set architecture (ISA). For example, for an ISA with 32 architectural registers, 128 registers are needed to maintain the precise state for a 4-threaded SMT, per each integer and floating point RFs. Our architecture supports the Alpha ISA, so each hardware context can address 32 architectural integer registers and 32 architectural FP registers. The register-renaming mechanism maps these architectural registers onto the different available physical registers for each context. The configuration of the register file size in our processor model is determined by the number of hardware contexts. We assume a contribution of 48 physical registers per context to the total RF pool available for register renaming. Hence, the architectural registers are renamed to a RF of 96 registers for a 2-context machine and 192 registers for a 4-context model.
4. Reorder Buffer: this structure keeps the program order of instructions. Using a separate ROB per thread would probably require less complexity in an actual hardware implementation, similar to replicate conventional superscalar ROB. However, we choose a shared ROB design to analyze any possible critical resource contention with this important buffer. Sharing a ROB among multiple threads introduces additional complexity. This additional complexity mainly lies in selectively squashing the speculative instructions. Nevertheless, this procedure is already implemented in an SMT processor with a shared ROB to recover from branch mispredictions supporting selective flushing of nonconsecutive entries. For most of the experiments of this work, our simulated processor uses a shared ROB for all the hardware contexts. As for RF, each hardware context has a contribution to the total ROB size of 64 entries in this case. We will study the influence of the shared ROB in our architecture in Chapter 3 as well.
5. Caches (the instruction cache, the L1 data cache, and the unified L2 cache): The I-cache and D-cache are multiported and multibanked. Sharing cache access ports between threads to maximize their utilization is one of the objectives of an SMT design. Hence, all caches are shared between threads, and then, all their

memory space are common to all threads. Caches are tagged with the identifier of threads so that independent threads do not share data and/or instructions but they competes for cache space.

2.1.2 Power tools

An important part of this work focuses on reducing the power consumption and improve the performance-energy efficiency of the proposed approaches. To perform these evaluations, we employ a power model based on Wattch [3]. In addition, caches (or similar structures) statistics such as delay or energy per access have been drawn from the CACTI tool [61].

Wattch

Wattch is an architecture-level power and performance simulator implemented on top of the SimpleScalar[5]. Wattch uses activity counters during the simulation in order to estimate the energy consumption of the different processor components. We integrate the Wattch power model in our simulator implementation using an improved conditional clocking for scaling the power dissipation of multiport structures. We model the energy consumption for all the main hardware structures of the processor (functional units, register files, branch predictor, queues, ROB, memory system, etc), including also the clock power estimation. With this accurate power model, we will give data of power and energy as result of several studies and different mechanism evaluations. We extend the power model during this research to obtain the power dissipation of the modified and new structures added to the processor hardware.

Before running the simulation, Wattch calculates the basic energy of every processor component consumes when it is fully utilized. Different estimation formulas, which are taken from CACTI¹, are used for each type of component. The main processor units that Wattch models fall into four categories:

1. Array Structures: Data and instruction caches, cache tag arrays, all register files, register alias table, branch predictors, the reorder buffer, and large portions of the issue queue and the load/store queue.

¹The original Wattch implementation uses an earlier version of Cacti whose code is built directly into the program. In our implementation, we run a separate and updated Cacti executable (version 4.2) and retrieve the data from it.

2. Fully Associative Content-Addressable Memories (CAM): Issue queue wakeup logic, load/store queue address checks, TLBs (if they are configured as fully-associative).
3. Combinational Logic and Wires: Functional units, issue queue selection logic, dependency check logic at decode stage, and result buses.
4. Clocking: Clock buffers, clock wires, and capacitive loads of the different structures.

During each cycle of the simulation, several counters keep track of the number of times each unit is accessed. Based on the number of accesses and the basic energy, the energy consumed during that cycle is calculated. For multi-port units, where only some of the ports are used, we implement an enhanced conditional clocking model. The power of these multiported units is scaled according to the number of ports used, with an unused port consuming 10% of the power a used port consumes (original Wattch uses the all-or-nothing approach).

Finally, the total energy consumed by the processor in function of each component is shown at the end of the simulation, along with the average power per cycle.

CACTI

CACTI is an independent tool which provides statistics such as timing, power and area model for cache memories. Although CACTI is theoretically modeled only for caches, we modify it to include also results for non-tagged similar structures (e.g. branch or data predictor tables). CACTI presents results in terms of area, energy consumption and delay broken down into the decoders, bitlines, wordlines, comparators, sense amplifiers, routing buses, output driver, etc.

Using CACTI, we estimate the access time for the cache configurations employed in our baseline. We also use this tool to estimate the energy of other structures and obtain additional information for our power consumption model, such as the number of bitline and wordline segments. This information is useful for Wattch to get the different hardware component features related to the energy.

2.2 Benchmarks

To compose the experimental workloads, we use independent set of programs from the SPEC2000 benchmark suite [64]. SPEC benchmarks, developed from real user applications, are used to measure the performance of the processor, memory and compiler on the tested system. This set of applications includes a wide range of codes with different behaviors both for integer (Spec INT) and floating point programs (Spec FP). In this sense, the experiments in the thesis are performed with multiprogramming workloads created by combining single-thread programs using both the integer and the FP benchmarks from that SPEC suite. Using multithreaded workloads composed of independent applications is still a frequently used technique by computer architecture researchers. Although these workloads are composed of non-cooperative applications that perform non-related work and do not communicate each other, they emulate real system workloads to properly perform multithreaded evaluations. In spite of the increasing trend to use truly parallel applications, they are still less common in real machines. We consider that simulation methodologies for parallel applications are a really interesting topic for future research, but it is out of the scope of this work.

All benchmarks are compiled to obtain Alpha standard binaries on a DEC Alpha AXP-21264 running UNIX V4.0 OS and using Compaq C/C++ V6.2-034 and f77/f90 Compaq Fortran V5.3-915 compilers with the -O3 optimization level and non_shared flag. For each benchmark, we select an interval of 300 million instructions representative of the entire program execution using the reference input set. To identify the most representative simulation point, we analyze the distribution of basic blocks using SimPoint[60] during benchmark executions. Table 2.2 presents the different SPEC benchmarks² with the reference input files used to run them, and the number of fast forwarded instructions for each benchmark. The last column shows the global memory miss rate for each program as well.

We characterize each program simulated in a single-threaded processor in order to classify the group of benchmarks and create the multiprogramming workloads. To make this classification, we use the IPC and the global cache miss rate. The IPC allows benchmarks to be classified into high-ILP and low-ILP according to their single-threaded IPC. The global miss rate is calculated with respect to the number of misses

²We have not include two SPEC FP (sixtrack and facerec) due to syscall errors in order to generate their execution checkpoints.

	Benchmark	input	Fast-forward (Millions)	Global miss rate (%)
Spec INT	164.gzip	graphic	68.100	0,08
	175.vpr	place	2.100	1,12
	176.gcc	166.i	14.000	0,07
	181.mcf	inp.in	43.500	14,97
	186.crafty	crafty.in	74.700	0,02
	197.parser	ref.in	83.100	1,05
	252.eon	cook	57.600	0,01
	253.perlbmk	splitmail.535	45.300	0,03
	254.gap	ref.in	79.800	0,33
	255.vortex	lendian1.raw	58.200	0,11
	256.bzip2	inp.program	13.500	0,10
	300.twolf	ref	324.000	1,23
Spec FP	168.wupwise	wupwise.in	263.100	0,92
	171.swim	swim.in	47.100	7,17
	172.mgrid	mgrid.in	187.800	0,88
	173.applu	applu.in	10.200	3,55
	177.mesa	frames1000 + mesa.in	294.600	0,14
	178.galgel	galgel.in	175.800	0,75
	179.art	c756hel.in,a10.img	13.200	14,75
	183.equake	inp.in	27.000	4,67
	188.ammp	ammp.in	13.200	0,95
	189.lucas	lucas2.in	30.000	7,03
	191.fma3d	fma3d.in	10.500	0,01
	301.apsi	apsi.in	192.600	0,34

Table 2.2: SPEC2000 benchmarks characterization

and accesses per each cache (Dcache and L2 cache). As we can observe in the table, floating-point benchmarks are more prone to have high miss rates due to their larger data sets. Those benchmarks with a global cache miss rate higher than 1% (one main memory access per 100 cache accesses) have a bad cache behavior, that is, they are considered memory bounded (MEM), and the others are considered CPU bounded (ILP). Therefore, the benchmark classification to compound the different workloads in are:

- Spec INT
 - [ILP]: gzip, gcc, crafty, eon, bzip2, gap, vortex, bzip2
 - [MEM]: vpr, mcf, parser, twolf
- Spec FP
 - [ILP]: wupwise, mgrid, mesa, galgel, ammp, fma3d, apsi
 - [MEM]: swim, applu, art, equake, lucas

Finally, in function of this characterization, we group them into three types of multiprogramming workloads: high instruction-level parallelism threads (ILP), memory-bound threads (MEM), and a mixture of both groups (MIX). Table 2.3 and 2.4 show our simulation workloads identified by the thread types (ILP, MIX o MEM) and the number of benchmarks they contain.

Table 2.3: Workloads of 2 threads used in this thesis

ILP2	MIX2	MEM2
apsi,eon	applu,vortex	applu,art
apsi,gcc	art,gzip	art,mcf
fma3d,gcc	equake,bzip2	art,vpr
bzip2,vortex	bzip2,mcf	art,twolf
fma3d,mesa	galgel,equake	equake,swim
gcc,mgrid	lucas,crafty	mcf,twolf
gzip,bzip2	mcf,eon	parser,mcf
gzip,vortex	swim,mgrid	swim,mcf
mgrid,galgel	twolf,apsi	swim,vpr
wupwise,gcc	wupwise,twolf	twolf,swim

Several studies [33][34][78] have shown that SMT performance saturates with more than 4 threads on the same core because of limitations in the shared instruction queue, fetch throughput or contention in the L2 cache. Likewise, it is not clear at literature that real implementation issues allow more than 4 active threads in a single core, the implementation complexity grows dramatically with more than four threads. Therefore, we consider only workloads composed of 2 or 4 benchmarks to create the multithreaded workloads. However, note that we create large groups of workloads to avoid result deviation due to the specific behavior of a particular workload. In total, we simulate

Table 2.4: Workloads of 4 threads used in this thesis

ILP4	MIX4	MEM4
apsi,eon,fma3d,gcc	ammp,applu,apsi,eon	art,mcf,swim,twolf
apsi,eon,gzip,vortex	art,gap,twolf,crafty	art,mcf,vpr,swim
apsi,gap,wupwise,perl	art,mcf,fma3d,gcc	art,twolf,equake,mcf
crafty,fma3d,apsi,vortex	gzip,twolf,bzip2,mcf	equake,parser,mcf,lucas
fma3d,gcc,gzip,vortex	lucas,crafty,equake,bzip2	equake,vpr,applu,twolf
gzip,bzip2,eon,gcc	mcf,mesa,lucas,gzip	mcf,twolf,vpr,parser
mesa,gzip,fma3d,bzip2	swim,fma3d,vpr,bzip2	parser,applu,swim,twolf
wupwise,gcc,mgrid,galgel	swim,twolf,gzip,vortex	swim,applu,art,mcf

30 2-thread workloads (10 per each category) and 24 4-thread workloads (8 per each category).

2.3 Evaluation methodology and metrics

Several methodologies [40][43][77][79] and metrics [28][44][63][75] have been proposed in order to evaluate multithreaded architectures. For this purpose, we select FAME [79] as our evaluation methodology since it provides more accurate measurements than previously used methodologies. This evaluation methodology re-executes all programs in a multithreaded workload until all of them are fairly represented in the final measurements. With this methodology, we ensure that all threads belonging to a workload are continuously in execution. In addition, any metric can use the measurements obtained with FAME to obtain fair final results among different workloads.

2.3.1 Performance metrics

We use the two most commonly used metrics for the SMT performance evaluation. One is the *throughput* [75], an execution-rate metric that measures the global multithreaded system performance. Throughput is frequently used as performance metric for evaluating multithreaded hardware executing multiprogramming workloads to reflect the total IPC from computer architect point of view. Total IPC is a valid metric in our case because the processor executes exactly the same instruction count for each

thread is used in all experiments. This metric consists of the average sum of IPC of all running threads in a workload:

$$Throughput = \frac{\sum_{i=1}^n IPC_{MT,i}}{n} \quad (2.1)$$

The second metric is the *harmonic mean* (Hmean) of the individual thread speedups proposed by Luo *et al.* [44]. This metric represents the fairness-performance balance which ensures that we are measuring real rate of performance speedup through the entire workload. It is calculated as the harmonic mean of IPC speedup of each thread in the multithreaded workload compared to its single thread performance:

$$Hmean = \frac{n}{\sum_{i=1}^n \frac{IPC_{ST,i}}{IPC_{MT,i}}} \quad (2.2)$$

with $IPC_{MT,i}$ and $IPC_{ST,i}$ being the IPC for thread i in multithreaded and single-threaded mode respectively, and n being the number of threads.

Both throughput and fairness are important when using an SMT processor. Whereas higher throughput ensures higher utilization of processor resources, fairness ensures that all threads are given equal opportunity and that no thread is forced to starve. Hmean of individual speedups is a single metric that encapsulates both throughput and fairness.

2.3.2 Power and efficiency metrics

During our work, we focus on reducing an important factor as power consumption. Different metrics related with power-efficiency have been proposed to compare different schemes [4]. Depending on what the constraints are, different metrics that measure power and performance together should be used. Regarding power studies in which the execution time is also important, the more appropriate metrics to characterize the power-performance efficiency of a processor are energy-delay related formulas. The metric of choice depends on the target market, which is determined by the kind of processor, the program being executed (multithreaded workloads in our case) and the power/cost ratio. For low-end or mobile processors the *energy · delay* (ED) formulation is more appropriate. To evaluate high performance machines, it may be appropriate the use of *energy · delay*² [4] (ED²) formulation, since the extra delay factor ensures a greater emphasis on performance versus power. The ED² is a power-performance char-

acterization metric that is voltage invariant, since it is derived from constant voltage-scaling arguments. For our studies, we alternatively use $(CPI)^3 \cdot W$ (the cube of cycles per instruction times power) as a reasonable metric for energy efficiency at the micro-architectural level. This is a direct and equivalent ED^2 metric applicable on a per-instruction basis. In this formula, delay refers to average execution time per instruction in cycles. We obtain the CPI, power and energy results from our integrated SMTSIM-Wattch simulator. A micro-architecture with a better ED^2 ratio will be higher performance than the other approach at the same energy, or use less energy at the same performance level.

Runahead Threads

As we explained in Chapter 1, large memory latencies and shared resource conflicts continue being the most important drawbacks for Simultaneous Multithreaded processor performance. The memory wall problem causes additional important difficulties due to the thread interactions in these shared resource processors. The critical case happens when threads with poor cache behavior (memory-bound or memory-intensive threads) are executed in the processor. A memory-intensive thread suffers from phases or periods with several cache misses and the subsequent memory accesses. A single thread of this type with poor cache performance can strangle overall SMT performance, by monopolizing resources that could be exploited by other threads. This type of threads can block the reorder buffer, because following dependent instructions can neither issue nor commit while waiting for long-latency memory accesses. During this period, this slow thread may have already allocated a lot of critical resources, starving other threads of required resources and preventing their forward progress. This effect would likely lead to global performance degradation in SMT processors.

We tackle these problems by *Runahead Threads*. Runahead Threads is an alternative mechanism to solve the memory wall and resource contention problem caused by memory-intensive threads but without restricting their performance. The overall idea of Runahead Threads is to transform a memory-intensive thread (that is, a possible problematic thread from resource and performance point of view) into a useful speculative thread by runahead execution. During all this chapter, we describe in detail this novel mechanism with its different features and implementation details on an

SMT processor model. We also show the advantages of this mechanism by performing different evaluations and analysis on the multithreaded scenario.

3.1 The idea of Runahead Threads

Runahead Threads mechanism arises as an alternative solution to alleviate the resource contention in SMT processors with the additional ability of exploiting MLP and improving performance at the same time. Programs with high MLP tend to have bursts of long-latency loads in the dynamic instruction stream in several phases of program execution[12]. This characteristic of loads makes the exploitation of MLP feasible while another long-latency is resolved. With most of the proposed SMT policies (described in Chapter 6), memory-intensive threads are stalled or restricted while they are waiting for an L2 cache miss to be serviced from main memory. The purpose of Runahead Threads is to utilize those idle cycles that are wasted due to SMT policy stalls for performing useful work by speculative executions of the program. The premise is that this speculative mechanism lets the thread, that suffers from a long-latency load, fetch and execute short-latency instructions that other mechanisms do not permit to exploit that MLP. The final goal is to improve the main memory latency tolerance without disturbing the execution of the rest of threads, thereby reducing thread resource contentions.

Fruit of this idea, we propose a new utilization of the Runahead execution on SMT processors. Runahead execution has the features and requirements that we look for our goal. This technique matches the idea of having a different speculative policy that improves the performance of memory-bound threads. Runahead technique allows the processor throw a long-latency memory instruction out of the instruction window. This action eliminates the need for allocating and holding resources for the long-latency operation and the instructions depending on it (such as issue queue entries, renaming registers, load/store buffer entry and so on). As consequence, it avoids the resource monopolization by this thread and creates space in the hardware resource pool to speculatively process operations independent of the long-latency operation. The novelty of this mechanism regarding previous SMT techniques is that the memory-intensive threads advance speculatively, instead of being stalled, doing useful processing to improve their performance without disturbing the other threads.

Runahead Execution

Describing a brief overview of Runahead execution is a good starting point to begin with. Mainly, Runahead prevents the processor from stalling on long-latency memory requests and it executes speculative instructions in the meantime. When the processor detects that a memory operation that missed in the L2 cache reaches the head of the reorder buffer (i.e. it is the oldest instruction), a checkpoint of the architectural state is taken. After that, the processor assigns an invalid mark (as not available value) to the destination register of the memory instruction that caused the L2 miss and removes this long-latency instruction from the instruction window. Next, it enters a speculative processing mode called *runahead mode* and then allows the following instructions to pseudo-commit.

During runahead mode, the processor continues speculatively executing instructions without blocking retirement due to L2 cache misses and the instructions dependent on them. The results of L2 cache misses and their dependents are identified as invalid. Instructions that operate over an invalid value will be considered invalid as well. Then, the invalid instructions are removed from the instruction window so that they do not prevent independent instructions from being placed into the pipeline. These instructions that do not depend on an invalid value are executed as normal, except that they do not update the architectural registers and memory state. All these runahead speculative instructions are pseudo-retired in program order at commit stage. This pseudo-retirement in runahead is very similar to retirement in normal mode except updates to the architectural state are not allowed. Therefore, runahead mode allows the processor to execute instructions (independent of L2 cache misses) that may miss in the memory system (instruction, data, or unified caches). Consequently, their miss latencies are overlapped with the latency of the cache miss that caused entry into runahead mode (i.e., runahead-causing cache miss).

When the memory access that started runahead mode completes, the processor needs to roll back to the previous state and resumes normal execution. The processor exits runahead mode by flushing the instructions in its pipeline. It restores the checkpointed state and resumes normal instruction fetch and execution starting with the runahead-causing instruction. As a consequence, all the speculative work done by the processor is discarded. Nevertheless, this speculative execution has not been useless since some of the subsequent data and instructions needed now during normal mode

are brought to the caches earlier as result of prefetching during runahead mode. Thus, once the processor returns to normal mode, it is able to make faster progress due to this effective prefetching. Hence, runahead execution overlaps idle clock cycles due to L2 misses to speculatively execute the application in order to generate accurate prefetch requests.

3.2 Fundamentals of Runahead Threads

Runahead Threads applies a multithreaded version of Runahead execution to any running thread that undergoes a long-latency load. When the SMT processor detects that the oldest instruction for a particular thread is waiting for an L2 cache miss, it turns the thread into a *runahead thread*. While being a runahead thread, this thread behaves as a fast speculative thread. It uses the different needed resources for executing its speculative instructions without blocking the available resources for other threads. At the same time, the issued prefetches during this speculative execution increase the memory-level parallelism, improving the own thread performance. In other words, our approach transforms a hoarder resource thread, such memory-intensive thread, into a light thread with fast instruction stream execution. So, Runahead Threads allows the memory-intensive threads to advance speculatively to do useful processing instead of stalling for several cycles due to resource contention.

The runahead threads improve the thread-level parallelism by removing long-latency memory operations from the instruction window, releasing faster the important shared resources and avoiding critical resource blocking among multiple threads. As we will see, the resource contention using RaT is reduced to similar ratios as when different fast computing-intensive threads are being executed in the processor. The important advantages of this speculative mechanism are: 1) to prefetch data using a minimum amount of hardware resources and 2) does not clog the shared resources.

We next describe in detail the operation of Runahead Threads once we have previously presented the basic idea and an overview of our approach. For this purpose, we identify three phases of RaT operation that we will explain in the following sections.

- Starting a runahead thread.
- Executing a runahead thread.
- Exiting a runahead thread.

3.2.1 Starting a runahead thread

Applying Runahead Threads on an SMT processor, a thread in a hardware context can operate in two different modes or states: normal mode or runahead mode. According to each mode, we have a *normal thread* or a *runahead thread* respectively. A normal thread turns into runahead thread when a load instruction belonging to that thread misses in the second-level cache (or off-chip last level cache) and that memory operation reaches the head of the reorder buffer for that hardware context. In our design, we assume that an L2-miss store instruction does not block retirement due to the use of the store buffer. However, in the case there is a load that depends on a previous long-latency store, this load can activate a runahead thread because it behaves as a long-latency memory operation as well.



Figure 3.1: A runahead thread is started when a long-latency load is detected

RaT takes the following actions to start the runahead thread when it detects that the previous condition for activating runahead thread is satisfied. The processor checkpoints the architectural state of the corresponding hardware context to correctly recover that state on returning from runahead thread to normal thread. This checkpoint saves the state of the architectural register file, together the branch history register and the return address stack. Furthermore, the address (program counter) of the load instruction that causes start the runahead thread is recorded to know the instruction where to return in normal mode. Next, the load causing the runahead thread is invalidated and retired from the ROB to proceed with the runahead speculative execution. With this simple action, we are avoiding two critical effects that a possible offending thread causes: (1) a possible reorder buffer blockage and (2) a hardware resource monopolization.

3.2.2 Executing a runahead thread

Once a thread becomes a runahead thread following the process just described, any instruction that is fetched in this thread is marked as a runahead instruction. That is, all instructions from this thread in the instruction window are identified as “runahead operations” until runahead thread finishes. The execution of instructions in a runahead thread is very similar to instruction execution in a normal thread. The differences are that the execution of runahead instructions is purely speculative and they do not update the architectural state.

The main issue involved in the execution of runahead instructions is the tracking of L2-miss instructions and the control of their dependents. During a runahead thread execution, we can distinguish two kinds of instructions depending on the validity of their source operands: valid and invalid instructions. An invalid instruction is any instruction that references a source register whose value depends on the dependence chain of the load instruction that causes the runahead thread (values not available - invalids). Thus, an invalid instruction is not executed and it is directly driven to the retirement. In case of a valid instruction, it is executed and updates its physical destination register with the corresponding result depending on the operation performed. Finally, both kind of instructions pseudo-retire in program order at the commit stage.

In this execution, the first instruction that introduces an invalid (bogus) value is the load instruction that causes to start the runahead thread. When this load is retired to start the runahead thread, this instruction marks its destination register as invalid (INV). In the same way, each valid L2-miss load executed in the runahead thread is marked INV as well although its corresponding memory access is performed. This action prevents the retirement logic from blocking for these speculative long-latency instructions. Next, whatever instructions that requires an INV register (i.e., an L2-miss dependent instruction) is marked as invalid (INV) together with its destination register. In this sense, conditional branches that are marked as invalids are really not resolved. Since the value for checking the test condition of an invalid branch is not available in the runahead execution, the processor relies on the prediction of the branch predictor for that branch rather than using a bogus stale value to resolve the branch.

The characteristics of valid and invalid instructions have some important and interesting benefits in multithreaded scenarios which we describe next and we summarize in Figure 3.2:

- When a thread is turned into a runahead thread, the invalid instructions almost do not use processor resources since they are immediately pseudo-retired. INV instructions are not executed and can be removed from the instruction window without waiting for the completion of the L2 miss they are dependent on. Then, they do not use functional units, registers, issue queues and so on. This reduces the resource requirements for runahead threads and allows other threads to make forward progress without suffering from resource starvation conditions.
- The other valid long-latency loads are also invalidated just like the load that started the runahead thread, but the memory references of these instructions are issued to the memory system. The L2 miss requests to main memory generated by these runahead memory operations are treated as prefetch requests. Besides, load and store instructions which are already invalids are not allowed to generate memory requests since their addresses would be bogus. This reduces the probability of polluting the caches with bogus memory requests. In addition, these invalid memory instructions do not saturates the memory bandwidth and cache ports because they do not used.
- The rest of the valid instructions executed in the runahead thread are usually short-latency instructions that quickly use the different shared resources during their execution. Valid instructions allocate and de-allocate the resources faster than long-latency instructions that are not explicitly execute in a runahead thread.

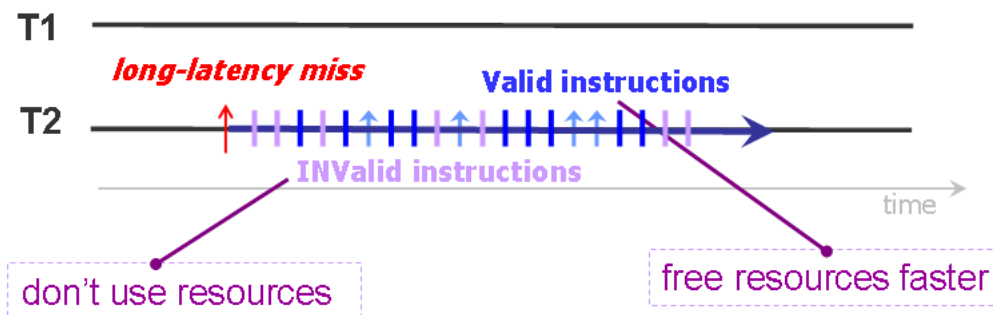


Figure 3.2: Type of instructions while a runahead thread is executed

As result of this kind of speculative execution, runahead threads behave as fast threads with low resource requirements. In this sense, runahead threads are much

less aggressive than normal threads (especially memory-bound ones) with the valuable processor resources, allocating and deallocating them in short periods of time. Besides, the issued prefetches in runahead mode increase the single thread performance by exploiting the memory-level parallelism as we will show later.

3.2.3 Exiting runahead thread

A runahead thread exits when the runahead-causing L2 cache miss is resolved. This point is not a fixed interval of time, since the cycles required to service an L2 cache miss is variable depending on bank conflicts, port contentions, signal delays, etc. In our memory model, an L2 cache miss has a latency of 300 cycles, that represents the minimum number of cycles.

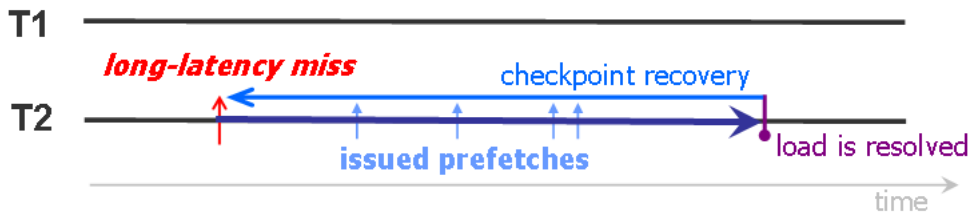


Figure 3.3: Runahead thread recovery

To turn back a runahead thread to a normal thread, we need to recover the hardware context state before starting the runahead thread. This requires a pipeline flush in which all runahead instructions belonging to that thread are discarded. So, all runahead instructions of this hardware context are flushed and the resources allocated for them are deallocated. Next, the context architectural state is restored from the corresponding thread architectural registers, branch history register and RAS checkpoint. Finally, the thread is returned to normal mode. From this point onwards, the thread starts fetching normal instructions starting with the instruction that caused the runahead thread.

3.3 Implementation of Runahead Threads

The translation of Runahead Thread operation to an actual implementation of the mechanism does not introduce any complexity in the design of a multithreaded processor. This section describes these implementation issues and addresses the design

trade-offs that need to be considered when integrating the Runahead Threads mechanism in an SMT processor. The final implementation details of Runahead Threads can be slightly different among particular multithreaded architectures, but the basic mechanism is applicable to anything taking into account the relevant design details. RaT implementation issues are common to all hardware context, since it treats them equally from the mechanism operation point of view. Next, we discuss the design trade-offs and required modifications regarding the three phases of runahead thread operation.

3.3.1 Implementation issues for starting a runahead thread

To decide to start a runahead thread, the processor has to know when a load instruction that is in the head of the reorder buffer has a long-latency access pending. To detect that an instruction missed in the L2 cache, the processor simply needs to control the miss signal from the L2 cache controller and associate this information with the memory instruction. With this information and when that instruction reaches the head of the ROB, the processor can turn a normal thread to a runahead thread. One bit per context is used (called runahead thread bit) to distinguish the thread state. Before a runahead thread starts, the corresponding runahead thread bit is set to identify that hardware context enters in runahead execution.

Maybe, the most important component that makes possible runahead threads is the checkpoint. Checkpoint mechanisms are needed to support precise state reconstruction and guarantee correctness of the execution. The mechanism for checkpointing the architectural state depends on the specific microarchitecture related with the multithreaded processor. In this sense, most of the hardware needed to checkpoint the state of the architectural register file and return address stack already exists in modern out-of-order processors to support recovery from branch mispredictions. For example, some processors checkpoint the architectural register map after renaming a branch instruction and the return address stack after fetching a branch instruction. A straightforward solution can be to use the periodic checkpoints of these conditional branches and recovering the thread state to the nearest branch to the load causing runahead thread. Another more accurate solution is to take similar checkpoints when the oldest instruction is an L2-miss load. Therefore, RaT only requires modifications to extend the misprediction branch checkpoint model for loads as well.

The processor architecture state to be recorded for enabling Runahead consists of: register mapping, the return address stack, the branch history register and some machine state registers. In our SMT model, with up to four hardware contexts, each hardware context has its own architectural registers and requires four independent register mappings, one for each thread. For architectures that use front-end and commit register renaming tables, a copy of the full physical register file is unnecessary. This full register file checkpoint would include the register information of all threads, taking long time and making it much more complex. Therefore, to correctly recover the own architectural state, each thread needs to checkpoint only its renaming map table, so that an older state can be saved and later restored. In this case, we simply restore an older mapping of logical to physical registers for a particular context once a particular runahead thread has finished.

Usually, content addressable memory (CAM)-based tables are used to hold the register mapping state. Both the map tables and the checkpoint of them could be physically realized with a single map table for each one by appending a two-bit thread identifier associated with a fetched instruction to the the logical register codes (5-bit in our model) extracted from the instruction itself. So, thread context 0 would use mapper logical registers 0 through 31, thread 1 would use mapper logical registers 32 through 63 and so on. In this scheme each quadrant of the mapper CAM would have the capability to be independently backed up in the corresponding checkpoint quadrant and restored as needed to maintain the correct processor execution. With this possible implementation, the number of transistors to store the context architecture states is a small fraction of the total of the processor.

3.3.2 Implementation issues for runahead thread execution

The main issues involved in execution of runahead threads are the treatment of the different runahead instructions and the propagation and communication of the INV results. Here, we describe the hardware required and other important factors related to the SMT scope to support this new functionality.

Register validation control

The control mechanism that communicates data between dependent instructions is already present in the processors (wake-up and select logic). Therefore, INV bits can

be propagated in the datapath with the data they are associated with. In the case of an SMT environment, as the register file is shared, we only need an INV bit associated to each physical register to track the propagation of register invalidations. This bit indicates whether or not a register has a bogus (INV) value during a runahead thread execution. INV bits are used to prevent bogus prefetches and resolution of branches using bogus data. In case of an invalid one, its value is not available for the runahead and the corresponding invalid instruction is not executed. Any INV instruction marks its destination register as INV after it is scheduled. Any valid operation that writes to a register resets the INV bit associated with its physical destination register. So, when a physical register is invalid (INV bit set to 1) this would be released soon and to be used for the same thread to continue exploiting MLP or for the rest of threads.

Load and store management

Similar to the registers, Runahead execution does not require significant hardware to handle memory data invalidations. The runahead store and load dependencies can be done through the store buffer. The forwarding of INV bits from the store buffer to a dependent load is accomplished similarly to the forwarding of data from the store buffer. For Runahead Threads, we need to add only one more bit (the INV bit) per entry in this structure and in the forwarding data path. When the address of a memory operation is INV, they are simply treated as a No Operation (NOP).

Mutlu *et al.* [51] introduce the runahead cache to provide extra communication of data and invalid status between runahead loads and stores for runahead execution in out-of-order processors. This structure holds the results and INV status of runahead stores that have already pseudo-retired. Based on this information, some loads dependent on stores can be identified as valid or invalid. Nevertheless, there are other cases in which this memory dependency cannot be identified. For example, a store that has an invalid effective address cannot save its status or data in this runahead cache.

From the SMT point of view, using a runahead cache results expensive in terms of extra hardware. In a multithreaded processors the runahead cache needs to be larger to avoid aliasing and line contention among threads. Likewise, it is necessary to include a new identification tag to distinguish the runahead cache block owner for each thread. So, the runahead cache would be a large structure required by runahead execution in this multithreaded framework. However, we measure the performance with and without the runahead cache to consider the need to include it in the final

Runahead Threads mechanism implementation (see Section 3.6.4). We will show that using the runahead cache does not have significant impact on performance in our SMT model. Based on this result and the fact that a runahead cache implies the use of more area in the SMT core, we decide not to use it in our RaT implementation. As runahead threads are purely speculative, there are no strict consistency requirements to satisfy correct propagation of data between store instructions and dependent load instructions. Hence, the functional difference is that some loads dependent on previous retired stores that were not identified as invalids, will use bogus values for speculative memory accesses, but it just affects runahead execution, i.e., it does not affect correct program execution.

Floating-point resources

The performance improvement of Runahead Threads mainly comes from the pre-execution of memory operations. Generally, the computation of the address for memory operations involves a base register plus an offset. This is an integer arithmetic operation, so floating-point (FP) operations are not needed to compute the effective addresses. According to this observation, we can decrease the resource demand of runahead threads by avoiding the execution of FP instructions in the RaT mechanism.

This modification was considered for runahead execution in out-of-order processors [50]. We also apply it in our implementation for an additional benefit in the SMT scenario. If a runahead thread does not execute FP instructions, it does not need the floating-point resources of the SMT processor. So, once an instruction is detected to be an FP operation in the decode stage, it is invalidated and directly proceeds to pseudo-commit. With this implementation, FP instructions in a runahead thread do not use any processor resources after they are decoded. Therefore, the FP issue queue, the FP functional units, and the FP physical register file are not used by most FP runahead instructions. The exceptions are FP loads and stores, which are treated as prefetch instructions because their effective addresses are obtained in the integer datapath.

Synchronization

Finally, an important issue in the context of multithreaded processors is that there can be both independent and parallel programs in execution. Independent multiprogramming workloads do not need any synchronization, since they are threads executing

different programs or belonging to different users. On the contrary, parallel programs normally use a scheme that allows threads to synchronize each other to use the shared memory. Usually, this is made by lock operations (basically through block, acquire, and release special instructions) which force to atomically use data from the memory. The basic mechanism relies on serializing the operations to ensure that critical sections are not executed concurrently by different threads in parallel programs.

As speculative execution, runahead threads cannot make any changes to the lock variable or to the critical data protected by the lock variable, thereby avoiding the inconsistency among parallel threads. In the case that a parallel thread switches to a runahead thread, these lock instructions are ignored, since the runahead thread does not need to obey the atomicity semantics. The instructions inside the critical section are speculatively executed because they do not modify program state and any critical shared data. This could enable faster progress in runahead executions without incurring the latency associated with lock operations.

3.3.3 Implementation issues for exiting from a runahead thread

When a runahead thread finishes, this requires a pipeline flush. The hardware needed for flushing the pipeline and restoring the checkpointed state is the same as the hardware needed for recovering from a branch misprediction. Therefore, branch misprediction recovery hardware can be extended for runahead thread exit without increasing processor complexity. The particular hardware required for restoring the checkpointed architectural registers depends on the checkpointing mechanism used.

3.3.4 Overall hardware cost and complexity

Figure 3.4 shows the additional hardware structures needed for RaT in the microarchitecture of our SMT processor model. The new components are shown in bold and they have been described in this section. Basically, the main hardware modifications consists of (1) a checkpoint structure to save the architectural registers, branch history register, and return address stack; and (2) a single invalid (INV) bit associated with every physical register and store buffer entry to control the runahead instructions and dependence validations.

As we show in Figure 3.4, implementation of RaT mechanism does not introduce any complications in the design of a multithreaded processor. None of these added

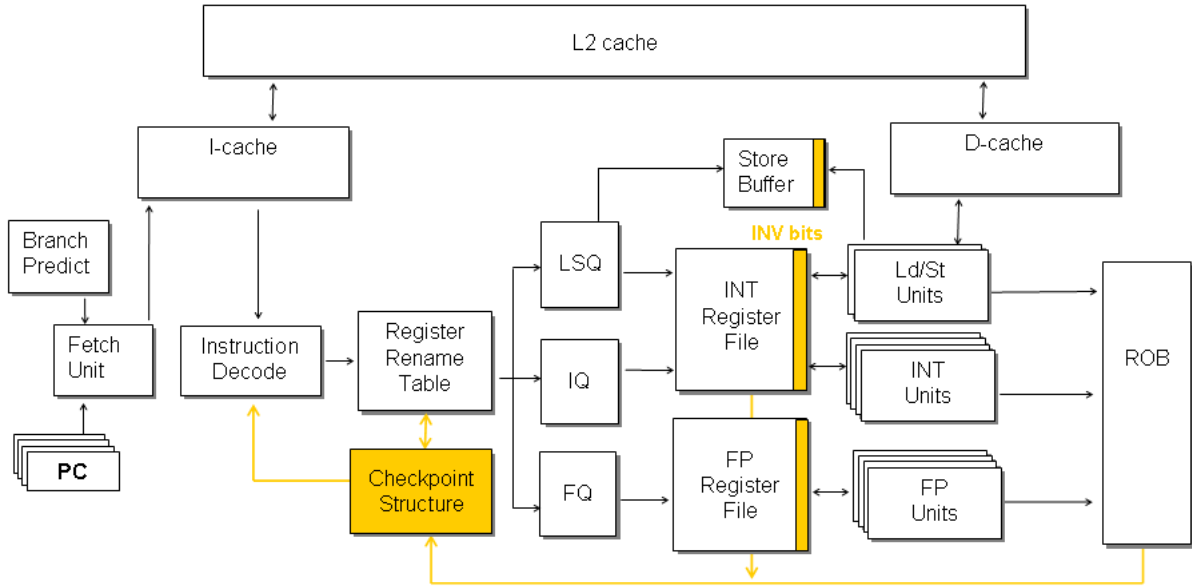


Figure 3.4: RaT processor architecture. SMT processor architecture remarking the modified and additional components for implementing RaT

structures are on the critical path of the processor or suppose increase the processor complexity. The most significant hardware component of RaT is the mechanism for checkpointing. Fortunately, it is not necessary to implement an additional new mechanism, since processors already incorporate checkpointing capabilities for branch mispredictions and interruption management. For this case, we only need to adapt and extend the mechanism to support checkpoints for long-latency loads as well. However, this depends on the particular microarchitecture implementation. In addition, it is possible to checkpoint the return address stack and BHR without significant hardware cost.

Therefore, RaT is a cost-effective choice that does not significantly increase SMT processor complexity. As we will show later, an SMT processor with RaT achieves a significant potential performance compare to the small changes required to implement it.

3.4 Evaluation of Runahead Threads

This section evaluates the Runahead Threads mechanism compared to our baseline SMT processor with ICOUNT. The evaluation of RaT is performed using the wide

variety of workloads on the SMT processor model described in Chapter 2. We provide the performance results and explain the different benefits of using RaT.

3.4.1 Performance of Runahead Threads

We compare the RaT performance results versus the baseline using the throughput and the Hmean metrics.

Throughput

First, we evaluate how RaT performs compared to the baseline processor with ICOUNT in terms of throughput. We show the overall SMT performance throughput for the baseline ICOUNT and RaT mechanism for every workload. Figure 3.5 shows the throughput for 2-thread workloads and Figure 3.6 shows it for 4-thread workloads grouped according to each workload category. There are three groups of bars in each figure, one for the ILP-intensive workloads (ILP), one for the mixed workloads (MIX) and one for the memory intensive workloads (MEM) respectively, each group divided by the corresponding average bar.

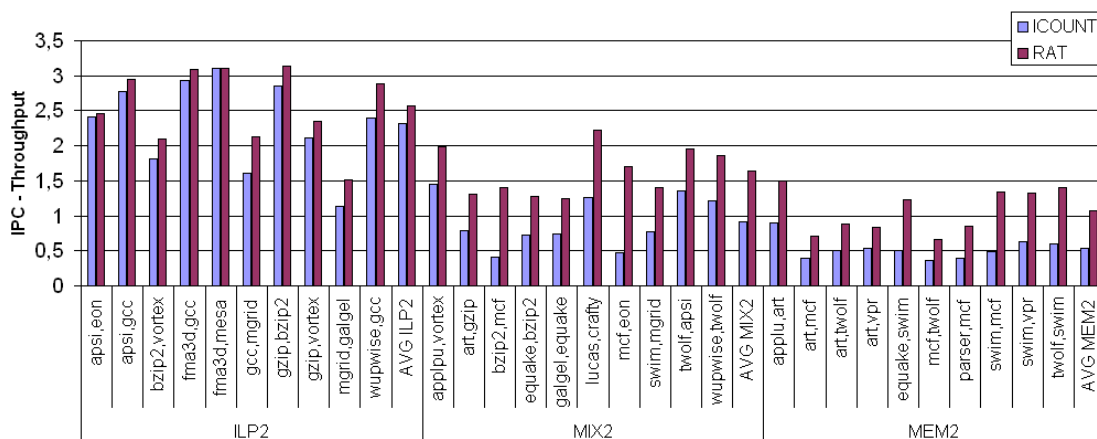


Figure 3.5: Runahead Threads throughput performance vs. SMT baseline for 2-thread workloads

These figures show as RaT gets higher throughput than ICOUNT for all kind of workloads. From Figure 3.6, the baseline throughput is higher than 2-thread hardware contexts because the greater number of executed threads for 4-thread workloads. They

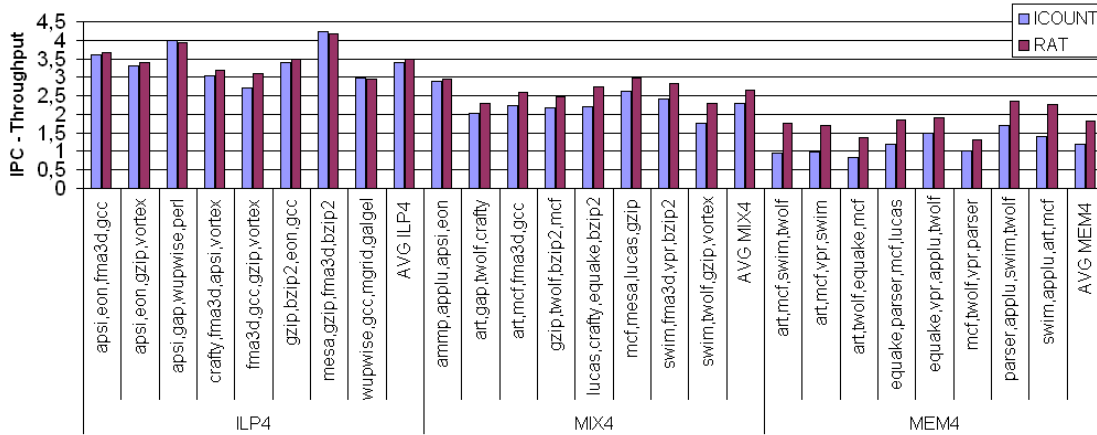


Figure 3.6: Runahead Threads throughput performance vs. SMT baseline for 4-thread workloads

also show that the influence of the runahead threads on the overall SMT throughput differs depending on workload characteristics. In the case of ILP workloads, the throughput difference among RaT and ICOUNT is lower than for the other two workload categories, MIX and MEM. ILP workloads do not contain memory intensive threads and then the prefetching benefits for them are more reduced. Even so, the average performance improvement is 11% for 2-thread ILP workloads and 2.5% for 4-thread ILP workloads.

For MIX workloads, the behaviour of workloads which include both memory bounded and high-ILP benchmarks is not the same as observed previously when only ILP benchmarks are executed. In this case, the performance of RaT is 78.6% and 15.1% better than ICOUNT for MIX2 and MIX4 on average respectively. On the one hand, RaT improves the performance of memory-intensive threads by prefetching. On the other hand, computing-intensive threads are also improved by eliminating the resource monopolization cases of the memory intensive threads.

Finally, RaT provides significant throughput improvements in the case of MEM workloads. As Figure 3.5 shows, RaT outperforms ICOUNT by 102.8% for MEM2 and 52% for MEM4. As we show later, these workloads benefits mainly by the prefetching effect thanks to the aggressive exploitation of the MLP.

Per-thread SpeedUp

In the previous figures, we show the total throughput performance of each workload entirely. Now, we study the performance speedup of each separate thread in the multi-programming workloads. That is, the IPC variation of RaT compared to ICOUNT for each benchmark in the multithreaded scenario. We compare the baseline performance (IPC) per each thread that composed the different workloads to the performance of the same thread under RaT mechanism execution.

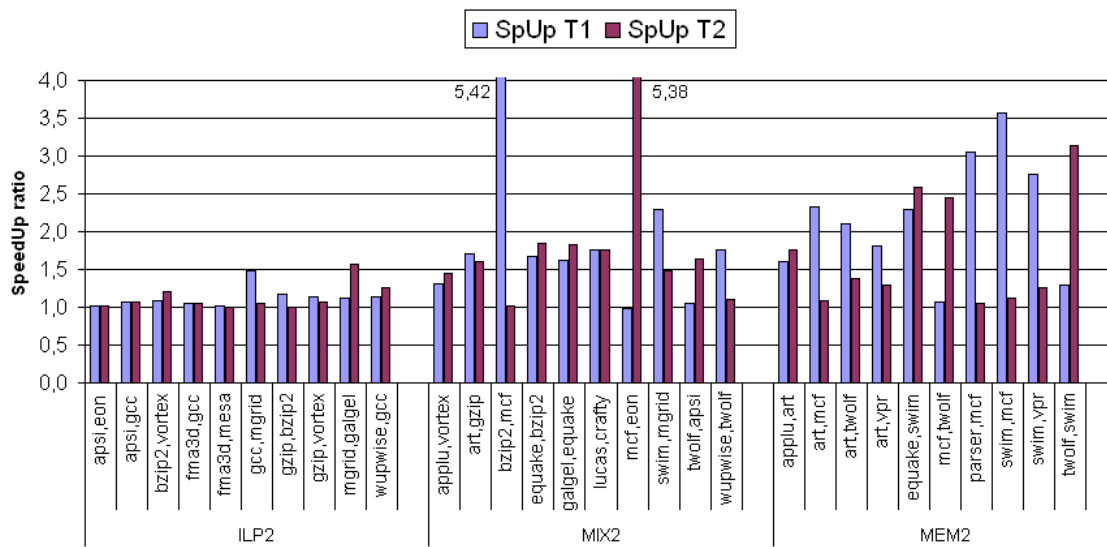


Figure 3.7: Individual thread speedup between baseline and runahead performance for 2-thread workloads

Figure 3.7 shows the speedup of RaT performance with regard to baseline ICOUNT for every individual benchmark from the 2-thread workloads. Each workload has two bars in the figure, one is the speedup corresponding to thread 1 and the other is the speedup of thread 2. The speedups for the ILP2 workloads are more or less uniform, only excelling some particular thread in some workload, such *gcc* or *galgel*. In the case of MIX2 workloads, we can see higher speedups per thread for both threads in almost all these 2-thread workloads. As we commented before, threads in MIX workloads get performance improvements in a cooperative way by RaT. We here demonstrate as the ILP thread performs better by the resource availability and the MEM thread improve its performance by prefetching. There is special cases for the workloads with the *mcf*. As far as *mcf* is concerned, this benchmark suffers from many loads which

are dependent among them, resulting more difficult to improve its own performance by runahead. In these cases, RaT eliminates the big periods of resource monopolization due to the instructions depending on the high number of long-latency loads of *mcf*. Therefore, RaT improves the resource availability to the other thread that compose the workload, thereby improving its performance (speedups around 5X).

Regarding the MEM2 workloads, the overall speedups are higher in comparison to the other workloads as the figure illustrates. In general, all MEM threads individually achieve performance improvements, being the minimum the 6% speedup of *mcf* benchmark for (*mcf,twolf*) workload and the maximum, the 3,5X speedup, for *swim* in (*swim,mcf*) workload. The ratio of these percentages depends on the ability of the executed runahead threads to issue in advance the different prefetches and to avoid the resource contention due to these memory intensive threads.

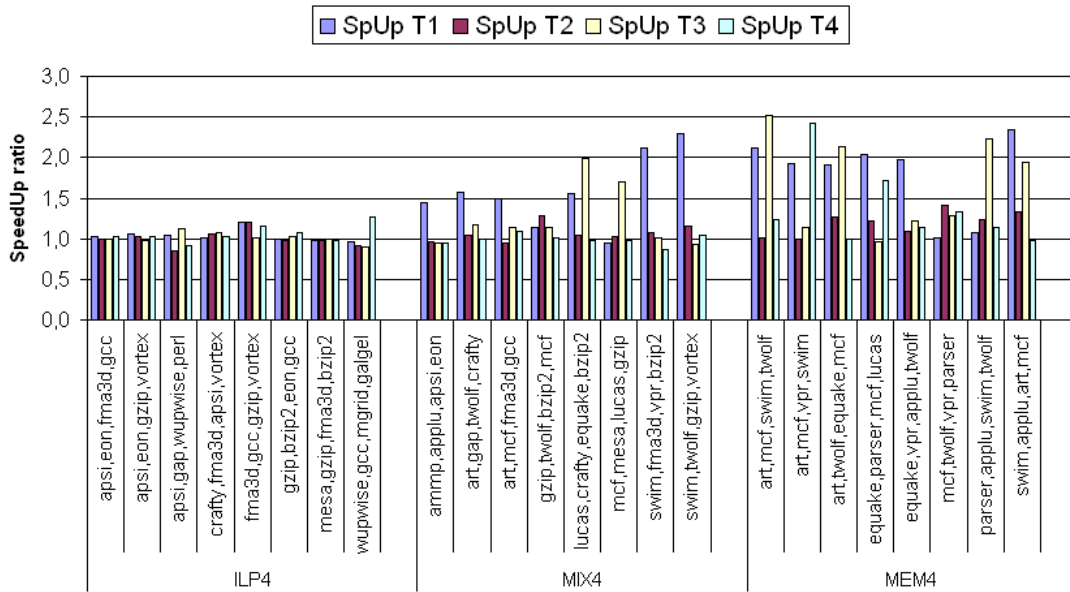


Figure 3.8: Individual thread speedup between baseline and runahead performance for 4-thread workloads

In the same way, Figure 3.8 shows the performance speedup for every thread for the case of 4-thread workloads. The improvement trend for these workloads is similar to 2-thread workloads, but with the performance speedup spreads over more threads instead of only two. There are few cases for 4-thread workloads in which some particular thread suffers for a slight performance slowdown, mainly for ILP4 threads, although the total workload throughput is compensated for the other thread speedups as we

show in Figure 3.6. This is caused because in absence of useful work in runahead, the speculative instructions of runahead threads can hinder the execution of normal ones for computing intensive benchmarks. However, this is not the general rule and RaT provides good speedups in most of threads, with ratios greater than 2X in benchmarks for MIX4 and MEM4 workloads.

Hmean

As other important factor to evaluate the Runahead Threads performance, we analyze the performance-fairness results using the harmonic mean of individual speedups (Hmean metric). We show the Hmean results for the baseline ICOUNT and RaT in Figures 3.9 and 3.10 for 2-thread and 4-thread workloads respectively. In these figures, a higher bar is interpreted as better.

These Hmean results confirm that the RaT mechanism presents a better throughput/fairness balance than the ICOUNT policy. In spite of computing-intensive benchmarks are not the objective of RaT, the gains in ILP workloads are not so outstanding but better than ICOUNT. The Hmean improvement over ICOUNT for ILP workloads is 12.5% for ILP2 and 2.4% for ILP4. The MIX and MEM type workloads contribute more to the whole average Hmean gains. For MIX workloads, RaT gets better ratios with 79% and 12.9% Hmean improvement over ICOUNT for 2-thread and 4-thread workloads respectively. Finally, Hmean metric indicates that RaT is also much more fair than ICOUNT from performance point of view for the MEM workloads, improving by 75.4% the Hmean for MEM2 workloads and by 42.9% for MEM4.

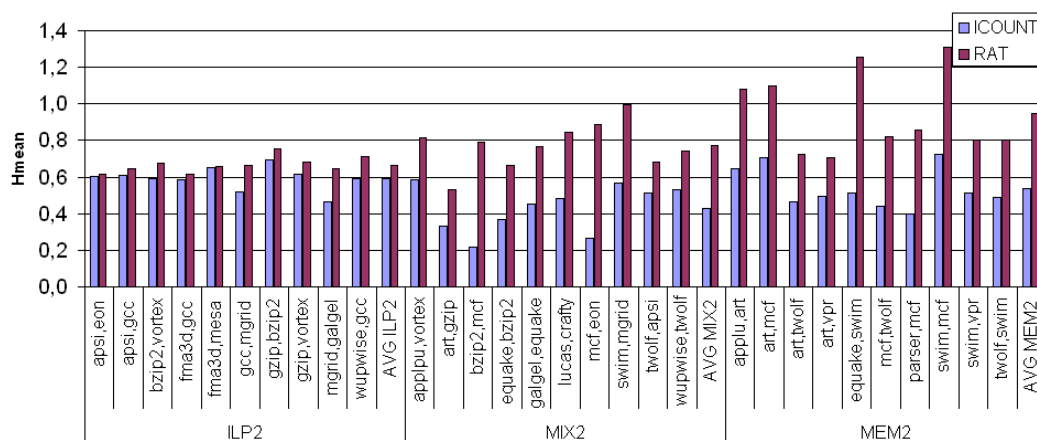


Figure 3.9: Hmean of Runahead Threads vs. SMT baseline for 2-thread workloads

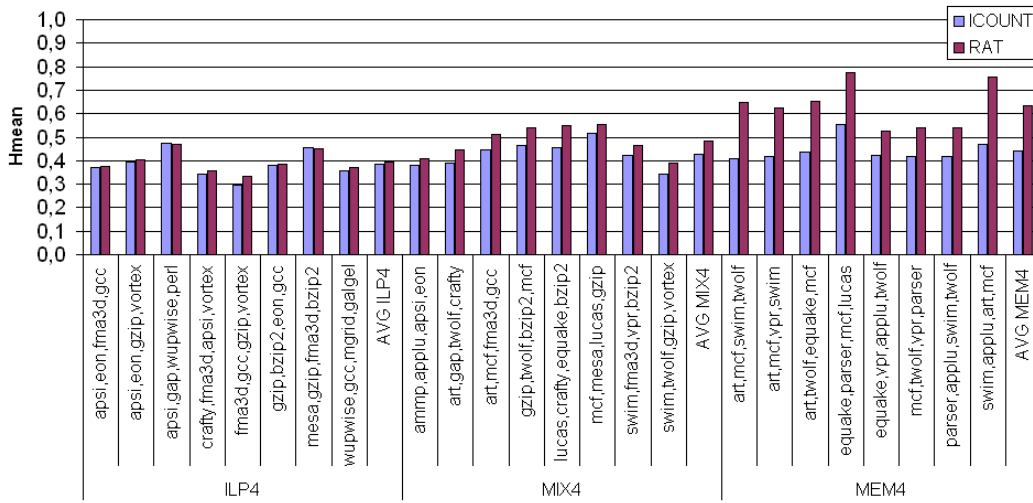


Figure 3.10: Hmean of Runahead Threads vs. SMT baseline for 4-thread workloads

Therefore, these results demonstrate that RaT provides good performance improvement as well as fairness, specially for memory intensive programs present in MIX and MEM workloads. RaT takes into account the fairness among threads, avoiding to favor only threads with high IPC, and boosting memory intensive threads (with low IPC) compared to ICOUNT.

3.4.2 Benefits and limitations of Runahead Threads

Once the performance of RaT has been evaluated, we make an analysis to distinguish the partial contribution of the sources of benefit provided by RaT. One interesting point to know is how threads are being improved by using Runahead Threads. This overall improvement comes from two distinct factors: (i) each thread itself is faster because of the prefetching effect via Runahead execution and (ii) Runahead Threads improve the overall performance because it does not block and release resources to other threads. The former exploits the memory-level parallelism whereas the latter reduces resource contention. Likewise, not all are advantages. We also expose the main limitation related to Runahead Threads due to the speculative instruction overhead.

Prefetching

The first sign of benefit due to prefetching is captured through a miss reduction in the caches. Figure 3.11 shows the average cache miss rates of the Dcache and L2cache

for the baseline SMT processor and RaT. In the case of RaT results, we compute the cache miss rate taking into account only the cache accesses by normal threads (no speculative ones), since normal threads perform the valid and actual memory accesses of the program executions. Overall, RaT reduces the miss rate of both Dcache and L2cache as Figure 3.11 shows. The workloads with a higher number of misses with the baseline processor get better benefits for the prefetching effect of RaT. For instance, the L2cache miss rate is reduced from 58.8% to 35.9% in MIX2 workloads and from 75.9 to 56.3% in MEM4 workloads.

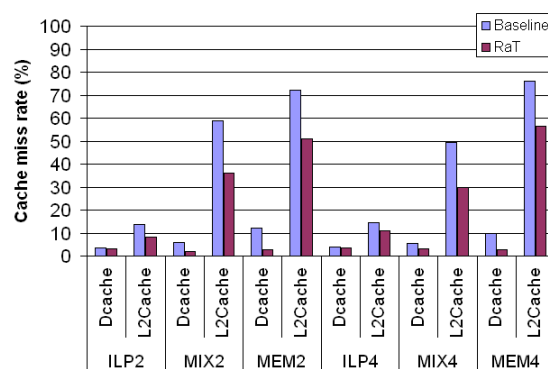


Figure 3.11: Dcache and L2cache miss rate reduction by RaT

To measure the effect of prefetching in terms of performance, we perform an experiment for that study. We compare the performance of proposed RaT mechanism with a modified version in which runahead threads do not issue prefetches. That is, threads effectively turn into runahead thread but they do not perform any access to the main memory during its execution. In addition, loads and branches are tracked during runahead thread to ensure that the runahead periods are the same in both normal and runahead without prefetching. So, L2 miss loads during RaT without prefetching will not switch again to runahead thread when they are encountered after recovering from Runahead.

Figure 3.12 shows the performance improvement of RaT compared to RaT without prefetching. According to these results, prefetching accounts, on average, for about 58.2% of the performance improvement. MIX and MEM workloads are the ones that benefit the most from this effect (56% and 109% respectively), as we also demonstrated before with the respective cache miss rate reduction.

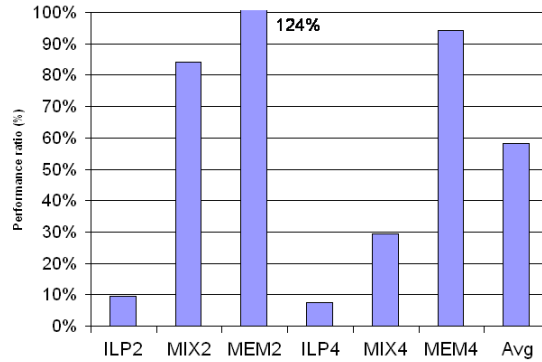


Figure 3.12: Prefetch improvement percentage by RaT

Resource contention

Using RaT, the resource availability comes from two sources. Firstly, most of invalid instructions during Runahead do not hold resources because they are not executed. Secondly, instructions executed in Runahead present short latencies, meaning that resources slots (e.g. registers or queue entries) are allocated for short periods of time. Measuring the contribution of the resource contention reduction individually to performance is much more difficult. There are a lot of resources and threads interaction during the execution. Nevertheless, to illustrate this benefit in some way, we count the relation between the percent of resources conflicts of the baseline compared to the percent of resources conflicts of RaT. Figure 3.13 shows the difference ratio of the different resource conflicts taken into account for all workload simulations. In this figure, we show the conflicts due to unavailable registers (FP reg and Int reg), the conflicts due to busy functional units (FP unit and Int unit) and the conflicts due to full queues (FQ,IQ,LQ and ROB) when an instruction tries to allocate the corresponding resource during the workload execution.

As we can observe, RaT reduces the conflicts for all types of resources except for the integer functional unit (Int unit). The reason is that without RaT, and specially for memory-bound workloads, the functional units are underused because instructions are stalled waiting for a memory access most of the time. Using RaT, runahead threads turn these blocked threads in fast speculative threads which require more computational resources to proceed its execution through the integer functional units (remember that RaT does not use FP units). On the contrary, the queues in the pipeline, such the IQ and ROB, result in higher conflict reductions due to the unblock action of runahead

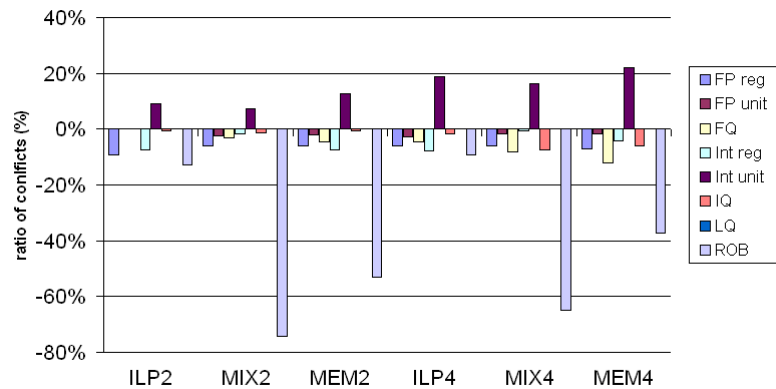


Figure 3.13: Resource conflict relation differences between baseline and RaT

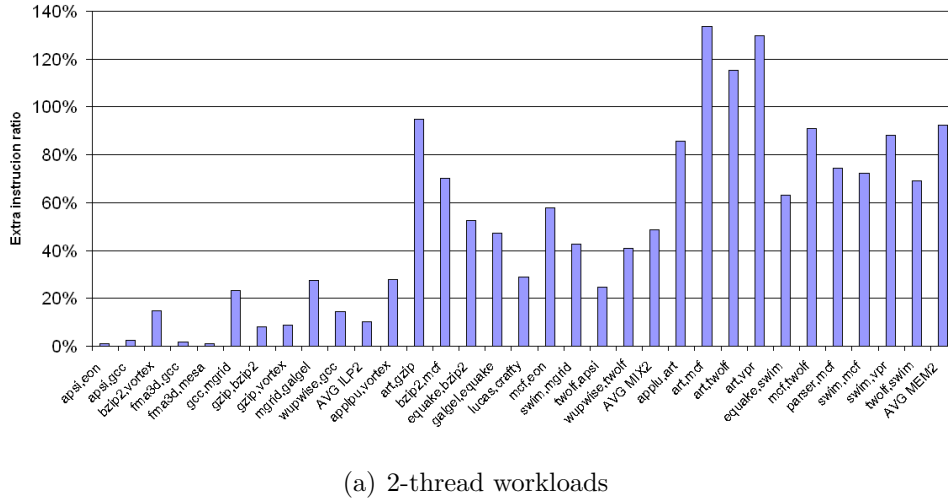
threads. For instance, RaT reduces 42% on average the number of conflicts in the ROB due to are occupied all its entries.

Instruction overhead

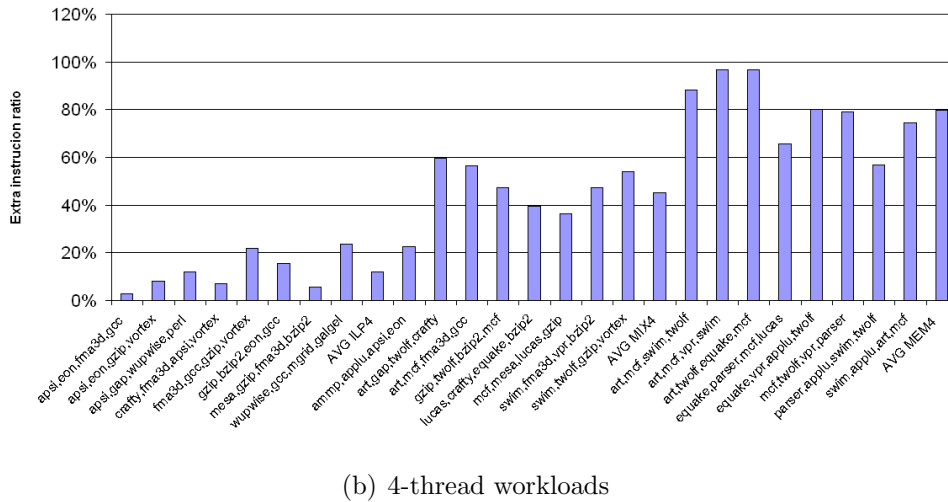
Runahead threads requires the speculative processing of extra instructions while an L2 cache miss is in progress. Since most of this speculative execution belongs to the future program stream, these instructions are later re-executed in normal mode. Therefore, Runahead Threads increases the number of instructions executed by the processor, which results in an instruction overhead compared to the baseline processor.

To expose this fact, we analyze the instruction overhead of Runahead Threads in SMT processors. Figure 3.14 shows the executed instruction ratio for the different workloads of RaT with regard to the baseline processor. As we can observe, the ratio of extra executed instructions due to runahead threads increases from ILP to MEM for both 2-thread and 4-thread workloads in Figure 3.14(a) and Figure 3.14(b) respectively. There are less runahead threads executed for ILP workloads than MEM workloads, therefore, much less extra instructions are generated. For ILP workloads, the percentages of extra instructions are 10.3% for 2 threads on average and 12.1% for 4 threads, with none of individual workloads over 30%. On contrary, these ratios are increased up to 92.3% and 79.7% for 2-thread and 4-thread MEM workloads respectively. The higher number of long-latency misses for these workloads generates more runahead threads, and thereby much more speculative instructions are executed

in MEM workloads (up to 133% in the case of art,mcf). While, these workloads achieve the most improvement in performance.



(a) 2-thread workloads



(b) 4-thread workloads

Figure 3.14: Ratio of extra executed instructions for 2- and 4-thread workloads due to Runahead Threads

However, this increase in executed instructions may not always result in a performance increase resulting in inefficient runahead threads for these cases. This drawback can be reduced by developing complementary mechanisms that address them. The following chapters of this dissertation describe new mechanisms that improve the energy efficiency of Runahead Threads.

3.4.3 Life time of runahead threads

To complete the RaT evaluation, we show an analysis about the life time of runahead threads. Firstly, we measure the average life time of the runahead thread executions in function of runahead thread activations per workload. Carrying out this calculation for the different simulations, the length in cycles of runahead threads is an average of 372 cycles. Although a different number of runahead threads can be activated according to the workload type, this average life time is more or less similar for overall runahead threads independently of the kind of workload. This is a reasonable average taking into account the minimum 300 cycles memory latency of our processor model (plus extra variable cycles depending on cache and bank contentions).

Throughout a total workload execution, there can be several actives runahead threads during certain periods (in function of the number of thread contexts). Figures 3.15 and 3.16 show the distribution of total execution time in function of the number of runahead threads in execution for each workload category. That is, each figure indicates the percentage of cycles regarding the total execution time in which there are a certain number of active runahead threads: none, one or two runahead threads for 2-thread workloads, and from zero (none) to four (all threads) for 4-thread workloads.

As it is expected, for ILP workloads, there is a high percentage of cycles in which there are no active runahead threads, that is, 76% for ILP2 and 59% for ILP4. This is because these workloads have few long-latency misses during their executions, thereby less runahead threads are activated. On the other hand, memory-intensive threads have the highest percentage of cycles in which at least one context is a runahead thread, being 89% for MEM2 and 97% for MEM4. In addition, for this type of workloads there are 42% for MEM2 and 11% for MEM4 of total cycles in which all contexts are executing a runahead thread. During this fraction of time, all threads are executing speculative runahead instructions and sharing the processor resources. For the rest of workloads, these percentages are not as high as for MEM ones, being 1.7% and 0.3% for ILP2 and ILP4 respectively and 14% and 1.2% for MIX2 and MIX4. For the other ranges of percentages depicted in these figures, different combinations of normal and runahead threads share the cycle time to advance in a cooperative way during their different executions.

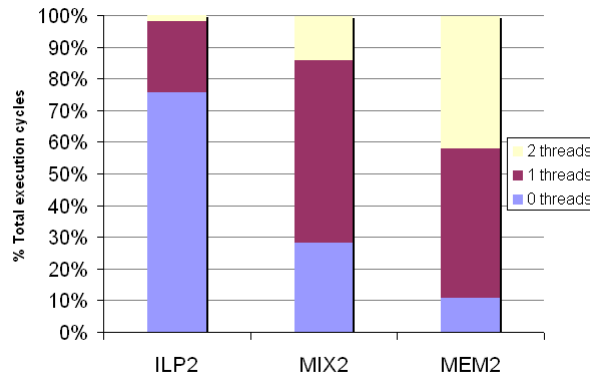


Figure 3.15: Total cycles distribution of runahead thread executions for 2-thread workloads

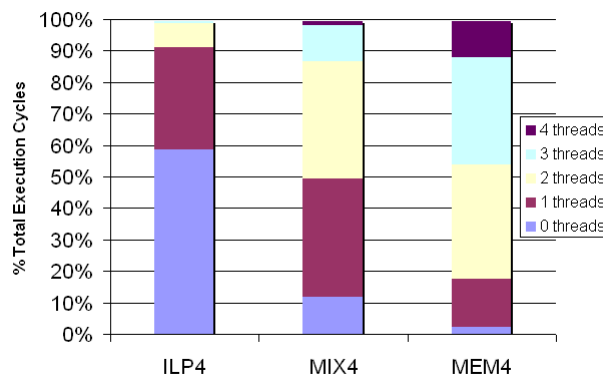


Figure 3.16: Total cycles distribution of runahead thread executions for 4-thread workloads

As particular example, Figure 3.17 shows in more detail the percentage of cycles that a benchmark becomes a runahead thread during its execution for MIX2 workloads. In this graph, for each workload composed by two benchmarks, we indicate the percentage of cycles that the corresponding thread is in runahead execution, T0 for one thread and T1 for the other thread. We can observe as depending on the percentage of runahead execution cycles we can identify which of the two threads that composed the mixed workload is the memory-intensive thread. For instance, the *mcf* benchmark reaches percentages near to 90%, or other memory intensive threads, such *art* and *equake*, over 60%. As we explain with the previous figures, there are portion of these percentages

(14% on average for MIX2) in which both hardware contexts are executing runahead threads.

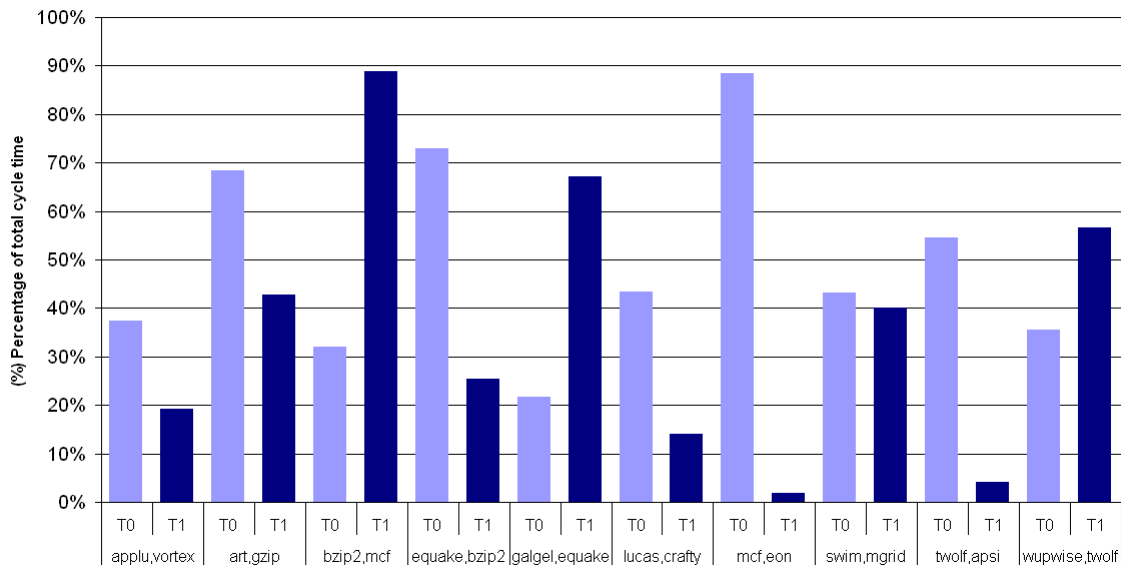


Figure 3.17: Distribution of runahead cycles per thread for MIX2 workloads

As examples to show the RaT effectiveness, we want to remark the (*equake,bzip2*) and (*lucas,crafty*) workloads. In these cases, the memory-intensive thread spends big periods of its execution time as a runahead thread (72% for *equake* and 44% for *lucas*). For this thread, RaT gets performance improvement by the prefetching effect (see Figure 3.7). Furthermore, RaT improves the performance of the other context, which executes the ILP thread (83% for *bzip2* and 76% for *crafty* respectively), due to avoid the resource monopolization of the memory-intensive thread.

3.5 Sensitivity of Runahead Threads to processor parameters

In the following sections, we study how RaT performs when different SMT processor parameters are changed in order to analyze the RaT sensitivity. We independently evaluate the RaT performance behavior in function of the number of executed threads, the main memory latency, cache sizes, reorder buffer features, and the register file size.

3.5.1 Hardware contexts

We evaluate what happens with RaT mechanism performance when the number of threads is increased. To explore the performance evolution of RaT with larger workloads, we extend the experiments with workloads of 6 and 8 threads, since up to now, the different evaluations were with multiprogramming workloads composed by 2 or 4 threads. We compose these additional workloads mixing the 2-thread and 4-thread workloads to obtain new mixed workloads of 6 and 8 threads with the same classification depending on the kind of benchmarks; ILP, MIX or MEM. Likewise, for the experiments with these larger workloads we scale the register file and the ROB size according to the rules of our modeled SMT processor described in Chapter 2. Hence, we increase the shared ROB size with 64 entries and the register file with 48 physical registers per each additional hardware context.

In figure 3.18, we show the average throughput for each kind of workload (ILP, MIX and MEM) for the set of 2,4,6 and 8-thread workloads for the processor with ICOUNT and RaT mechanisms respectively. Logically, the total throughput of the SMT processor increases while the number of threads is incremented as this figure shows. However, despite of the peak performance achievable (up to 8 instruction per cycle), the total throughput does not exceed the value of 4 instruction per cycle (IPC) on average for none of the results. The lack of more instruction level parallelism is one cause of this performance trend. In addition, the impact of resource contention is more critical as the number of threads executing simultaneously increases, reducing the ability to exploit more thread level parallelism among the threads. This can be observed in Figure 3.18 according the throughput speedup curve which is going lower as workloads are larger for both the baseline ICOUNT and RaT results. For instance, the throughput ratio of MEM4 versus MEM2 workloads is 68%, whereas the ratio for MEM8 versus MEM6 is only 6% for RaT mechanism. These results are in the line with ones that previous works have shown [33][34] as we explain in the chapter about our framework. SMT processors should not be interesting for the execution of more than 4 threads per core, since performance starts to saturate (or even degrade) for workloads with more than 4 threads.

Regarding the particular comparative between ICOUNT and RaT, the next table sums up the speedup average ratios for the different workloads of RaT performance over ICOUNT.

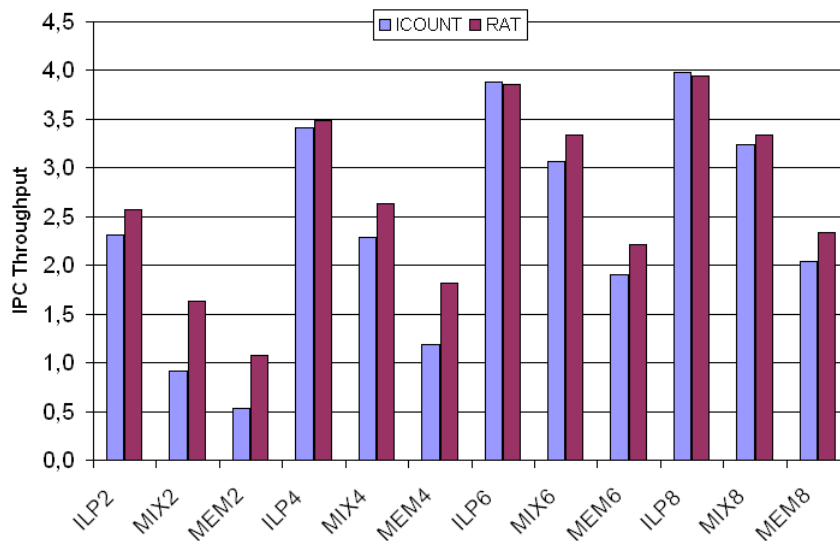


Figure 3.18: RaT performance evolution according to the number of threads per workload

#threads	ILP	MIX	MEM
2-threads	11.1%	78.6%	102.8%
4-threads	2.5%	15.1%	52.0%
6-threads	-0.6%	9.1%	15.8%
8-threads	-0.8%	3.0%	14.2%

Table 3.1: RaT performance speedup according to the number of threads

These data show as the performance difference is gradually decremented as more threads are executed in the processor. However, for almost all workloads RaT performs better than ICOUNT, and only in the cases of ILP6 and ILP8 workloads there is a negligible slowdown inferior to 1%. A higher number of threads requires a larger quantity of resources available to proceed with their executions. Using RaT, we are introducing more speculative instructions as we have more threads in the processor. In benchmarks with a good memory behaviour, such ILP workloads, the lack of prefetch opportunities do not compensate the extra speculative executions when there are more threads, since runahead threads are limiting the available resources for the other normal threads.

For the MIX and MEM workloads, there is also a performance difference reduction compared to ICOUNT for larger workloads. When there are more threads with high cache miss ratio, the memory related structures (such as load store units and cache ports) suffer from a higher pressure. However, RaT continues performing better for these workloads achieving good speedups as we can observe in the table 3.1. Indeed, RaT has 14.2% speedup over ICOUNT for the MEM8 workloads.

3.5.2 Memory latency

As processor and system designers continue to push for shorter cycle times and larger memory modules, and memory designers continue to push for higher bandwidth and density, main memory latencies will continue to increase in terms of processor cycles. When the main memory latency increases, the performance loss due to main memory latency becomes more significant. On the one hand, tolerating the increased latency would result in larger performance degradations per thread. On the other hand, the resource clogging problem would be worse because with a longer memory latency, memory intensive threads can retain more resources during more time causing resource starvation of the other threads in the processor.

Figures 3.19 and 3.20 show the average throughput performance of SMT processors of 2 context and 4 context respectively for different main memory latencies with ICOUNT and with RaT. For each group of workloads, each pair of bars shows the throughput of ICOUNT (left) and RaT (right) for the corresponding memory latency from 100 to 700 cycles. Overall, as memory latency increases, the performance of the SMT baseline processor with ICOUNT decreases, whereas the performance improvement due to RaT increases for all kind of workloads.

RaT improves the throughput by 16.3% on average on 2-thread SMT processor with a relatively short 100-cycle memory latency, while for a 4-thread processor, the throughput suffers from a slight reduction of 2.2% for that memory latency. RaT slightly degrades the throughput in some workloads on 4-context processor only for 100-cycle memory latencies (specially in ILP and MIX workloads). With a short memory latency, the runahead thread does not execute enough instructions to discover further L2 misses (prefetches) to compensate the speculative execution. Besides, the resource contention periods are smaller, not being as critical as for larger memory latencies. Since this short runahead execution does not provide enough benefits to outweigh the

cost of activating and deactivating the runahead thread (flushing all the speculative instructions each time), these factors result in performance loss in case of having 4 threads on the machine. Even so, RaT provides a performance improvement on average, albeit small of 2.1%, on MEM4 workloads for 100-cycle memory latency.

However, with a 700-cycle memory latency, RaT significantly improves the throughput by 137.1% on average on 2-thread workloads and by 65.8% on 4-thread workloads respectively. With a longer memory latency, the time a thread spends in runahead mode increases. Hence, the runahead thread can execute more instructions further ahead in the instruction stream. This results in the discovery of L2 misses further in the instruction stream, which could not have been discovered with a shorter memory latency. On the other hand, the likelihood of clogging resources are much higher with 700 cycles of memory latency. This factor would create more resource monopolization situations. Nevertheless, RaT allocates and deallocates the different resources dynamically by the runahead threads. Thus, RaT also alleviates these situations because it avoids the large allocation time of resources during these larger memory accesses.

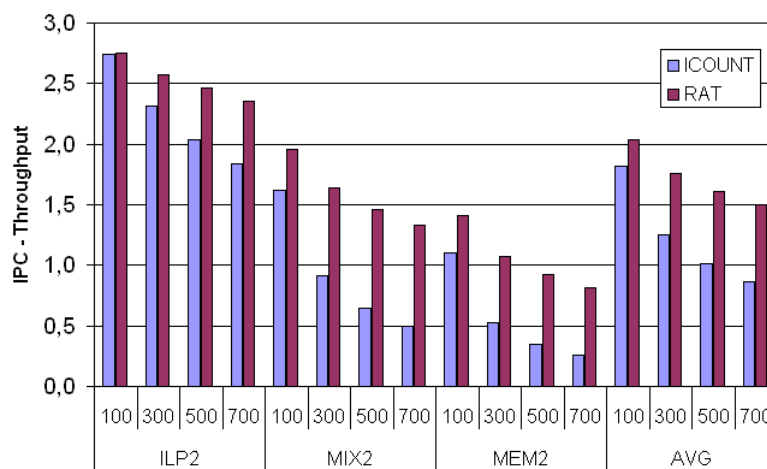


Figure 3.19: RaT performance with different main memory latencies for 2-thread workloads

It is worthy to note that the performance of SMT processor with 700-cycle memory latency using RaT is 20% higher than the performance of the baseline SMT processor (without RaT) with 300-cycle memory latency. So, RaT can preserve, and even improve, the absolute performance of an SMT processor with a short memory latency as the memory latency gets longer. Therefore, RaT is a good mechanism for increasing the

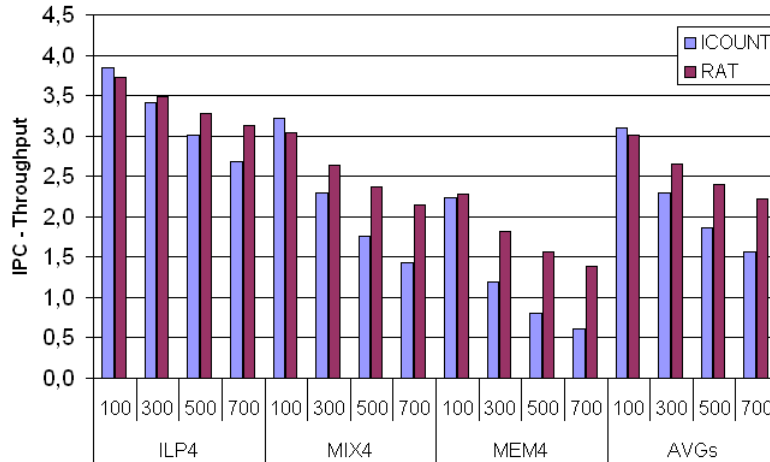


Figure 3.20: RaT performance with different main memory latencies for 4-thread workloads

tolerance of long main memory latencies on SMT processors. As future processors are expected to have significantly longer main memory latencies than current processors, we expect that RaT will become more effective in future processors.

3.5.3 L2 cache size

The size of the caches is another parameter to take into account to evaluate RaT sensitivity. For example, the capacity of the second level (L2) cache affects the number of accesses that result in misses, and therefore, the runahead threads executed and overall processor performance. Figures 3.21 and 3.22 show the performance throughput of baseline ICOUNT and RaT when the L2 cache size is varied. We perform evaluation for cache sizes of 512KB, 1MB, 2MB and 4MB.

Increasing the size of L2 cache benefits the baseline SMT processor as we can observe the ICOUNT results. From 512KB to 4MB, the performance of ICOUNT increases 36.4% for 2-threads and 28.3% for 4-threads. Likewise, the ratio of improvement of RaT compared with the baseline is still significant for each group of workload (except ILP4) while the L2 cache size is increased. Thus, RaT outperforms ICOUNT by 38% and by 28.5% on average for 2MB and 4MB of L2 cache size respectively. Therefore, the benefits of RaT by prefetching and resource contention reduction are kept even for larger size of L2 caches.

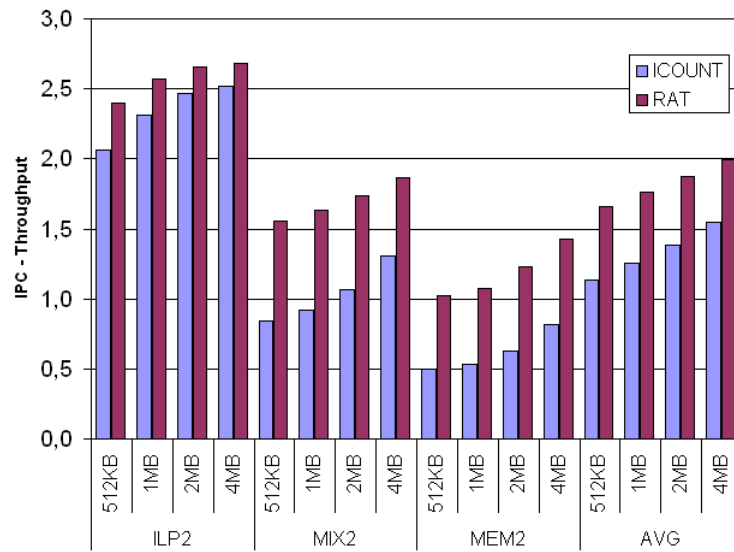


Figure 3.21: Performance throughput with different L2 cache sizes for 2-context processor

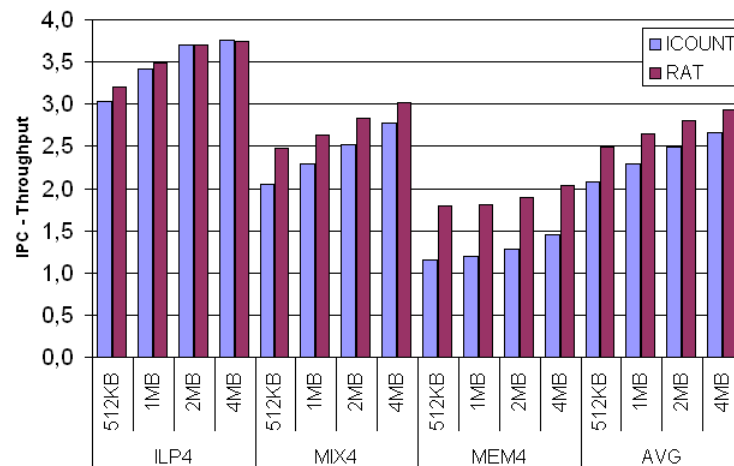


Figure 3.22: Performance throughput with different L2 cache sizes for 4-context processor

3.5.4 Reorder buffer size

The reorder buffer is a critical structure on the processors that not only ensures the commitment order of instructions but it also establishes the number of instructions that can be executed in-flight (that is also known as the instruction window). In this section, we want to compare the instruction processing model of an SMT processor with

Runahead Threads to SMT processors with different reorder buffer sizes, enlarging the instruction windows.

In order to make this performance comparative, Figures 3.23 and 3.24 show the throughput of four different SMT processors with different reorder buffer sizes for each set of multiprogramming workloads. From left to right in each workload group, the performance is shown for baseline ICOUNT (the light bar) and RaT (the dark bar) with 128, 256, 512, and 1024-entry reorder buffer, respectively. The sizes of other processor structures (register files and issue queues) are scaled proportionally to the increase in the reorder buffer size to support the larger number of in-flight instructions.

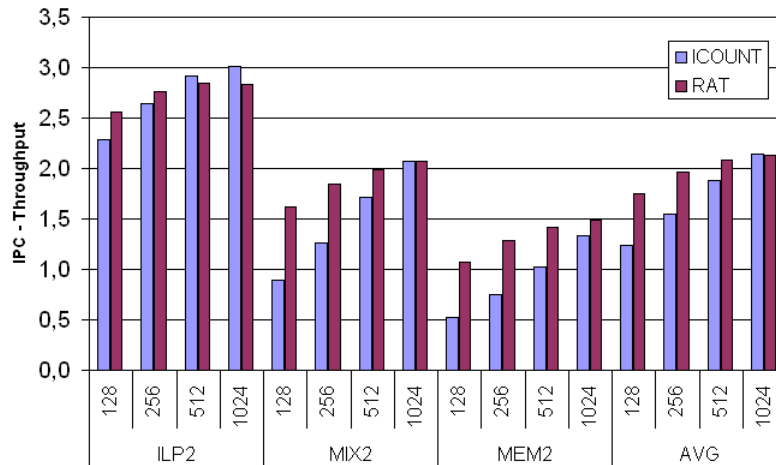


Figure 3.23: Performance throughput with different ROB sizes for 2-context processor

In these figures we can observe as while the reorder buffer size is increased, the processor performance differences between using or not RaT is decreased. Overall, up to 512-entry ROB, RaT achieves better performance throughput, but from this size onwards, the RaT performance improvement starts to cut over the baseline with larger ROB sizes. For small sizes, RaT provide better memory latency tolerance by providing the ability to look farther ahead in the program to discover later L2 misses, since it emulates the possibility of enlarging the instruction window to execute the runahead instructions. When the processor increases the instruction window with larger ROBs, it also acquires this ability to tolerate long memory latencies. However, whereas RaT has to flush and re-execute a larger quantity of speculative instructions due to a larger ROB, a processor with a larger ROB without RaT does not. Therefore, in spite of

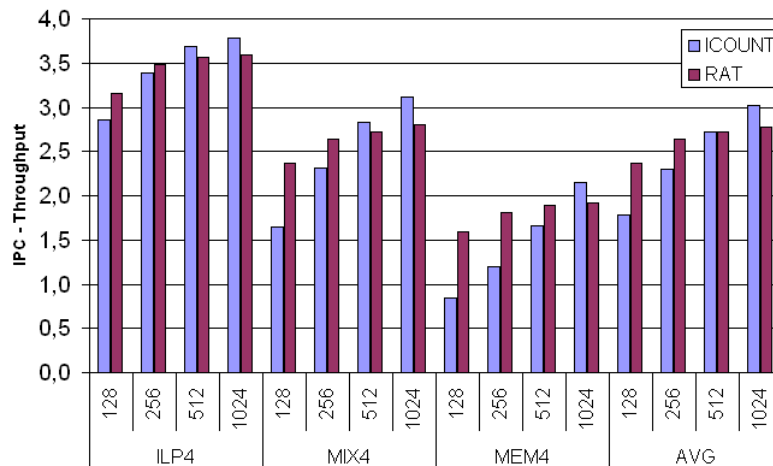


Figure 3.24: Performance throughput with different ROB sizes for 4-context processor

the benefits from prefetching are similar, the performance benefit provided by RaT is limited by the flush and resume process from all speculative instructions in case of larger ROB.

However, based on our results, RaT with a 128-entry ROB achieves similar performance (except for ILP workloads) to a 512-entry ROB SMT processor without RaT. That is, RaT approximates or surpasses the performance with a quarter of the ROB size in this case. In addition, in terms of hardware cost and complexity, an SMT processor implementing a larger window has significantly higher hardware cost than RaT with a small window. RaT does not add large structures in the processor core, nor does it increase the size of the structures that are on the critical path of execution. In contrast, a larger window requires the size of processor critical structures to be increased proportionally to the instruction window size. So, a larger instruction window requires large and complex buffers that are needed to implement and keep the high instruction stream of a large window processor. Therefore, we think implementing RaT on a small instruction window without significantly increasing hardware cost and complexity becomes a more attractive alternative to building large instruction windows, specially in multithreaded processors.

3.5.5 Shared vs. non-shared ROB

As explained in the Chapter 2, our SMT simulated processor implements a shared reorder buffer in order to analyze any possible critical resource contention with this important structure. Here, we analyze the impact that have this feature in the Runahead Threads mechanism. We evaluate the implications and performance differences of having a shared ROB versus a non-shared one.

To this evaluation, Figure 3.25 represents the performance throughput of ICOUNT and RaT mechanisms when they use a non-shared and shared ROB respectively. We setup the experiments so that the total number of reorder buffer entries are the same for both configurations (shared and non-shared). Thus, we configure an SMT processor with 64 reorder buffer entries per context for the non-shared ROB version and an unique ROB of 128 total shared entries in the case of 2 thread contexts and 256 shared entries for the case of 4 thread contexts.

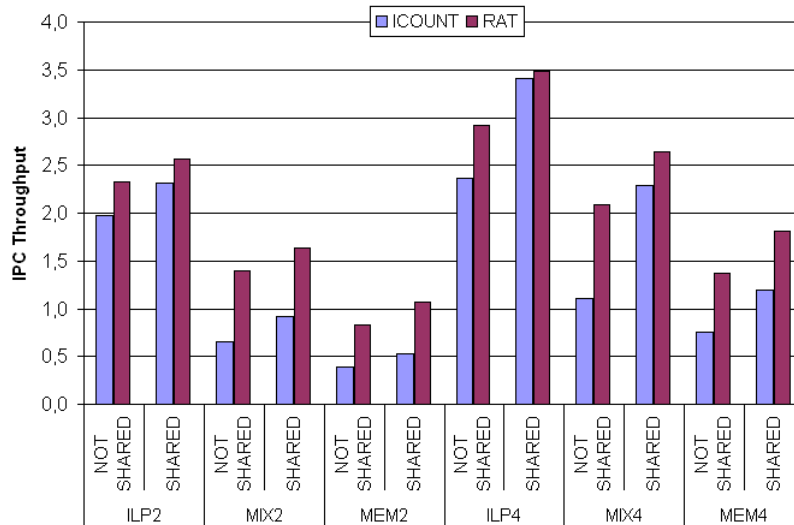


Figure 3.25: Shared vs. Non-Shared ROB performance

Both ICOUNT and RaT performance results are better for the shared ROB than having a partitioned ROB per context. The average performance speedup of ICOUNT with the shared ROB versus non-shared is 50%, whereas for RaT is 22%. In addition, the ICOUNT improvement is much better for 4-thread workloads (70%) than for 2 threads (31%). In the case of partitioned ROB, the threads with limited ILP do not use all the ROB entries, so these entries cannot be used by the other threads to exploit

the TLP. With a shared ROB, each free entry can be used by whatever thread of the mix of workload. Therefore, a thread with more pressure over the ROB can take profit of free slots not occupied by other threads to exploit the corresponding ILP achievable.

Regarding RaT versus ICOUNT performance in function of this ROB feature, RaT achieves a higher speedup over ICOUNT for the non-shared version by 74%. For the shared ROB, that is our baseline configuration by default, the overall speedup of RaT versus ICOUNT is 45%. The reason of this fact is because there is a major room for improvement in the case of a partitioned ROB due to previous explanation.

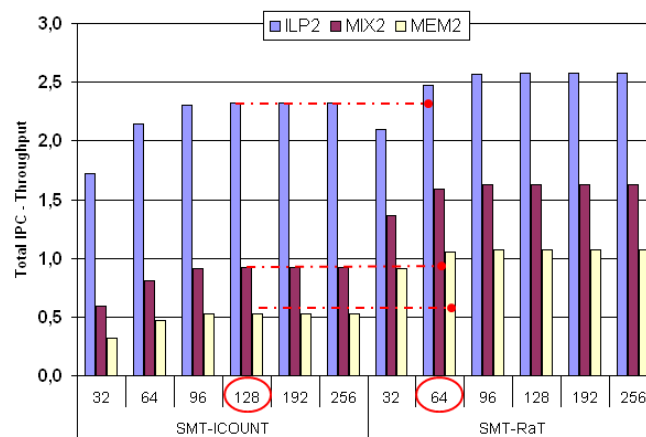
We also evaluate the partitioned ROB with different sizes as in the previous section. The results obtained follow the same tendency for both ICOUNT and RaT as in the shared ROB case. The average difference of speedups between RaT and ICOUNT for shared and non-shared rob when the size is varied is below 4%.

3.5.6 Register file size

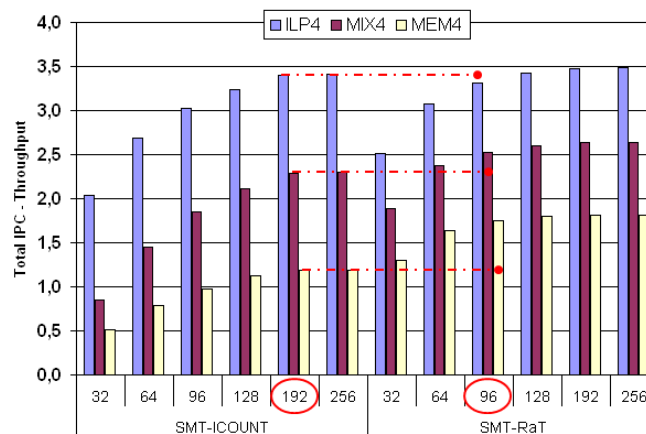
The register file (RF) is an important shared resource inside an SMT processor, and its size is one of the key issues in the SMT design. RF needs to be sized very generously to support the full architectural state for each thread as well as to provide a sufficient number of additional registers for renaming. Therefore, such critical resource, related to the ISA, sets the number of threads that an SMT processor is able to simultaneously execute in its core. For instance, in our 4-context SMT processor model, 128 physical registers (that is $32 * 4$) are needed for keeping the precise architectural state of all threads. Then, we enlarge the register file with 48 additional renaming registers per each hardware context. Therefore, the total physical registers of the processors are 320 (i.e. $48 * 4 = 192 + 128$) but only 192 are available for sharing between the threads for renaming.

Larger register file requires more complex design with higher access time that affects the final performance and power consumption. In this sense, we demonstrate through the study of this section that, with the benefits of RaT, an SMT processor using RaT with fewer registers can perform better than an other SMT processor with larger register file without RaT. To show this fact, we compare the performance of our SMT baseline to the SMT with RaT for different register file sizes. Figures 3.26(a) and 3.26(b) show the performance throughput for the 2-thread and 4-thread workloads respectively as a function of the available registers for renaming (from 32 to 256 reg-

isters). As we can observe, the throughput decreases when the number of registers is reduced, especially in the case of 4-thread workloads. However, this reduction is less pronounced when the RaT mechanism is used. For instance, the MIX2 and MIX4 workloads suffer from 36% and 63% slowdown on the baseline machine with ICOUNT when the register file is reduced from 256 registers to 32. When RaT is applied, this slowdown is only 16% and 28% respectively. Therefore, an SMT processor with RaT is less sensitive to register file size.



(a) 2-thread Workloads



(b) 4-thread Workloads

Figure 3.26: Performance throughput for different number of renaming registers

On the other hand, if we compare the throughput of SMT-ICOUNT for all experiments in Figure 3.26(a) to only RaT results with 64 registers, the latter overcomes

the former for almost all configurations except for the ILP4 workloads with more than 128 registers. In fact, for 2 threads, the performance of RaT using 64 physical registers is better than the performance of baseline SMT with ICOUNT using 128 physical registers, that is, using the twice of the register file size¹. This performance gain is 6.8%, 72.8% and 98.4% better with RaT for ILP, MIX and MEM 2-thread workloads respectively.

In the case of 4 threads, as ICOUNT saturates with 192 registers, the equivalent comparative is for 96 registers with RaT. The performance difference of RaT with 96 registers versus the baseline with 192 registers is -2.4%, 10.5%, and 46.5% for the same 4-thread category of workloads respectively. These results demonstrate that RaT using the half of register file performs 59.3% for 2 contexts and 18.2% for 4 contexts better than the SMT without RaT. Therefore, this is a very important advantage of RaT, since it allows using smaller register files on SMT processors even improving the performance.

To explain these results, Figure 3.27 shows the average amount of allocated physical registers (both integer and fp) per cycle for each kind of workload. There are two bars per workload in this figure. The left bar shows the average number of allocated physical registers per cycle in normal threads. The right bar shows the average number of allocated physical registers per cycle in runahead threads. This data shows that the part of programs executed speculatively with runahead threads use less registers than in normal execution. Among the different workloads, the MEM ones keep allocated a higher amount of registers more cycles in normal threads, while runahead threads reduce this data as figure shows. In particular, memory-intensive workloads with RaT use 27% less registers than they would use without this mechanism for MEM2 (20% in case of MEM4). On average for all workloads, this physical register requirement is reduced to 16%. In addition, as runahead threads behave as speculative threads with fast instruction execution, RaT reduces the average time the registers are allocated making them available sooner to other instructions or threads.

A recent work [58] in the SMT context proposes a mechanism to release physical registers early belonging to those instructions that are independent of an L2 cache miss. The basis of this mechanism consists of traversing the ROB several times to identify which registers can be early deallocated (those that are not dependent on the L2-miss

¹the performance of the baseline saturates with this register file size, since it gets the same for larger sizes.



Figure 3.27: Average physical registers used per cycle between normal and runahead threads

load). However, even if the idea is simple, the hardware overhead of the proposed mechanism is not when compared to our proposal.

3.6 Design analysis for Runahead Threads implementation

This section evaluate the design tradeoffs related to the different alternatives about the general RaT mechanism explained previously. We will explain through several evaluations and results why certain elements are chosen or not in our final Runahead Threads mechanism implementation. We present several conclusions drawn from the experimentation with the different RaT design issues.

3.6.1 Fetch policy setup

First of all, we evaluate the more suitable configuration of the underlying ICOUNT fetch policy for the runahead thread execution. The ICOUNT policy gives fetching priority (and thus resource accesses) to threads with fewer instructions in the pre-execute pipeline stages (from fetch to dispatch). We analyze three configurations of ICOUNT parameters to operate as fetch policy in cooperation with our RaT mechanism. We evaluate the performance throughput results of each combination when we change the number of threads and the total instructions that can be fetched in a single cycle

according to the ICOUNT policy parameters. The different configurations evaluated are:

- RaT with ICOUNT 1.8: one thread per cycle is allowed to fetch until eight instructions.
- RaT with ICOUNT 2.4: threads fetch four instructions from each of two threads for a maximum of eight instructions.
- RaT with ICOUNT 2.8: two threads can fetch up to eight instructions per cycle.

Figures 3.28 and 3.29 show the throughput results for these three configurations for all 2-thread and 4-thread workloads respectively. As these figures show, RAT 2.8 configuration performs better than the other two configurations evaluated (this also follows similar results from other work that studied the ICOUNT policy alone [75]). With the 2.8 fetch configuration, there are a higher mixed stream of instructions from several threads due to a high fetching capacity. Hence, ICOUNT 2.8 favors the fetch opportunities of memory-intensive threads once they become runahead threads in the mixed workloads.

Regarding the two figures, the performance gains are bigger in 4-thread workloads than for 2-thread workloads. On average, RAT 2.8 performs 5% better than RAT 1.8 and 1.5% than RAT 2.4 in the case of four threads, whereas for 2-thread workloads these percentages are 1% and 3.5% respectively. When we have few contexts in the processor (e.g. 2 threads), the maximum fetch bandwidth per thread is more important. This is shown in our results in which RAT X.8 configurations (with up to eight instructions) are better than 2.4. Using RaT, as most of them are basically fast threads (both normal and runahead ones) during their executions, it is important to can fetch instructions from several threads. In the case of having more execution contexts (4 threads), they require to interlace the fetching from different threads to advance in an uniform way. Thus, a thread is still able to add instructions in the same cycle even being not of the highest priority, when the other thread cannot fill the fetch bandwidth. In particular, RAT 2.X setups perform better than 1.8 for 4-thread workloads as we can observe in Figure 3.29. They allow to fetch from two threads per cycle and, therefore, exploiting better the thread-level parallelism even in our new scenario with normal and speculative threads. Due to these performance results, we select the 2.8 configuration as the best for the underlying ICOUNT setup in combination with RaT.

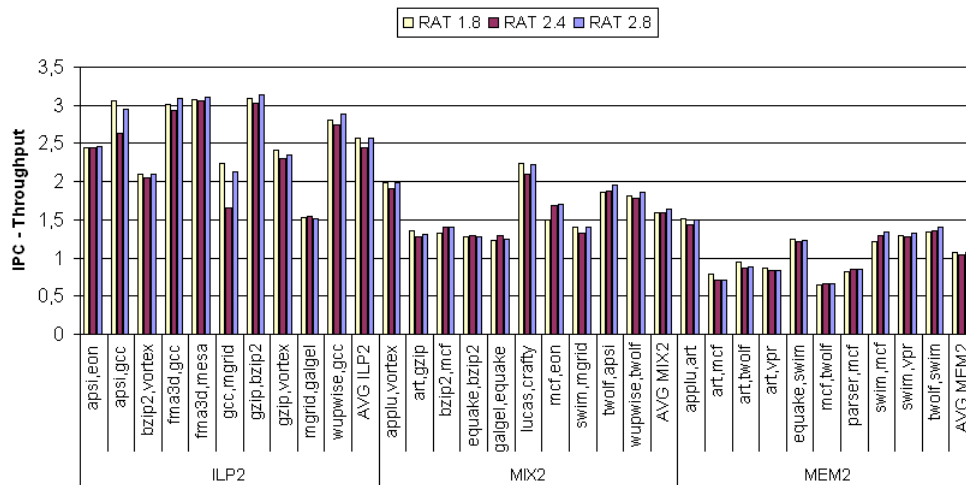


Figure 3.28: RaT throughput according to underline Icount setup for 2-thread workloads

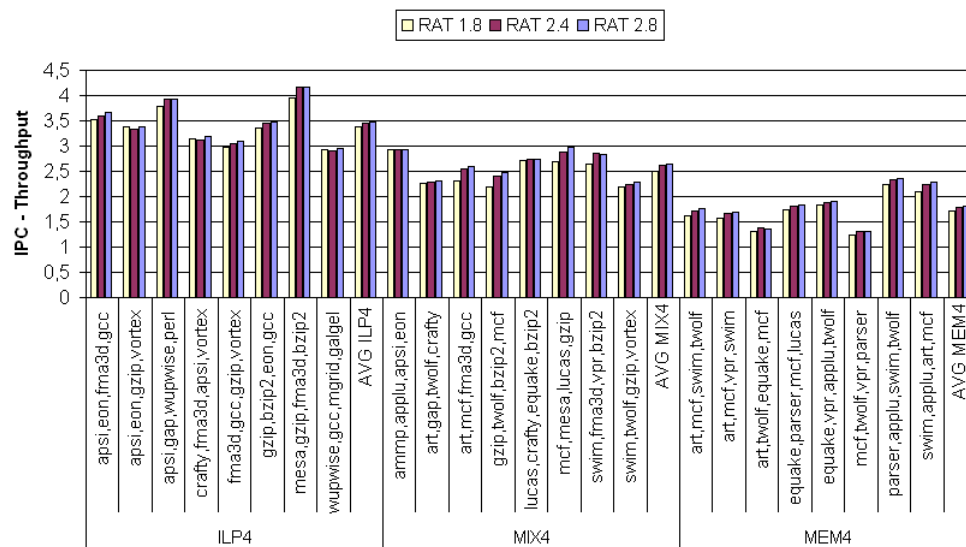


Figure 3.29: RaT throughput according to underline Icount setup for 4-thread workloads

3.6.2 Different thread priority schemes for Runahead Threads

In the previous study, we evaluate ICOUNT configuration assuming that all kind of threads have the same opportunities in order to be considered to fetch instructions according to ICOUNT policy. Nevertheless, with the introduction of RaT mechanism

in the SMT architecture, there are two types of threads that try to use the processor resources: normal threads (non-speculative) and runahead threads (speculative). The question is whether different rules of thread priorities can be used instead of the standard ones of ICOUNT policy as better scheme to manage this new situation. In order to analyze this issue, we investigate several modification of ICOUNT scheme varying the thread priorities depending on the kind of thread. We recall that ICOUNT till now only takes into account the amount of instructions in the pre-execute stages to calculate the thread priority, independently of the kind of thread (speculative or not).

We evaluate three different priority proposals derived from ICOUNT. One of the proposals consists of giving high priority to normal threads against runahead ones at the fetch stage. Thus, the normal threads have the opportunity to get into the pipeline in the first place. Subsequently, the runahead threads take profit of the possible remaining fetch slots after scheduling the normal threads. However, this new policy follows the same priority rules that standard ICOUNT imposes among the normal and runahead threads respectively. That is, it decides the thread priority in function of instructions in the pre-execute stages for the threads belong to the same category (normal or runahead ones). The second fetch priority scheme we analyze here consists of inverting the rights with regard to the previous one, that is, giving priority to runahead threads opposite to the normal ones. The third approach is similar to the first one, but also adding a new level of priority with the same criteria at the issue stage: first instructions from normal threads can be issued and then runahead ones. In this case, normal threads have not only the maximum priority at the fetch stage but also at the moment of taking the functional units in the issue stage.

Figure 3.30 shows the average performance obtained by RaT in function of the different thread fetch priority policies. The four bars of each kind of workload represent the average throughput for RAT with ICOUNT base policy and the different thread priority schemes respectively described above: fetch priority to normal thread (FPNT), fetch priority to runahead threads (FPRT) and, both fetch and issue priority to normal threads (BPNT).

This figure shows that giving priority to runahead threads (FPRT) is not a good choice since the performance considerably decreases. This approach favors speculative runahead threads while the non-speculative threads, which do the “real” work, are being delayed. The consequence is that FPRT has a slowdown compared to RaT+ICOUNT of 9% for 2-thread workloads and 19% for 4-thread workloads. The

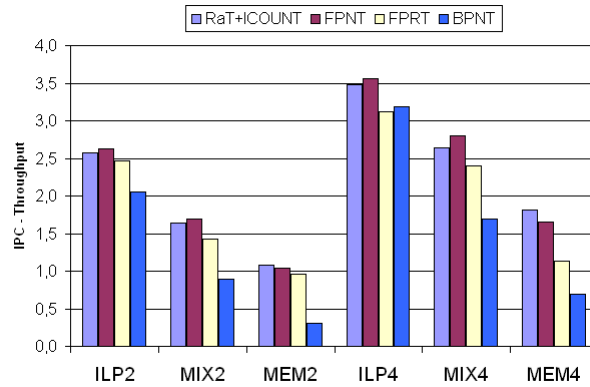


Figure 3.30: Performance of RaT in function of the different thread priority schemes

performance loss is higher for 4-thread workloads since having more threads, it is more likely to have more runahead threads, thereby more interference with the normal threads.

However, the worst performance priority scheme evaluated is BPNT, with an average performance loss of 40%. This slowdown is specially significant for memory-intensive workloads (66%) following by mixed workloads (41%). We analyze the reason of these bad results. The cause of this poor performance is the side effect of the different priority between normal and runahead threads at the issue stage. Since normal instructions have the highest priority in the issue stage, these instructions always are issued to execute before the runahead instructions each cycle. Then, most of the times the issue bandwidth is filled with normal instructions and runahead instructions have no room to advance. The negative effect of this is that BPNT scheme does not let the processor execute enough runahead instructions, causing a lot of very short runahead threads (20 executed instructions on average per runahead thread). Our results show that BPNT produces four times more runahead thread activations on average than RaT with original ICOUNT rules. In consequence, these short runahead threads are totally useless since they cannot advance and almost do not issue any prefetching, causing at the same time a big performance degradation due to the activation and deactivation of even much more runahead threads due to every long-latency load. Therefore, this is not obviously a valid priority scheme.

On the other hand, giving higher priority to normal threads only at the fetch stage (FPNT) seems a good option, especially for ILP and MIX workloads. This scheme favors the threads that have long periods of high-level instruction parallelism. The

instruction streams flow quickly in the multithreaded pipeline, having fewer instructions in the front-end stages. However, this effect harms memory-bound threads (according to MEM workload results) due to it impedes the speculative runahead instructions to advance quickly in order to anticipate the prefetching. Then, while for ILP and MIX workloads FPNT gets a performance improvement of 2% and 4.5% respectively, for MEM workloads it suffers of a 6% slowdown.

Therefore, in relation to the throughput results obtained in this study, we consider that original ICOUNT is the best choice to combine with RaT as the underlying thread priority policy. ICOUNT is able of distributing effectively the threads independently of they were speculative or not. Besides, it is the simplest option, not requiring additional logic or hardware complexity to manage the priorities according to the thread type.

3.6.3 Checkpointing delay

Typically on a multithreaded processor, a thread switch takes a much longer time than 1 clock cycle. However, RaT mechanism does not require a context switch (as other speculative multithreaded techniques), but only an execution mode change. This thread mode change and the runahead execution is supported through hardware checkpointing as we have explained above.

This section shows to what extent the checkpoint mechanism delay can affect the RaT performance. The checkpoint is a procedure very important in RaT mechanism since is the base to be able to perform the runahead speculative execution. Although a particular checkpoint implementation depends on the underlying micro-architecture, we evaluate specifically the RaT performance in function of the number of cycles to perform that checkpoint according to possible implementation models. These cycle delays affect both the time to start the corresponding runahead thread (generating the checkpoint) and the period to resume to a normal thread once the runahead thread is finished (restoring the checkpoint).

Figure 3.31 shows the evolution of RaT performance when the checkpoint delay is varied from zero to ten cycles. As the number of delay cycles increase, the performance throughput is slightly degraded. From 0 cycle to 1 cycle checkpoint delay there is not noticeable performance loss in all workload categories. With higher delays, there is an increasing slowdown, specially for ILP workloads which suffer 4.7% of performance degradation in ILP2 and 2.1% for ILP4 with 10 cycles. Mainly, the re-start time afer

a runahead thread affects to this kind of workloads which provide a high IPC. For the rest of workloads, the loss of performance is staggered but it is not steep while the delay is increasing. For instance, MEM4 workloads only lose 0.3%, 2.1% and 3.8% in throughput when the checkpoint delay is 3, 5 and 10 cycles respectively.

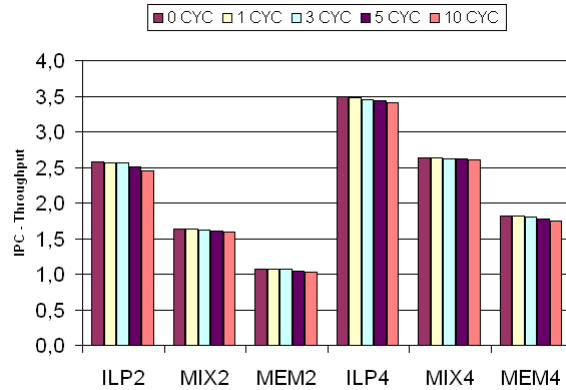


Figure 3.31: Evaluating performance of RaT with different checkpointing delays

Therefore, the checkpoint delay for RaT is not critical for the final performance as these results show due to the ability of SMT processors to overlapped the different thread executions. While a checkpoint are being generated or restored, other threads take benefits of processor resources to proceed without affecting the total performance throughput. Likewise, we estimate that one cycle delay is enough to perform our checkpoint model, which as we just observe does not cause any performance degradation compared to 0 cycle delay results.

3.6.4 Use of the runahead cache

As we comment in Section 3.3.2, a previous implementation of runahead execution for out-of-order processors [51] integrates a Runahead cache. This structure is used to communicate and propagate the invalidation status between dependent loads and stores that have already been committed.

In this section, we evaluate the performance throughput of our Runahead Threads mechanism with and without the use of the runahead cache. Figures 3.32 and 3.33 show the throughput for 2-thread workloads and 4-thread workloads respectively corresponding to this evaluation.

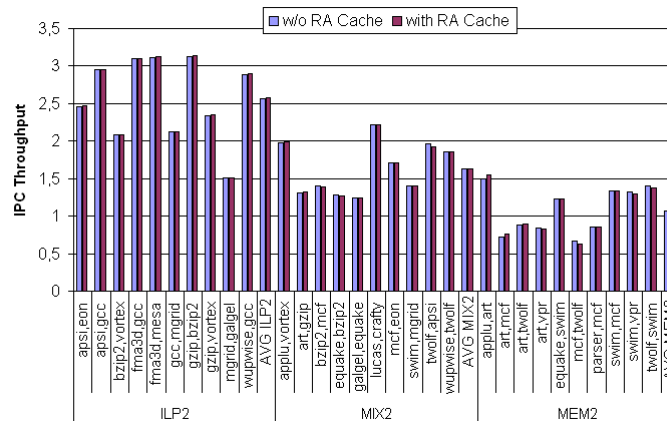


Figure 3.32: Runahead Threads performance without and with runahead cache for 2-thread workloads

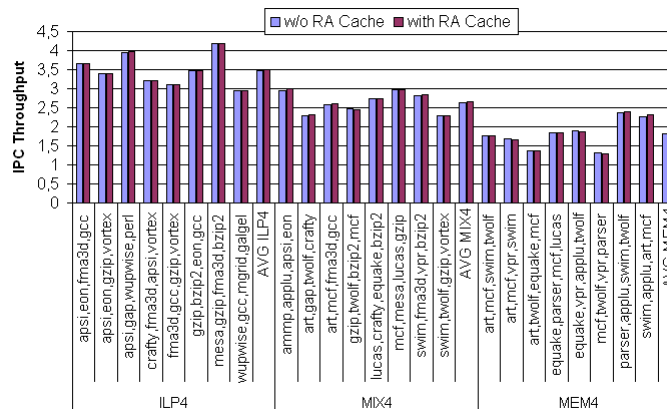


Figure 3.33: Runahead Threads performance without and with runahead cache for 4-thread workloads

As we can see in these figures, using the runahead cache does not have any significant impact on performance in the runahead thread executions for our SMT model. Overall, the different multiprogramming workloads have the same performance. Only, in some memory-intensive workloads we can observe a little performance gain using the RA cache, but, there are also other workloads with higher performance improvement without using the runahead cache. These small variations depend on the degree of dependencies between load and stores that generate more or less accurate prefetches. In addition, these results follow the trend obtained in the single-thread runahead execution work [51], in which the performance deviation without RA cache in SPEC2000

benchmarks is also very small for out-of-order processor in that study. Due to these results and the reasons explained in section 3.3.2, we decide not integrate the runahead cache in our implementation.

3.7 Summary

Memory-intensive threads can clog up shared resources due to long-latency memory operations without making progress on SMT processors, thereby hindering overall system performance. In this chapter, we have presented *Runahead Threads* as alternative mechanism to alleviate these problems related to the SMT scenarios.

In contrast to existing fetch policies and resource control schemes that usually restrict memory-bound threads in order to get higher throughput, RaT implies a new point of view in the context of resource management through a speculative execution mechanism. RaT turns any running thread into a *runahead thread* when the SMT processor detects that thread undergoes a long-latency load. While being a runahead thread, this thread behaves as a fast speculative thread by runahead execution until the load is resolved. This runahead thread uses the different shared resources without having a negative impact of their availability for the other threads.

This simple functionality of RaT mechanism has several important advantages on the SMT processors:

- First, RaT alleviates the SMT problem of handling the long-latency loads, specially in the case of memory-intensive threads. RaT allows memory-bound threads to advance speculatively, instead of stalling the thread, doing beneficial work without disturbing the other threads. In this sense, RaT balances resource usage between computation-intensive and memory-intensive threads.
- Second, RaT significantly improves the single-thread performance by prefetching which allows exploiting the memory-level parallelism available while a long-latency load is serviced. This provides benefits on a single threaded application, which is not provided by multithreading, therefore also improving the overall processor performance.
- Third, RaT provides not only a high-performance but also an efficient way of using shared resources in SMT processors in the presence of long-latency memory

operations. On the one hand, it avoids the possible resource monopolization of memory-bound threads, transforming them into light speculative threads and allowing the other threads to continue executing with the remaining resources. On the other hand, RaT also prevents threads from falling in resource under-use situation, since the execution of runahead threads take profit of the free resources to perform the speculative execution.

- Fourth, RaT increases the register file efficiency and provide higher performance for the same number of registers. It is also worthy to note that an SMT processor that implements RaT can benefit from smaller register file with even performance improvements.

To contrast RaT advantages, a detailed evaluation of the mechanism is provided. Our detailed evaluation has shown that RaT performs better than the SMT processor baseline in terms of throughput (44%) and Hmean (38%). RaT outperforms ICOUNT on average for all categories of workloads, especially in the case of MIX and MEM workloads. These evaluation results show the significant performance benefits of using RaT, whereas higher throughput ensures higher utilization of processor resources to improve the performance, good fairness results through Hmean metric ensure that all threads are given similar opportunities and that no threads are forced to starve.

Code Semantic-Aware Runahead Threads

In the previous chapter, we introduce Runahead Threads (RaT) as a promising solution to alleviate the memory-intensive thread problem in SMT processors. RaT employs runahead execution to enable a thread to speculatively execute instructions and prefetch data instead of stalling because of a long-latency load. However, as runahead threads speculatively executes large portions of the instruction stream, an SMT processor with RaT executes more instructions than a not speculative SMT processor. Therefore, RaT improves overall processor performance by prefetching and alleviating the resource contention among threads but RaT has a shortcoming: these benefits come at the cost of executing a large number of instructions speculatively due to runahead executions. If a runahead thread execution does not provide prefetching benefits, this can degrade energy efficiency by executing a large number of useless instructions without performance gain.

This chapter addresses this drawback of runahead threads in which we propose several solutions to enhance the effectiveness of RaT. The objective is to decrease the number of useless instructions executed with the runahead threads, while still preserving the performance improvement provided by RaT. In this chapter we present a research line for improving runahead thread efficiency by simple and complementary code semantic control techniques. These proposals perform coarse-grain analysis to capture the prefetch opportunities (usefulness) of executed code structures, such loop and subroutines, during the runahead thread executions. We propose to dynamically use code semantic information for detecting these particular program structures and

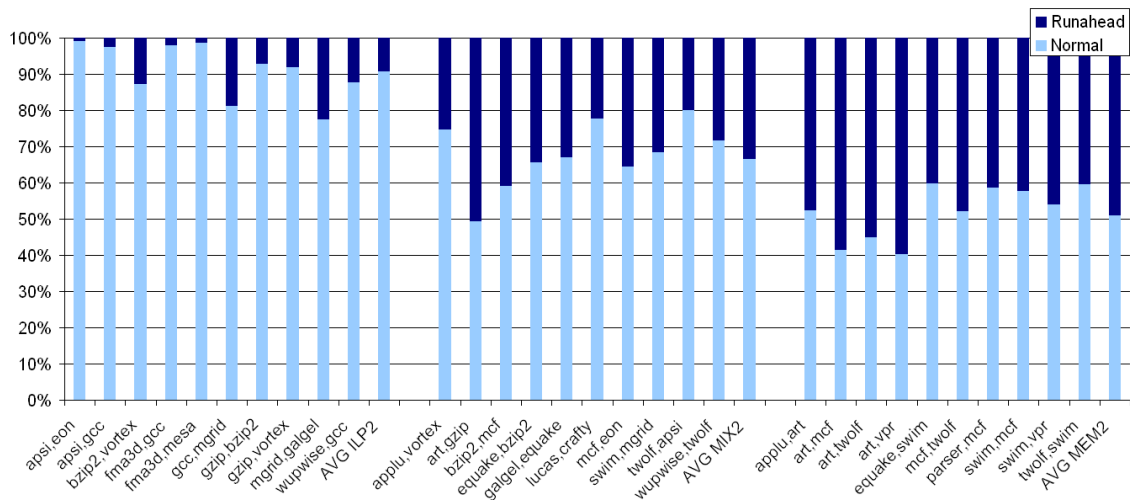
to analyze when they are useful or not in order to control the runahead thread execution. In function of this runtime dynamic analysis, the proposed techniques make a control decision either to avoid or to stall the particular loop or subroutine execution in runahead threads. By means of these control actions, our goal is to make runahead threads more efficient and reduce the dynamic energy consumption of SMT processor that use RaT.

4.1 Efficiency and Runahead Threads

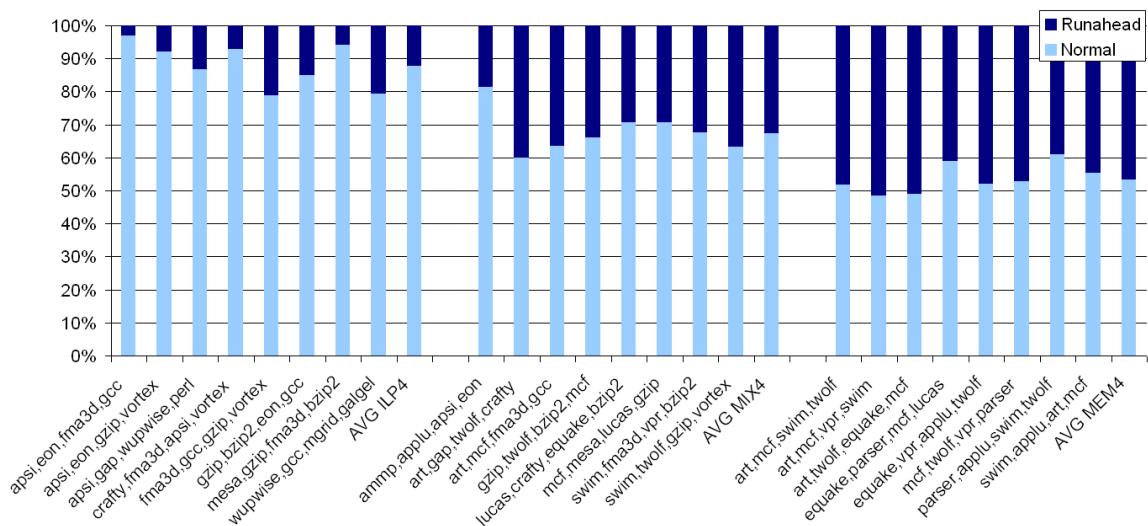
By using RaT, SMT processors provide a performance and complexity-effective framework to improve memory latency tolerance and reduce resource clogging on long-latency loads. Nevertheless, RaT requires the speculative processing of extra instructions while an L2 cache miss is in progress for that thread. When every runahead thread ends its speculative execution, the processor restarts the normal thread flushing the hardware context pipeline and beginning with the instruction that caused turn into runahead thread. Hence, an SMT processor with RaT mechanism can execute the same instructions in the instruction stream several times. Therefore, RaT increases the number of instructions executed by a conventional SMT processor.

To expose this fact, Figure 4.1 shows the distribution of speculative runahead instructions over the total number of instructions executed for the different workloads. Each bar is composed of the ratio of normal instructions and the runahead ones with regard to the total executed instructions. As we can observe, the portion of additional executed speculative instructions due to runahead threads increases from ILP to MEM for both 2-thread and 4-thread workloads. There are much less cache misses executing ILP workloads than MEM workloads, therefore, much less runahead threads are generated. For ILP workloads, the percentages of runahead instructions with regard the total number of instructions are 8.9% for 2 threads and 11.7% for 4 threads. On contrary, these ratios are increased up to 47.9% and 46.3% for 2-thread and 4-thread MEM workloads respectively. That is, in case of memory-intensive workloads, almost the half of total instructions executed are speculative runahead instructions.

Therefore, this figures exposes the execution of additional amount of speculative instructions by RaT, which results in an increase in the dynamic energy dissipated by the processor. Nevertheless, the inefficiency problem arises due to RaT has no information in advance about the existence of useful future prefetches for a particular



(a) 2-thread workloads



(b) 4-thread workloads

Figure 4.1: Distribution of executed instructions for 2- and 4-thread workloads

runahead thread. As consequence, it can perform useless extra work if there is no available prefetching. Thereby, when there is a runahead thread executing without doing prefetching, this thread does not contribute to improve the performance meanwhile it executes useless speculative instructions. This useless extra work impacts on the overall power consumption, thereby energy per instruction wasted. The main drawback for these cases is that RaT benefits can be spoiled if the thread executes a large number of useless speculative instructions without doing prefetching.

To be worth, the executed runahead threads should maintain an efficient relation between the performance gain and the extra speculative work performed. That is, the clue about the efficiency of a runahead thread can be seen as the relation between the performance improvement provided due to prefetching and the number of additional executed speculative instructions due to runahead execution. For instance, RaT increases the number of executed instructions by 29.7% to achieve a 64% speedup on average for 2-thread workloads whereas for 4-thread workloads, RaT increases the throughput by 24% at a cost of increasing the number of executed instructions by 30% on average. Therefore, even considering acceptable this extra work from the performance point of view, we can make RaT a more efficient mechanism if we control the useless portions of runahead threads.

4.2 Overseeing program structures to improve RaT efficiency

To improve runahead thread efficiency, we propose to control the useless extra work through coarse-grain analysis based on code semantics. The main idea is to analyze the execution of large section of code (coarse-grain) to reduce the number of speculative instructions executed. To this goal, we investigate several mechanisms that address the usefulness of particular program structures detected by code semantic analysis during runahead thread execution. As the speculative execution of instructions in runahead threads targets the discovery of useful prefetches from cache misses, instruction processing during runahead mode must be optimized for maximizing the number of useful misses generated during runahead execution. At the same time, it is needed to minimize the number of speculative instructions that do not generate such prefetching. For this purpose, we propose detecting common code semantic structures with a high instruction granularity to oversee and control the runahead thread efficiency in this sense.

We devise these mechanisms with two main features in mind. First, the code semantic patterns should be easy to detect in order to design low-complexity mechanisms. The new structures or components for the implementation should not complicate more the hardware to keep RaT a feasible mechanism. Second, these code structures should have a large number of instructions involved, that is, the highest possible granularity

of speculative instructions to capture a big ratio of extra work. Therefore, we choose two semantic patterns very common in the program codes which fulfill the requested features:

- *the loops*, which basically are repetitive sequence of instructions in the programs.
- *the subroutines*, typically functions or procedures which are called frequently inside a program.

Our goal is to analyze and oversee when a runahead thread is executing a useful loop or subroutine. In case that such code pattern execution is not useful, the processor takes a decision from efficiency point of view. The key point of our proposals is to obtain dynamic information about these structures, loops and subroutines, to decide whether they are useful or not during runahead thread execution. We assume such code pattern as *useful*, when the runahead thread issue at least one useful load to the memory system during its execution. Likewise, we define *useful load* as a valid runahead load which misses at the first level cache, and therefore can prefetch data from upper memory levels.

Based on this information, we present several techniques that decide (1) to skip the execution of useless parts of the program detected in loops or subroutines toward a better useful part or (2) to stall the runahead thread execution to avoid unnecessary speculative execution and allows shared resources to be used by non-speculative threads.

In the next sections, we describe deeper the functionality and implementation for each of the approaches proposed for controlling the loops and the subroutines in runahead threads.

4.3 Loop control techniques

The first approach is based on controlling loop executions during runahead threads. The idea is to detect when a runahead thread is executing a loop and then to analyze the loop execution to find out that the runahead thread is really doing useful work inside that loop. We consider a loop as *useful*, when the current runahead thread can issue at least one useful load to the memory system during each loop iteration. Finally, depending on the dynamic usefulness of the speculative loop execution, our mechanism

does a control action: either it forces the exit of the loop and continue the execution after that, or it directly stalls the runahead execution.

The loop usefulness control techniques we proposed consist of three parts. First, we need to identify a loop during the runahead execution. Second, once the loop is detected, we need to determine if this loop is useful for the runahead thread. Third, we need to take a control decision based on the loop usefulness. The first two steps are common for each of techniques, since is the procedure to detect the usefulness of the loops. The last step is different depending on the action to take, which fixes a different purpose for each particular case.

Next, we explain the design trade-offs and implementation details for each of the three phases of our runahead thread loop control techniques.

Step 1. Loop detection

Previous studies [20][30][59][72] have proposed mechanisms to dynamically detect the loops that are executed in a program. In this line, a current research presents the *Loop Processor Architecture* (LPA) [30], focuses on capturing the semantic information of high-level loop structures and using it for optimizing program execution. The LPA design uses a buffer, namely the loop window, to dynamically detect and store the instructions belonging to simple dynamic loops along with all the information needed to build the rename mapping. Therefore, the loop window can directly feed the execution back-end queues with instructions, avoiding the need for using the front-end stages of the normal processor pipeline.

Following similar lines to detect loops, we just take into account simple loops forms made by the compiler in order to simplify the design of our proposal. The dynamic execution of a simple loop implies the repetitive execution (loop iteration) of the same group of instructions (loop body) which are enclosed between a loop branch and its target [21] [30]. The most common forms of loops, such as `for`, `while`, and `repeat`, are compiled by the Compaq C and FORTRAN Alpha compilers with the conditional branch check at the bottom of the loop with a negative displacement. Then, a loop is normally identified from a backward branch which is taken (once per iteration). The backward branch is considered the loop end (last instruction) and its target address is considered the first instruction of the loop body.

Therefore, the procedure to detect a loop is simple: when a backward branch is predicted taken during a runahead thread execution, our mechanism starts the loop

detection process. To do this, we store the branch program counter and its instruction target in two registers, `LOOP END` and `LOOP START` respectively. We need this information to control when the runahead thread is executing inside the range of the loop body. If the same backward branch is found and it is taken again, then we identify it as a loop branch. We indicate this state to the runahead thread using a new flag (`INLOOP`), which is set to one.

In Figure 4.2 we illustrate the loop detection overview. This figure shows an example of a loop structure with a set of instructions. The loop body starts at instruction `@20` and finalizes with the loop branch `@44`. When that sequence of instructions are executed and detected by our mechanism, instruction `@20` are stored in the `LOOP START` and `@44` in the `LOOP END` registers respectively, as it is showed in the figure. The second time this pattern of instructions is repeated consecutively, the `INLOOP` flag is set to identify that loop and let the corresponding loop control technique know this fact. Once the `INLOOP` flag is set, the `LOOP END` and `LOOP START` registers do not change until a decision about the usefulness of the loop is taken or the runahead execution leaves that loop.

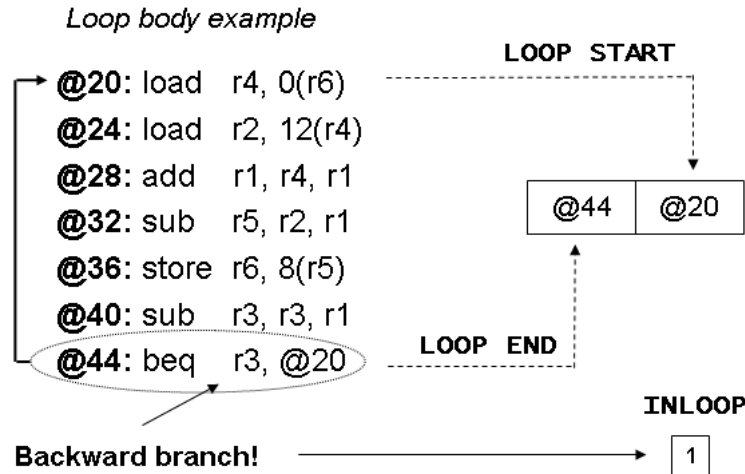


Figure 4.2: Loop detection example

Based on this simple algorithm, the technique can detect the most common kind of loops with a low-complexity mechanism. In case of nested loops, the procedure presented focuses on handling and detecting the inner loops. If the number of iterations is enough to detect the loop, the instructions belongs to the innermost loop should be the last ones in being stored in the `LOOP END` and `LOOP START` registers.

Step 2. Loop usefulness check

Once the loop detection procedure confirms the existence of a loop, the mechanism is ready to oversee the loop usefulness. We use a simple *useful load flag* to capture the useful work made during each loop iteration. This flag is set to zero at the beginning of the next iteration of the loop (which is detected when the backward branch is taken). From this point onwards, this flag is set to one when there is a runahead load that misses in the first level cache and the flag is always reset when each next iteration starts. In addition, each time we find the loop branch again, the useful load flag is checked. If the flag value is zero, this means there were not useful loads in the previous iteration and, then, the loop is candidate to be considered as useless. Otherwise, the iteration has executed at least one useful load. The advantage of our mechanism is that this usefulness test is made during the own execution of the loop. That is, we use current dynamic information while the runahead thread is inside the detected loop execution. Therefore, we use accurate and actual information and we do not need large structures to store that information for all possible executed loops.

Once the backward branch is taken by the third time, the loop control mechanism is able to make a decision over the control action to do during the runahead thread execution. In this point, our technique contains information about the usefulness of the previous iteration of the loop. This depends on whether the useful load flag is set (useful) or not (useless). Although the control action could be done directly using only this flag, we introduce an additional level of confidence by means of a saturated counter (See study in Section 4.6.1 for performance details).

To accomplish this, we modify the original design to include a saturated counter of two bits. We name it as *usefulness saturated counter*. We keep the same philosophy (check the loop usefulness of the loops) but now, the mechanism also increments this saturated counter for each iteration with the useful load flag equal to one and it is decrement in the other case. According to our studies, this helps the mechanism to tune the decision being a bit conservative with regard to the different loop body structures. For instance, some loops contain different execution paths that produce useful and useless iterations alternatively. Using the saturated counter we can prevent the mechanism from taking an incorrect decision in an intermittent useful loop.

Following with our loop example, Figure 4.11 shows an iteration in which the load instruction @24 has caused a cache miss. In this point, the miss event is taken into

account and recorded by the useful load flag. Hence, at the end of this iteration, the usefulness saturated counter is incremented by one because the useful load flag is set.

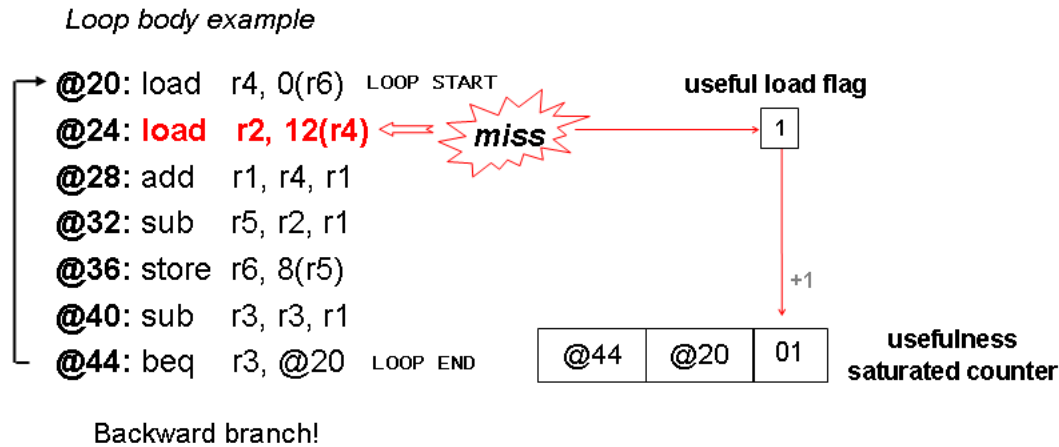


Figure 4.3: Loop usefulness mechanism example

Step 3. Loop control decision

Finally, thanks to the previous dynamic analysis we can find out which loops are useful during a runahead execution. On the contrary, if we detect a loop execution as useless, we propose to make a control decision. With the described implementation, a loop is considered useless when the usefulness saturated counter has reached the value zero. Then, this mechanism is able to do the control action the next time the backward branch is taken and the saturated counter is zero.

In this third step, we propose two possible actions associated to two loop control techniques to improve the RaT efficiency. That is, each action represents a different technique in this sense in which one focuses mainly on performance and the other on energy consumption.

The first one is *Loop Reverse -LR* which forces the exit of the useless loop reversing the backward branch taken prediction. This allows runahead thread continue the speculative execution beyond this loop trying to capture further prefetching. By means of this action, the runahead thread avoids the execution of the useless loop iterations and looks ahead for a useful part of instruction stream in the program. The goal of this

technique is to improve the efficiency of runahead thread maximizing the performance gain by more future prefetches.

The second one is *Loop Stall -LS*, which simply stalls the runahead execution in order to avoid to continue executing useless speculative instructions in the loops. In this case, the goal is to directly cut down the extra work in order to reduce the energy consumption. This stall action entails the termination of the runahead thread, following the procedure to exit from runahead mode and stall the thread until the L2 miss causing runahead is finally serviced. In addition, this action can help the processor to alleviate the shared resource pressure when the runahead thread is stopped because there would be one thread that does not compete for them.

Hardware requirements

The different presented loop control techniques are simple and cost-effective. This feasibility is twofold. On the one hand, the new introduced components require small additional hardware. On the other hand, the required modifications present a simple control logic that can be easily integrated in the processor.

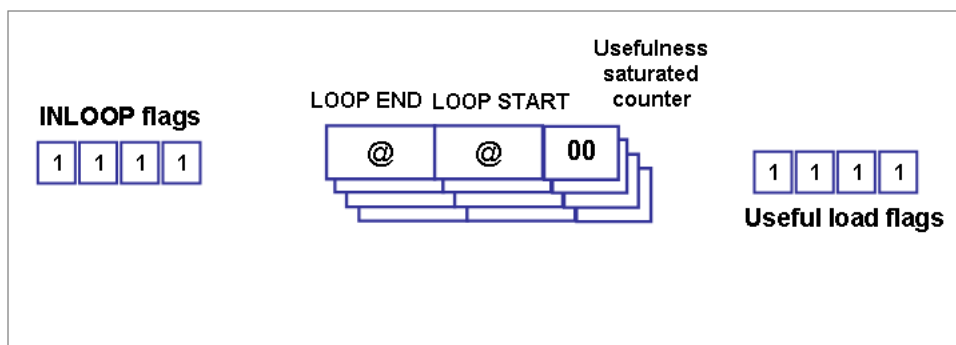


Figure 4.4: Hardware requirements of loop control techniques. One component of each structure is associated to each hardware context.

As Figure 4.4 shows, these loop control techniques introduce the (INLOOP) flag and the LOOP END and LOOP START registers per hardware context to detect and control when a runahead thread is in a loop. Additionally, each pair of these registers have the associated useful saturated counter and usefulness flags per thread to oversee the prefetch possibilities of each iteration.

4.4 Subroutine control techniques

The second approach on the line of code semantic control to improve the RaT efficiency focuses on the subroutines. The idea is the same that previous mechanism for loops, but applying the *usefulness* concept to subroutines. That is, we consider a subroutine as useful for runahead if during its execution the runahead thread issues some useful load.

We refer *subroutine* to all the so-called procedure, method or function at the high-level languages which performs a specific task in a program. A subroutine is a piece of code so that it can be executed (called) several times and/or from several places during a single execution of the program, and branch back (return) to the point after the call once its task is done. This feature makes subroutine a good choice of code structure in order to control a high granularity of speculative instructions per runahead thread.

Like the control loop mechanism, the subroutine control mechanism consists of three steps: the subroutine detection, the usefulness analysis and, finally the control decision in function of the technique applied.

Step 1. Subroutine detection

A language's compiler translates subroutine calls and returns into machine instructions. The instruction sequences corresponding to call and return statements are called the subroutine's prologue and epilogue. These prologue and epilogue are described by a sequence of ordinary instructions whose rules and methods are defined according to a calling convention. Each particular compiler (according to the programming language) has its default calling convention. Depending on the underlying architecture (ISA) and Operative System (OS), the compiler adds the required assembly (machine) prolog- and epilog-code automatically to reflect the particular settings. This determines among the most relevant issues, where to put arguments, in what order, where the called function will find the return address, and how to communicate certain information between the caller and the called subroutine during invocation and return.

However, runahead execution is independent and transparent to the calling convention. Runahead threads execute speculative instructions without modifying the memory or the stack state. This speculative execution relies on the not invalid instruction execution assuming that required state for executing the subroutines is correct.

Therefore, for the detection process of subroutines, we only need to know that subroutines are clearly delimited in the instruction stream due to the language's compiler translation of subroutine calls and returns into specific machine instructions. Hence, the subroutine detection can easily be performed at runtime since the process simply consists on capturing the subroutine call at the decode stage once the corresponding instruction is identified. For example, the Alpha ISA has the instructions `jsr`, `bsr` and `jsrcouroutine` to call a subroutine. In the same way, the end of the subroutine execution is delimited by the corresponding return assembler instruction. Most ISAs provide these kinds of instructions (e.g the instruction `ret` for x86 and Alpha machines), so our techniques can be easily ported to any processor.

The action to perform in the processor to record a subroutine detection is straightforward. Each time a subroutine call is detected, the program counter of the subroutine instruction is stored in a new register, called `INSUBROUTINE`), which, at the same time, indicates the runahead thread is in a subroutine. Notice that we can only detect subroutines the compiler has not inlined, but due to their characteristics (short body) they are not interested in this study.

Step 2. Subroutine usefulness check

Although the concept of usefulness for subroutines is the same as loops, in this case the usefulness check mechanism should do a deeper analysis due to the different levels of callings. If inside a subroutine body another subroutine is called, we consider that the usefulness information of the called subroutine also belongs to the caller subroutine. Hence, it is needed to track the usefulness through the nested subroutine calls or subroutines inside the others. In this sense, we need to control the deep level of subroutines between parent and child calls. Additionally, we need to transfer the usefulness information from the child subroutines to the parents to avoid eliminating useful subroutines in the lower level of calls.

For this purpose, we can easily implement this procedure using some of the logic already presented in current architectures: the return address stack (RAS) [35][81]. The RAS consists on a storage buffer for reducing the number of wrong predictions due to the subroutine call/return paradigm. By using a set of stacks, this mechanism identifies when a `call` is made and then supplies the correct branch target when the `return` is encountered. Basically, the RAS keeps track of pairs of call and return instructions. Each time a return instruction is executed, it matches with one particular call instruc-

tion. Thus, a call instruction pushes a return target on a RAS, and the corresponding return instruction pops its predicted target off the stack. The RAS is typically implemented as a circular buffer and it is managed through a top-of-stack pointer (TOS) pointing to the current top of the stack [62]. In a multi-thread environment, there is generally one RAS per hardware context. At any moment, the stack contains only the pairs of the calls that are currently active (namely, which have been called but have not returned yet) per each thread. Therefore, the processor already controls the subroutine levels by means of the RAS. During runahead execution, the RAS must be updated for speculatively fetched instructions, but remembering that RaT makes checkpoints of the RAS per each runahead thread. Then, its previous state is not affected by the speculative execution when a runahead thread finished.

Therefore, we can use the RAS employed for return address prediction for our mechanism. We extend each entry of the RAS to store an additional bit for the useful load flag associated to each subroutine (represented by the tag PC of subroutine call and return address pair). This modification represents an insignificant hardware cost. Now, when a subroutine call is detected and an entry associated with it is inserted in the RAS, we also reset its corresponding useful flag to start overseeing its prefetching usefulness. During each particular subroutine execution, the useful load flag is updated (set to 1) when at least one L1-miss load is found. The corresponding useful flag of the current subroutine is always at the top of the RAS, which is pointed by the TOS. Likewise, when the return instruction is executed, the processor pops the corresponding entry from the RAS. In this point, we also get the associated useful load flag value from the ended subroutine, which indicates this subroutine usefulness.

In addition, as a subroutine can be called from other subroutines, the usefulness information of a child subroutine should be transferred to the parent subroutine. Then, if a subroutine was called from another subroutine (the parent subroutine), its flag value is added (using an OR operation) with the useful load flag of the parent subroutine when it is pop from the RAS. This operation is simple since the parent subroutine was the previous last entry of the RAS and it is set at the top of the RAS once the child subroutine is returned. Figure 4.5 shows a snapshot about this procedure for the subroutine control mechanism, in which A, B1 and A2 represent active subroutines that have been pushed in our new extended RAS. As we show in this figure, when subroutine A2 is removed from RAS, its associated useful flag value (one in this case) is transferred to the parent subroutine B1. At the same time, we update the

INSUBROUTINE register with the current subroutine PC (B1). In the case there are not a parent subroutine in the RAS, the runahead thread exits from the top subroutine and then, the INSUBROUTINE register is reset to zero. This procedure is carried out during all program execution, both in normal and runahead mode to keep the consistency of subroutine levels.

- **Subroutine call example**

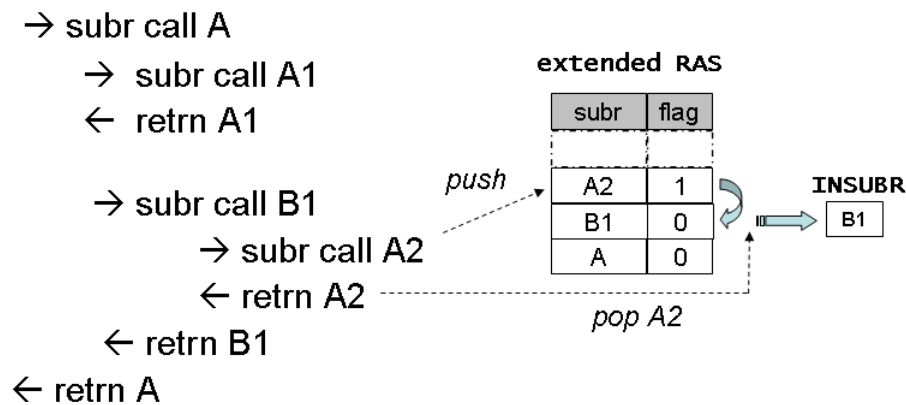


Figure 4.5: Subroutine control mechanism

Step 3. Subroutine control decision

Whereas for loop control techniques we make the decision dynamically during the own loop iterations, in the case of subroutines we need to make it based on its past history. That is, we collect information about the current subroutine execution to decide what to do when this same subroutine is called again. Therefore, it is needed to introduce a small table, we name (**SUBR TABLE**), to store the subroutine program counter (PC) and usefulness information about its previous executions. Basically, each entry of this table consists of a tag (part of subroutine PC) and a saturated counter of two bits with the same functionality as for loops. That is, to control the usefulness hysteresis among different dynamic executions of the same subroutine.

The **SUBR TABLE** is updated when the execution exits of a subroutine (return instruction), taking the PC from the RAS entry and updating the saturated counter depending on the useful flag value from its corresponding entry. If this flag is set, the counter is incremented and if the flag is zero, the counter is decremented. Following

with our previous example of subroutine calls, in Figure 4.6 we show this procedure when the subroutine A2 return. The PC of subroutine A2 is used to index the SUBR TABLE and update the corresponding entry. This process is made in parallel while the processor pops the subroutine entry off the stack. Thus, it takes the PC of subroutine A2 to access the SUBR TABLE and increments the saturated counter by one due to its useful load flag was set.

- **Subroutine call example**

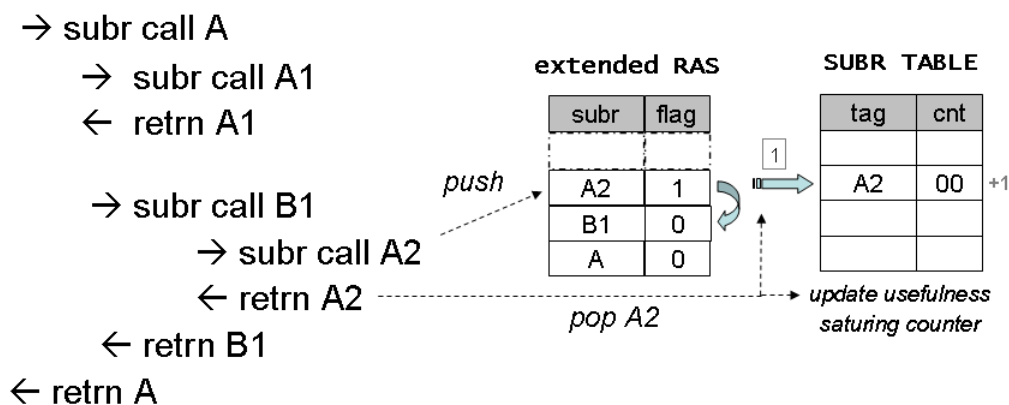


Figure 4.6: Subroutine control mechanism

The SUBR TABLE is looked up when a subroutine call instruction is decoded during runahead execution using the instruction PC in order to get the usefulness information from its saturated counter. Based on this information, the different subroutine control techniques make a decision. For this subroutine control mechanism, we propose three different actions when the usefulness saturated counter of a particular subroutine indicates a possible useless execution (value equals zero).

The first technique *Subroutine Skip -SK*, skips the subroutine execution. This technique avoids the subroutine call instruction execution, skips the subroutine body and jumps to the instruction pointed by the return address to continue the runahead execution. In most cases, the return address is the address of the instruction following the one that transferred control to the called procedure. Thus, we simply assume the continuation point is always the next instruction in program order (PC+4) of the subroutine call instruction. Nevertheless, this return address also can be found in different places depending on the processor instruction set architecture. For an Alpha

standard call, the return address is passed and returned in the return address register R26. In addition, it is possible to obtain it from the RAS as well, which is more common in most of current architectures.

This SK technique has some issues regarding the calling convention of the program executed. On the one hand, for caller clean-up standards (which it is the more common one), the prolog and epilog instructions are speculatively executed in spite of skipping the detected useless subroutine. Their execution does not cause any inconvenience excepts executing some extra useless instructions in case of avoiding the corresponding subroutine. On the other hand, for callee clean-up case, only the prolog instructions are executed since the callee subroutine (that executes the epilog in this standard) is skipped. As we explain before, the prolog instructions are oriented to pass information among subroutines, so this action would cause these instructions sometimes write bogus values for some specific registers to a subroutine that will not be executed. We want to remark here that the memory and by default also the stack, are not changed during runahead execution. As runahead threads do not modify the memory, the stack of the thread is not modified neither before nor after the subroutine execution. In addition, exceptions generated by runahead instructions are not handled. In case that a runahead thread causes an exception, for example due to an execution inconsistency or an illegal memory accesses, RaT marks as INV the destination register to avoid using bogus values due to exception-causing instructions.

The second technique to control subroutines is *Subroutine Stall -SS*, which stops the runahead thread without doing the subroutine execution. This technique directly stalls the execution before doing the subroutine jump once the SUBR TABLE have been accessed and detects its useful saturated counter is zero. Like *Loop Stall* technique, SS finishes the runahead execution for this thread and waits the L2 miss is resolved to resume to normal execution.

The third one is a combination of the previous two (SK and SS), in which we introduce an additional level of information to choose one particular action or another. This technique takes profit of the L1-miss tracking process to update the useful load flag and to extend its functionality to find out if there is available nearby prefetching after a subroutine execution. Depending on this usefulness information, this control subroutine technique decides whether skip the subroutine or stall the execution. We called this last technique *Post-Subroutine Usefulness -PSU*.

To implement this last technique, we simply add one bit more per entry into the `SUBR TABLE` to indicate if there are L1 cache misses after the subroutine execution (*PSU flag*). Besides, we need to remember the PC of the previous subroutine after returning from it to be able to record its post-subroutine usefulness information. To do this, we add a new register called `PREV SUBR`. This register is written with the PC of a subroutine that is at the top of the RAS each time there is a return instruction. Then, we check whether this `PREV SUBR` register is not zero when there is a next subroutine call or return instruction to update the corresponding flag belonging to the previous subroutine. Thus, if there were at least one cache miss after the subroutine execution pointed by the `PREV SUBR` register, its `PSU` flag in the `SUBR TABLE` is set to one. Otherwise, this flag is reset to zero. Therefore, when applying this `PSU` technique and the useful saturated counter reaches zero, the runahead thread skips the subroutine execution if the `PSU` flag that indicates there were L1 misses after this subroutine is set or it stalls the runahead execution if this flag is zero.

In Figure 4.7, we illustrate the update process of this `PSU` technique. After subroutine A1 execution there was a cache miss. Then, when the next subroutine call is executed, B1 in this example, the `PREV SUBR` register is checked and used to update the A1 subroutine `PSU` flag in the `SUBR TABLE` corresponding entry. As there was a cache miss after the subroutine execution, the `PSU` flag is set. Later, this information will be used to choose the “skip” action if the subroutine A1 is consider useless.

- *Subroutine call example*

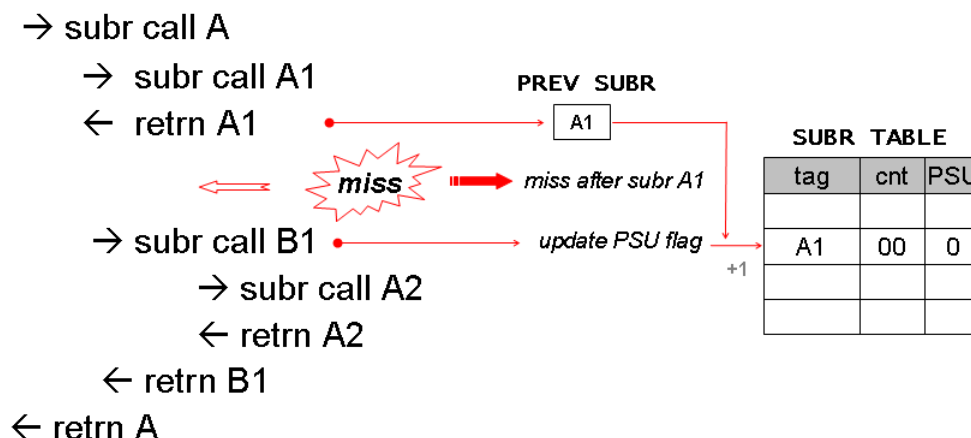


Figure 4.7: Subroutine `PSU` control technique

Hardware requirements

Regarding hardware components, the subroutine control techniques require a bit more hardware than loop control ones. These subroutine control techniques require a slight modification to the RAS in order to analyze the different subroutines usefulness according to the level of calls. Basically, this consists on adding a flag which represents one more bit per entry. Related to this, the RAS is not usually larger since most programs have relatively a small call-depth. For example, the Alpha 21464 processor has 32-entry RAS [16] whereas the Intel Core microarchitecture has two 16-entry Return Stack Buffer(RSB) tables [13]. This only entails 32 or 16 bits more for our RAS extension.

In addition, we introduce a new register `INSUBROUTINE` and a small table, `SUBR TABLE` to record subroutine information per hardware context. Only the PSU technique requires some additional bits into the `SUBR TABLE` and the `PREV SUBR` register. This tables have the double number of entries compared to the RAS. In our case, 64 entries each one per hardware context. All these structures needed for the different subroutine control techniques are summarized in Figure 4.8.

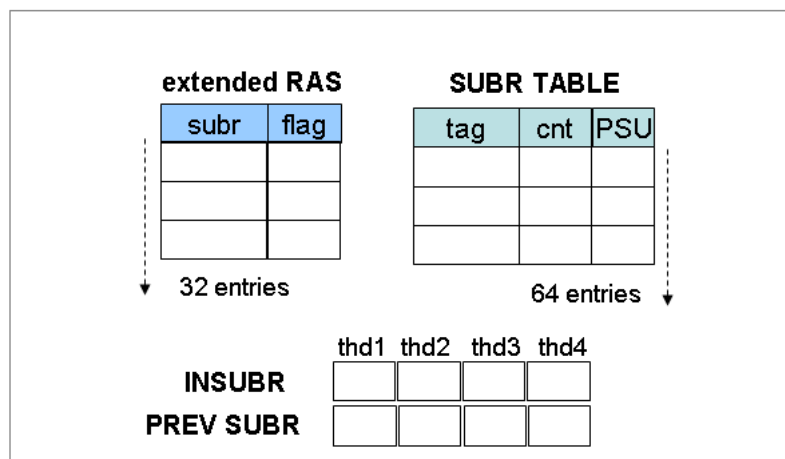


Figure 4.8: Hardware requirements of subroutine control techniques

In sort, the complexity of all these new structures are simple enough and it makes feasible all of the proposed techniques. None of these added structures are complex or are on the processor's critical path.

4.5 Loops and subroutines analysis

Before showing the evaluation results, in this section we show several data and statistics related to the analysis of branches and subroutines involved in the functionality of proposed techniques. Showing how many branches can be detected as representative of a loop or the possible number of subroutines to be controlled are useful information with a view to understand the different results later. For this analysis, we only employ the detection loop and subroutine mechanisms to get the corresponding results. In these case, none of the control actions (the last step in each technique) are done to simply obtain the reference values about these code semantic structures during runahead thread executions.

4.5.1 Loops in runahead threads

Among all the branches executed in the different threads, we are interested in the backward branches. This kind of branches are the candidates to be the origin of a loop. Figure 4.9 shows the percentage of backward branches detected with regard to all executed branches for the different workloads. For each set of workloads, we show the percentage of backward branches founded under normal thread execution and the same for runahead threads. Depending on the kind of workload, these ratios range between 10% and 20% of backward branches approximately. This ratio of backward branches follows a similar tendency independently of the thread is being a normal or a runahead thread. Although the percentages of runahead threads are a bit higher than normal threads, specially when memory-intensive threads are executed, since the execution spends more time in runahead mode, and therefore, more likely to detect more loops.

In Figure 4.10 we show the ratio of backward branches identified as loops in normal and runahead threads respectively for each kind of workloads with the detection loop mechanism. In the case of normal threads, the average is 33% for 2-thread workloads and 27% for 4-thread workloads, whereas for runahead threads these percentages are 25% and 19% for 2- and 4-thread workloads respectively. This difference is due to the lower number of loop identifications detected in runahead thread executions, specially in the case of memory intensive workloads as we can observe in the figure. One reason of this lower ratio is because there are memory bounded bechmarks with a higher number

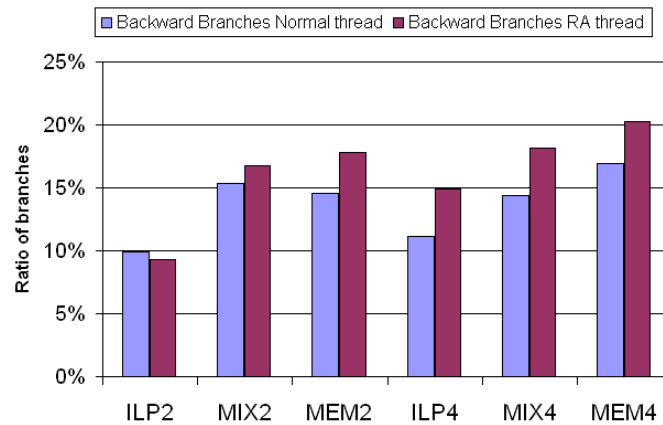


Figure 4.9: Backward branches founded during normal threads and runahead threads execution

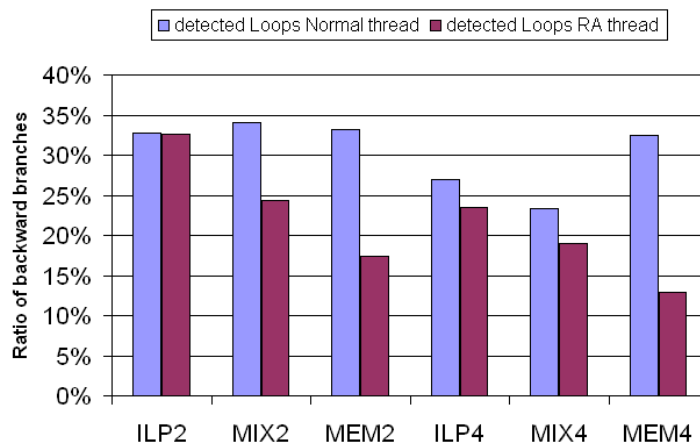
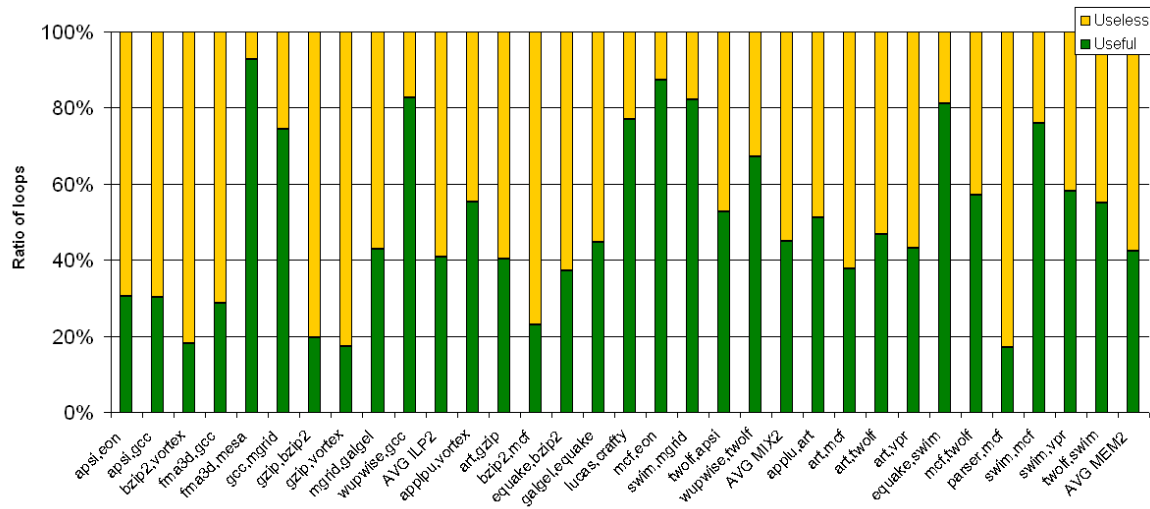


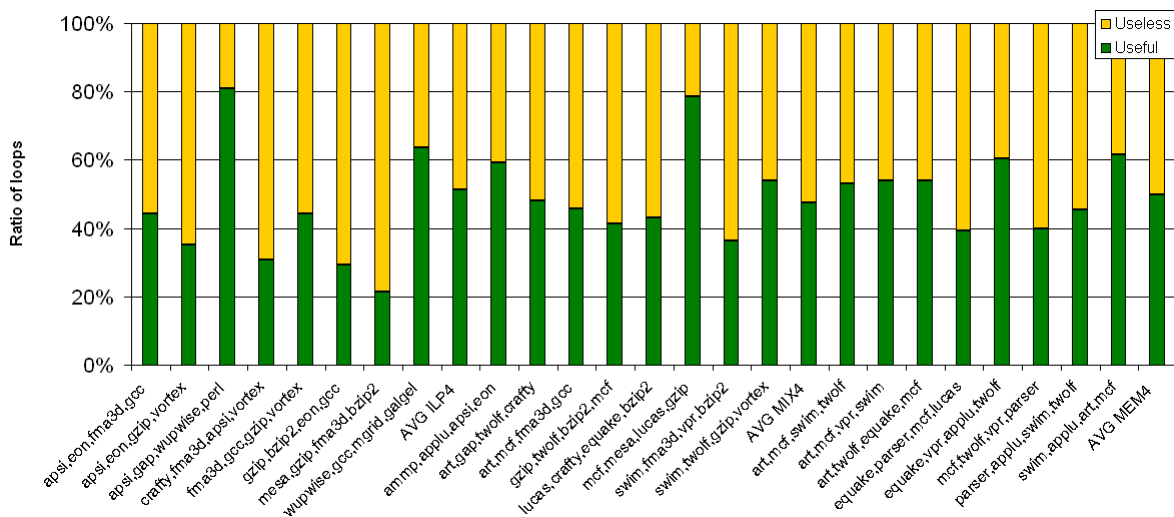
Figure 4.10: Loops detected applying our detection loop mechanism for normal threads and runahead threads

of backward branches dependent on long-latency loads (e.g. the *mcf* presents in some of these workloads). These dependent branches are invalidated and not executed as normal, therefore these backward branches are not taken into account to be identified as a loop branch reducing the percentage of loop detected for MEM workloads.

Finally, in the figures 4.11(a) and 4.11(b), we depict in detail the distribution of detected loops by our loop control mechanism during the runahead threads. For each workload, we show the fraction of loops classified as useful (that is, which contains L1 cache misses) and useless ones. Overall, the tendency of useful loop executions for both



(a) 2-thread workloads



(b) 4-thread workloads

Figure 4.11: Loop usefulness breakdown for 2- and 4-thread workloads

2-thread and 4-thread workloads is similar, with an average of 51% for 2 threads and 48% for 4 threads. This percentage is lower in the case of ILP workloads, since there are less cache misses, and then, more probability to find useless loops. Thus, if we detect and avoid the other percentage of loops which do not contain misses (49% and 52% on average respectively) in runahead threads, we will improve the RaT efficiency reducing the useless executions. In particular, there are workloads with a larger number of useful loops than others, such as *fma3d*, *mesa* (93%) *mcf*, *eon* (87%), and *equake*, *swim*

(81%) for 2-thread workloads or *apsi,gap,wupwise,perl* (81%) and *mcf,mesa,lucas,gzip* (78%) in the case of 4 threads. In the other extreme, there are workloads with a high percentage of useless loops as *gzip,vortex* (83%), *bzip2,mcf* (77%) and, *parser,mcf* (83%) for 2 threads and *mesa,gzip,fma3d,bzip2* (83%), *swim,fma3d,vpr,bzip2* (64%) and, *equake,parser,mcf,lucas* (61%) for 4 thread workloads. Controlling and limiting a big percentage of these useless loops allow the loop control techniques to enhance the RaT mechanism efficiency.

4.5.2 Subroutines in runahead threads

Now, we analyze the data and features of subroutines executed during the runahead threads. Figure 4.12 shows the percentage of detected subroutines according to normal and runahead thread executions related to the total program subroutines for each group of workloads. For the ratio of subroutines corresponding to runahead threads, we take into account the subroutines that are completely executed during runahead threads. That is, subroutines identified by the corresponding call instruction and return instruction in the runahead execution. We only consider valid for usefulness analysis the full subroutine executions since a runahead thread can start or end in the middle of a subroutine execution leading to incomplete usefulness information.

As Figure 4.12 shows, the percentage of subroutines detected in runahead threads is going up according to the workloads with more runahead thread activations (that is, from ILP to MEM workloads). This result is logical, since the more time in runahead executions, the more subroutines can be executed and detected. Likewise, the ratio of subroutines is similar for each type of workload independently the number of threads: 26% for ILP2 and 23% for ILP4, 46% for MIX2 and 45% for MIX4, and 71% for MEM2 and 74% for MEM4.

From the percentage of detected subroutines showed in the previous figure, Figure 4.13 shows the subroutines distribution between useless and useful ones for each workload in detail. Overall, the percentage of subroutines without cache misses (useless) is 72% and 74% for 2-thread and 4-thread workloads respectively.

In particular, there are workloads with more than 90% of useless detected subroutines, like (*lucas,crafty*), (*twolf,apsi*), and (*wupwise,twolf*) in the group of MIX2 workloads. For 4-thread workloads, all of them have a percentage of useless sub-

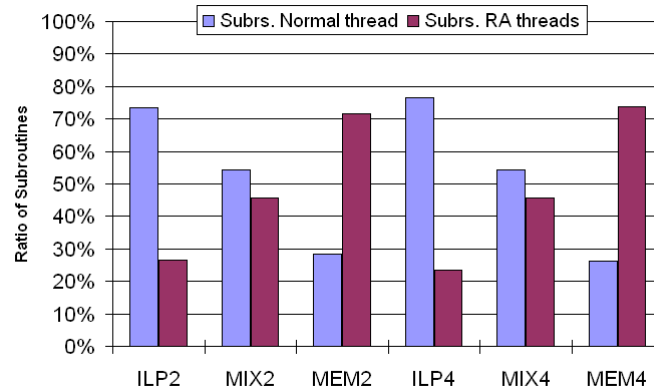


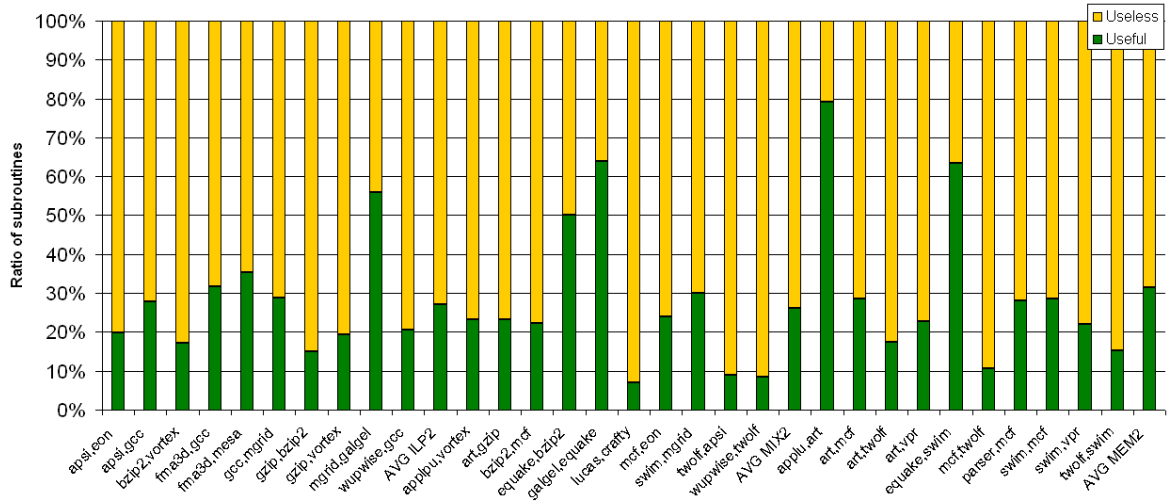
Figure 4.12: Subroutines detected in the runahead threads

routines higher than 50%, being (*lucas,crafty,quake,bzip2*) the lowest, with 53% and (*gzip,twof,bzip2,mcf*) the largest with 85%.

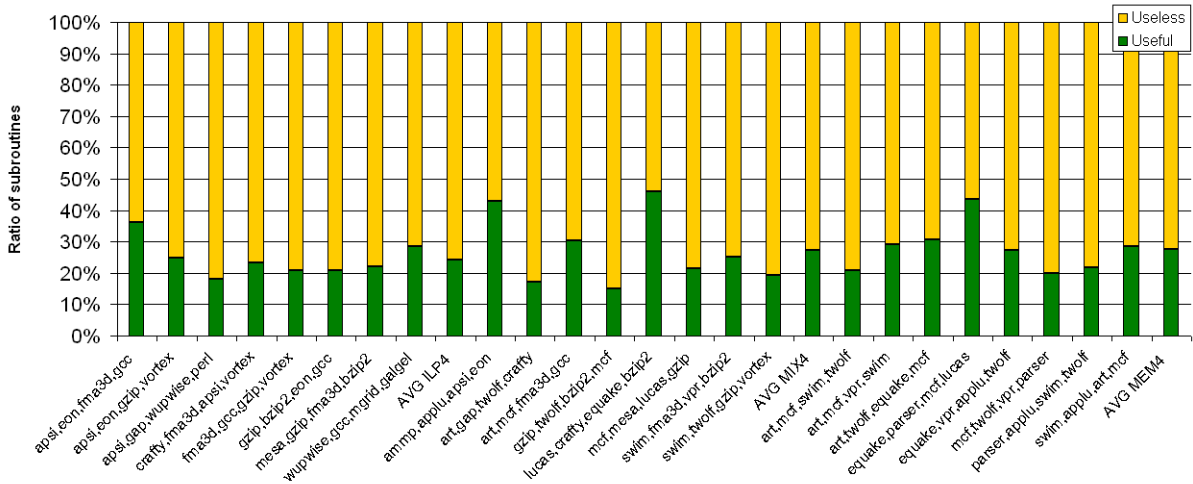
4.5.3 Statistics about the controlled loops and subroutines

To complete this analysis, we provide insights into how each technique manages the executed code semantic structure according with their different control actions. We show the percentage of these useless loops and subroutines that our approaches are able to capture dynamically. Figure 4.14 shows the percentage of reversed and stalled loops according to LR and LS techniques respectively. As reference, we also show the average percentage of loops previously detected as useless for each group of workloads as we analyze in section 4.5.1. Regarding this data, the ratio of loops stalled is lower than the reversed ones. LR technique continues executing the runahead threads instead of stalling as LS does, and then, there are more probabilities to find and control more loops. In this sense, the total average percentage of loops detected and controlled as useless is 43% for LR and 23% for LS. Both are lower percentages than 50% initial detected useless loops (see Figures 4.11(a) and 4.11(b) for more details), but the LR technique is closer to capture that initial ratio than LS technique. Although this data can seem a higher accuracy of LR technique, the difference is mainly due to the different strategy that causes executing and detecting a small number of useless loops by LS technique.

Likewise, Figure 4.15 shows the reference percentage of analyzed useless subroutines for the workloads along with the ratio of skipped and stalled subroutines when SK



(a) 2-thread workloads



(b) 4-thread workloads

Figure 4.13: Subroutine usefulness breakdown for 2- and 4-thread workloads

and SS techniques are used respectively. Like loops techniques, subroutines stalled is lower than skipped: there are 78% of subroutines that were detected as useless and skipped using SK and 64% of subroutines that were stalled with SS technique. However, SK technique shows a bit higher percentage of useless subroutines than the reference detected ones when no action over them is done, which percentage is 73%. This result is consequence of the skip action, that allows the mechanism to continue

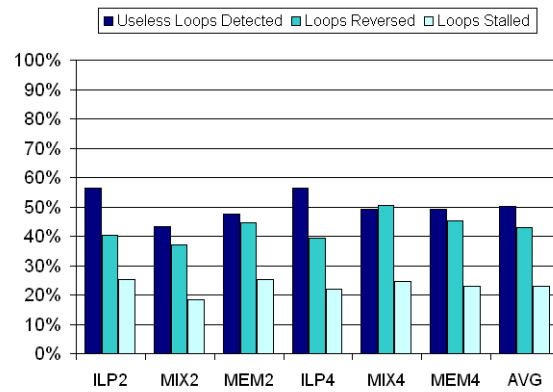


Figure 4.14: Percentage of loops controlled by LR and LS techniques

executing instructions during runahead and, thereby it detects and skips further and more subroutines.

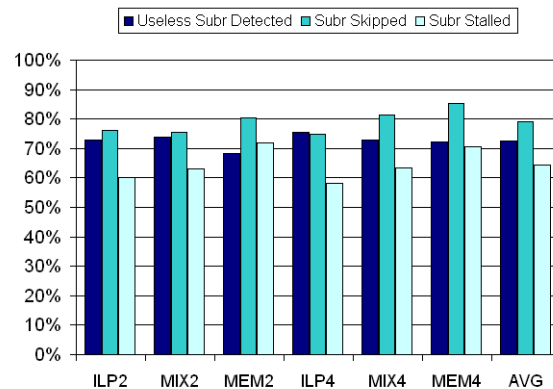


Figure 4.15: Percentage of subroutines controlled by SK and SS techniques

4.6 Evaluation of code semantic techniques

In this section, we firstly show the performance benefits of using the saturated counters for each code semantic-aware control technique. Next, we evaluate and compare the efficiency, in terms of the performance and extra work, of the different described proposals in this chapter.

4.6.1 Impact of using the saturated counters

First, we show the study of introducing the saturated counters to enhance the different code semantic control techniques. Figure 4.16 show the average performance throughput of the two loop control techniques, LR and LS, using only the usefulness load flag to take directly the corresponding control decision (LR Flag and LS Flag respectively) versus the same techniques using the additional saturated counter (LR SatCnt and LS SatCnt). This figure shows as each stall and skip based action techniques with the saturated counter get better performance results than not using it. In particular, LR technique using the saturated counters (LR SatCnt) is 14% better than LR Flag and LS SatCnt is 11% better than LS Flag. This performance difference is specially significant in the case of MEM workloads with up to 38% of performance gain.

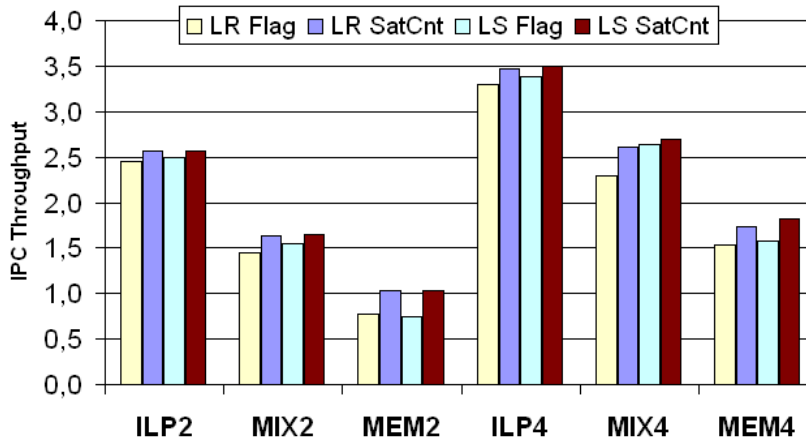


Figure 4.16: Performance differences when saturated counters are used for loop control techniques

Figure 4.17 shows the same study but for the subroutine control techniques. We compare the performance results of the SK and SS techniques when the control action is made directly with the usefulness flag (SK Flag and SS Flag) or it is made with the saturated counter instead (SK SatCnt and SS SatCnt respectively). In this case, the control techniques are less sensible to the utilization or not of the saturated counter but they also have a performance degradation. SK performs 5% better and SS performs 4% better with the additional level of confidence of the saturated counter.

Therefore, if the counter is not used, the code semantic control techniques rashly cause stall or skip some useful loop or subroutine executions. We have just shown that

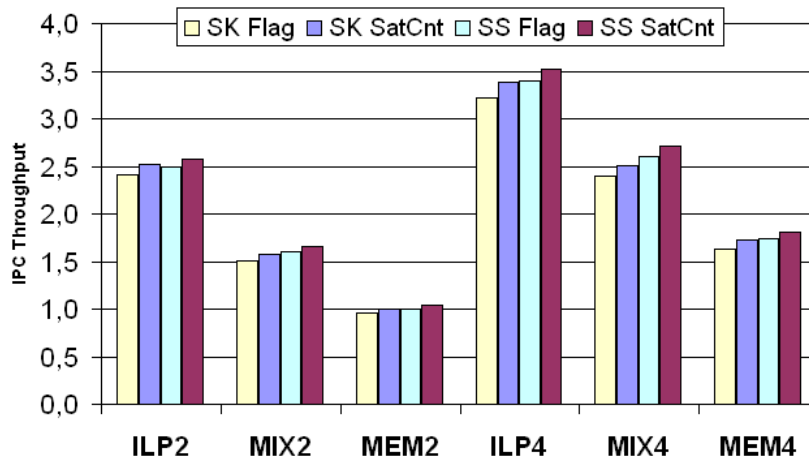


Figure 4.17: Performance differences when saturated counters are used for subroutine control techniques

this is detrimental to performance because it eliminates useful runahead instructions along with useless ones. The remainder of evaluation in this section are shown using the saturated counter versions for each particular technique since they present the best results.

4.6.2 Performance evaluation

To evaluate the level of success of our approaches with regard to the initial goal, we first analyze the performance as one of the factor in the efficiency relation. As the previous chapter, we evaluate the techniques performance in terms of total throughput and Hmean. Using these two metrics, we show both total system performance and user performance-fairness perception. For each graphic of this section, we show the performance results of all the two loop control techniques (described section 4.3): *Loop Reverse* (LR) and *Loop Stall* (LS), and the three subroutine control techniques (described in Section 4.4): *Subroutine Skip* (SK), *Subroutine Stall* (SS) and *Post-Subroutine Usefulness* (PSU). Note, we also show results for ICOUNT as the SMT processor baseline and our original RaT proposal for comparison purposes.

Figures 4.18 and 4.19 show the throughput and Hmean respectively grouped by the different kinds of workload (ILP, MIX and MEM). It is observed in Figure 4.18 as the performance throughput is very close to RaT for almost all proposed techniques. The worst proposal is Subroutine Skip (SK) which loses a slightly 4% on average compared

to RaT. The execution of more far useless instructions, as Figure 4.15 indicates for this technique, harms a bit the performance. However, there are other techniques, like Loop Stall (LS) or Subroutine Stall (SS), which achieve even throughput improvements over RaT in some cases, especially for MIX workloads. For instance, SS achieves a 3% improvement in MIX4 workloads regarding RaT. In this case, stalling the useless part of a code execution benefits the other threads that take profit of more available resources.

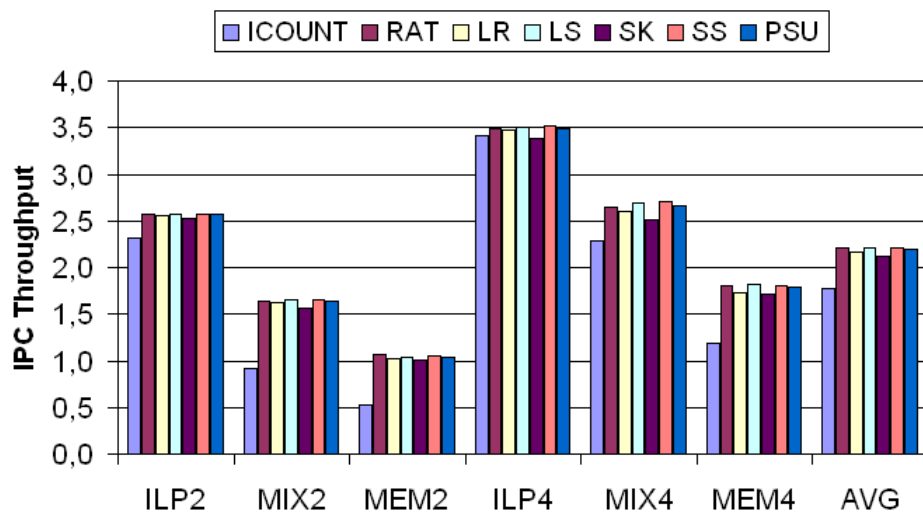


Figure 4.18: Throughput results for the code semantic-aware techniques

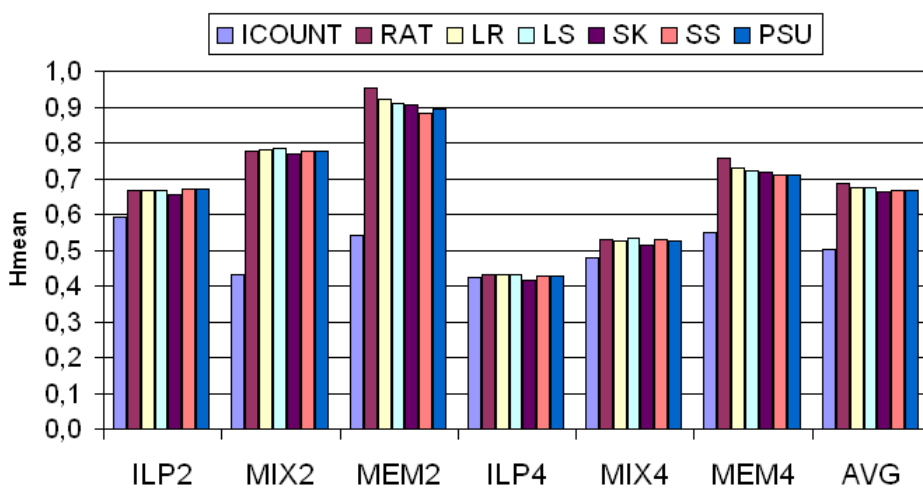


Figure 4.19: Hmean results for the code semantic-aware techniques

Regarding Hmean metric, Figure 4.19 shows that usefulness control techniques are more sensible to Hmean results than original RaT. There is an overall little Hmean reduction for all techniques in memory-intensive workloads (with a maximum of 7% of reduction with SS for MEM2). For the rest of workloads the Hmean results are similar compared to RaT. Even, there is a slight improvement (1% on average) for LS technique in the case of MIX workloads.

We also evaluate the combination of the different loop and subroutine control techniques to analyze the effects in performance when these are considered together. Figure 4.20 shows the average throughput for each code semantic-aware technique combination (the results are very similar among them in term of Hmean). This figure shows that the combination of the LS technique with PSU (LS+PSU) represents the best mix among the proposed techniques. According to these results, the average throughput for LS+PSU combination is 1% better than RaT.

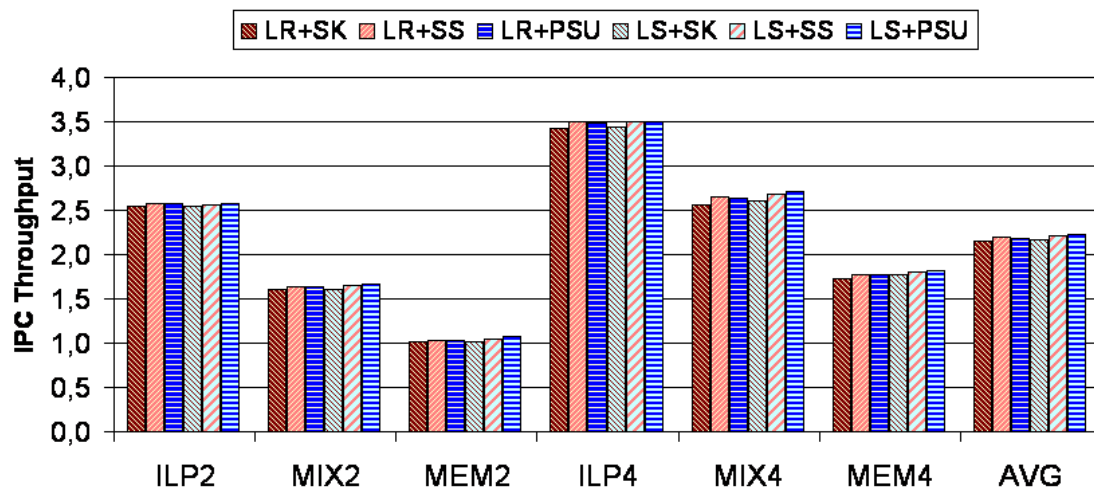


Figure 4.20: Throughput results for code semantic-aware technique mixes

For the rest of sections, we only show the LS+PSU combination which also represents the best results for the next evaluations.

4.6.3 Extra work

Once we have shown that proposed code semantic control techniques obtain similar performance to original no-controlled RaT, the other factor to enhance the RaT efficiency is consequently reduce the number of executed instructions under runahead

threads. Achieving this issue, we also reduce the power consumption as we show in this section.

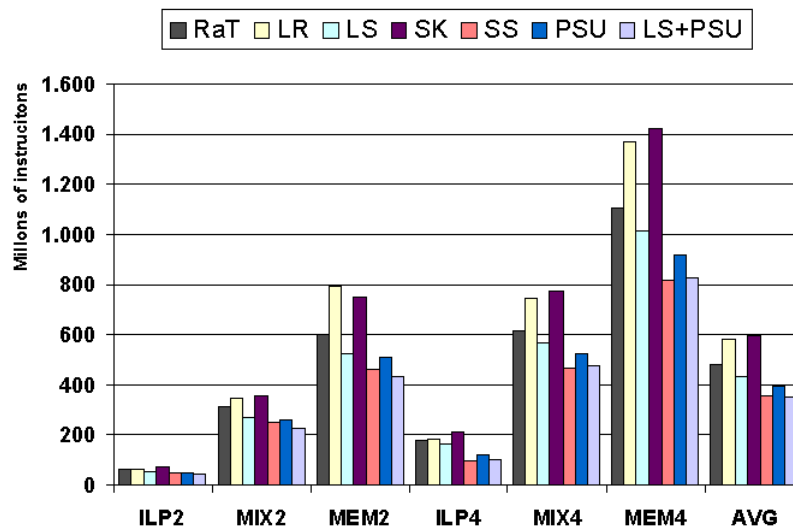
Speculative instructions

Figure 4.21 shows the number of speculative instructions executed by runahead threads for each evaluated mechanism (Figure 4.21(a)) and the percentage of these instructions when applying our control techniques compared to RaT (Figure 4.21(b)). As we can observe in these graphs, the results are quite different among the kind of proposals. While the stall action techniques (LS, SS) and PSU reduce the number of speculative instructions for all workloads (up to 48% in the case of ILPs for PSU), the other techniques (LR and SK) increment this amount (15% and 20% on average respectively). LS and SS cause activating more runahead threads (8% more) due to the stall action that avoids capturing more long latency load in some runahead thread executions. However, the stop action reduces significantly the number of extra speculative instruction executed, specially for MIX and MEM workloads as demonstrate Figure 4.21(a).

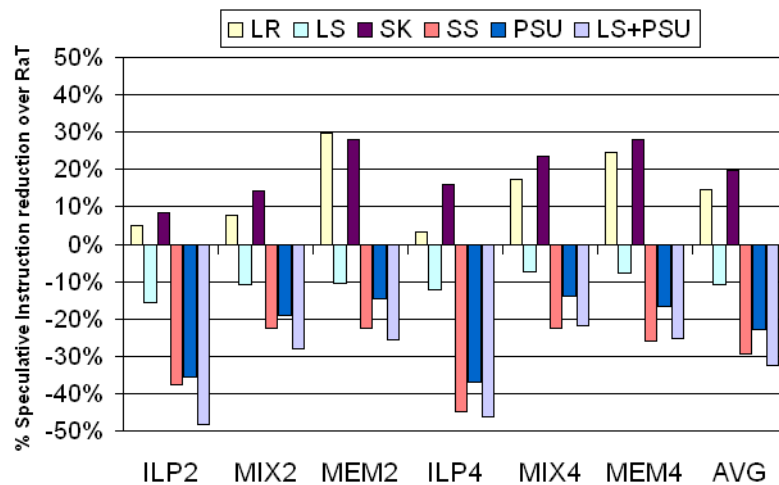
On the other hand, the LR and SK techniques execute a higher number of instructions since their control actions are allowing runahead threads execute faraway each time a loop is reversed or a subroutine is skipped. In the case of MEM workloads, these techniques executed between 25-30% more of speculative instructions. So, they are able to keep the RaT performance but they cause a negative impact on the extra work executed. Therefore, LR and SK are not a good choice to improve RaT efficiency, being the other techniques based on stall action more suitable taking into account these results. Among the best performing techniques, we remark the combination of LS+PSU which achieves 33% reduction on average in the amount of speculative instructions executed compared to RaT. This mechanism exploits the benefits of both techniques (LS and PSU).

Power consumption

Finally, we quantitatively evaluate the power reduction obtained by the different techniques. To measure the power estimation, we use the implemented power model based on Wattch integrated in our simulator. We give results about power reduction compared to RaT for the different mechanisms evaluated that we show in Figure 4.22. This power results depict a similar trend to instruction reduction results from Figure



(a) Number of speculative executed instructions



(b) Speculative executed instructions ratio

Figure 4.21: Speculative executed instructions analysis in function of the code semantic-aware techniques

4.21. Note that MIX and MEM workloads power consumption are reduced more than ILP ones in spite of the higher percentage of speculative instruction reduction previously shown. The instruction percentage represents only a ratio but the average power (which is correlated with energy) is very much affected by the number of executed

instructions. Thus, a higher quantity of instructions reduced (no the percentage) in the cases of MIX and MEM (see Figure 4.21(a)) produces a higher energy savings.

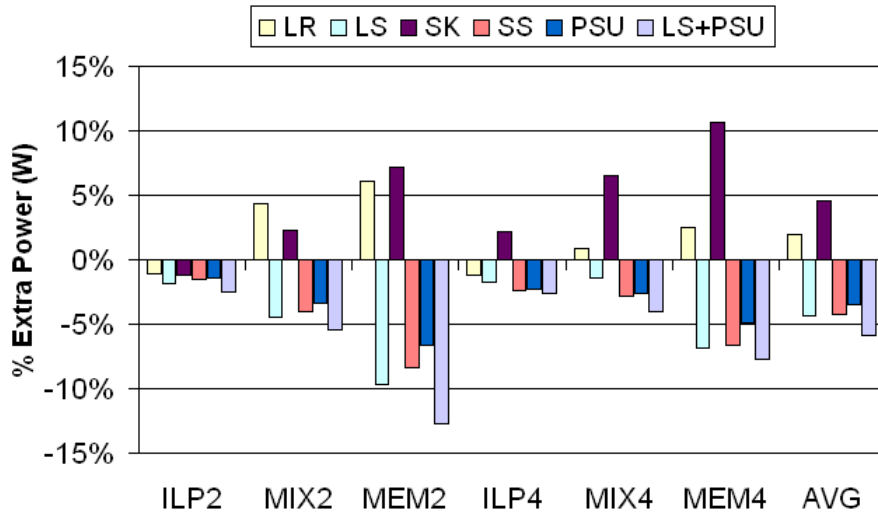


Figure 4.22: Average extra power reduction

Regarding the techniques in particular, whereas LR and SK get a slight average power increment (2% and 4,5% respectively), the LS+PSU technique achieves the best result with an average 6% power reduction compared to RaT. Also, LS and SS techniques reduce the power consumption close to 5% each one.

4.7 Summary

Runahead Threads mechanism generates an excess of speculative work that produces an energy overhead. If this overhead does not improve the processor performance in reward, the performance-power efficiency of RaT is diminished. To avoid this shortcoming, we have presented code semantic-aware techniques that enhance RaT mechanism in a more efficient way. Basically, these mechanisms are based on runtime analysis to detect code patterns which identify loops or subroutines. Likewise, this coarse-grain analysis oversees the usefulness of loops or subroutines depending on the prefetches opportunities during runahead thread execution. By means of this information, different control techniques decide either stall or skip the loop or subroutine executions to reduce the number of useless runahead instructions.

These techniques achieve comparable performance with regard to original RaT mechanism, even improving the performance of RaT (1%) in some cases. The evaluation in this research shows that the techniques that stall the useless loop or subroutine execution present better results from efficient point of view. The best combination of these techniques results in a performance-efficient mechanism that maintains the performance improvement of RaT while reducing the extra speculative work required (33% less speculative instructions on average) and power consumption (6%).

Efficiency-aware Runahead Threads

As we explained in the previous chapter, RaT benefits come at the cost of executing a large number of instructions speculatively due to runahead execution what leads to an extra power consumption. The proposed code semantic-aware techniques reduce part of that useless extra work by performing coarse-grain analysis of code patterns executed by the runahead threads. Thus, these techniques improve the runahead thread efficiency avoiding the useless speculative work of RaT by means of overseeing the usefulness of loops and subroutines during runahead mode. Therefore, the effectiveness and potential energy reduction for them are highly dependent on the presence of loops and subroutines during the runahead thread executions.

In such sense, code semantic-aware runahead threads are effective but specific for dealing with loops and subroutines. The ability to make runahead threads more energy efficient using these code semantic-aware techniques are determined by the amount and features of that loops and subroutines. The kind of software or the way the programmer develops an application can determine this fact. In addition, some compiler techniques, such loop unrolling or subroutine in-lining, can make that high-level language loops or subroutines presented in the original source code are not represented as such in the optimized machine code generated by the compiler. Hence, aggressive compiler optimizations may diminish the number of loops or subroutines in the final binary code. Therefore, code semantic-aware techniques are effective in function of the program features, and not the own runahead thread features.

In this chapter, we devise a more generic scheme to enhance the efficiency of RaT. This scheme predicts the efficiency of the runahead threads based on the execution of a particular runahead thread and does not what type of code are being executed. Thus, useless runahead executions will be detected independently of the executed code patterns. The key idea behind this different scheme is to perform a fine-grain analysis of each runahead thread to collect information focused on optimized the speculative work done. Based on this information, it is possible to predict how far a thread should run ahead speculatively such that speculative execution will be efficient.

5.1 Runahead distance prediction

As we have already pointed out in previous chapters, the usefulness of a runahead thread is given by the total amount of prefetching that a particular runahead thread is able to exploit during its speculative execution. Following the goal to make RaT mechanism more energy-efficient, we propose new approaches that try to analyze the number of long-latency loads per runahead thread to balance between useful prefetching and useless instructions. In other words, we want to dynamically find out the useful lifetime of a runahead thread to expose as much MLP as possible with the minimum number of speculative instructions.

5.1.1 Useful runahead distance

A full runahead thread execution consists of the total number of instructions that a particular thread executes from the load that triggers the runahead mode to the last runahead instruction executed before flushing the pipeline when the L2-miss is resolved. Figure 5.1 illustrates an execution example of a runahead thread generated by a long-latency load miss. This figure depicts all instructions executed by a particular runahead thread (indicated by vertical short lines), remarking (with arrows) the long-latency loads (labeled as LLLi). If we observe this runahead thread snapshot, after the execution of LLL4, there are no more long-latency loads until the point the runahead-causing load is serviced, that is, the end of this runahead thread execution. Intuitively, executing a runahead thread beyond the LLL4 instruction does not provide significant performance gain and, in addition, it wastes energy.

Therefore, we define *useful runahead distance* as the number of executed instructions during a particular runahead thread between the long-latency load that triggers the runahead thread and the last speculatively-executed load instruction that caused an L2 miss (i.e., LLL4 in the example). According to our analysis, there are 12% of static loads that generate the 90% of runahead threads (similar results for another work related to cache misses [15]). Therefore, we can track the runahead distance of the vast majority of runahead threads for a small number of static loads in function of this data.

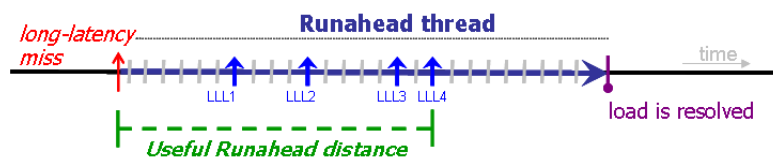


Figure 5.1: Illustration of runahead thread execution and the concept of *useful runahead distance*

Based on these observations, we develop some proposals that predict and control that a runahead thread executes instructions only up to the useful runahead distance. The idea is to capture dynamically the *useful runahead distance* in order to execute the useful part of each runahead thread associated to the long latency loads. Combining RaT with this runahead distance prediction mechanism, the former exploits the maximum possible MLP up to the point set by the useful runahead distance and at the same time avoids the useless execution of the remaining instructions until the end of the runahead thread. In addition, in case the predicted useful distance is small, this mechanism also avoids activating a runahead thread to not execute useless runahead instructions.

Doing so would have three major benefits: 1) it would reduce the extra energy consumption due to avoid the execution of useless speculative instructions, 2) it would reduce the pressure useless speculative instructions exert on shared SMT resources, thereby allowing other threads in the processor to make faster progress and thus improving the utilization of the SMT system, 3) it would minimize any other possible cause of performance degradation (e.g., due to cache pollution or bandwidth interference) that might be caused by executing useless runahead instructions.

5.1.2 The first approach

To execute a runahead thread only up to the useful runahead distance, we firstly propose a mechanism called *Runahead Distance Predictor (RDP)*. This technique predicts most of the usefulness of runahead threads create by frequent long-latency loads based on the previously-observed useful runahead distances. So, RDP records the useful runahead distances of previous runahead executions caused by the same static load instruction to predict the useful runahead distance of future runahead threads. By means of this prediction, RDP helps the processor to decide (1) whether or not a runahead thread should be initiated on an L2-miss load, and (2) if runahead thread is initiated, how long the runahead execution should be, thereby avoiding the execution of useless runahead periods. With this prediction mechanism, we would eliminate all useless runahead threads that do not provide prefetching as well as the portions of runahead threads that do not provide performance benefits but unnecessarily cause wasteful executions.

We introduce in the SMT processor a hardware table called Runahead Distance Information Table (RDIT) to implement the Runahead Distance Predictor. Each entry of the RDIT holds the useful runahead distance information associated to a static load for a particular thread. The RDIT entry associated for the long-latency load is updated once its particular runahead thread finishes. So, in each runahead thread execution, RDP computes the runahead distance while tracking the last L2 miss load that is issued. When the runahead thread is finished, RDP stores the corresponding useful runahead distance in the RDIT entry associated with the L2-miss load that caused the runahead thread. If there is at least one such long-latency load issued during the runahead thread, the useful runahead distance must be different of zero. If there are no L2 misses in that runahead thread, the associated runahead distance is zero.

RDIT is accessed in the execution stage for every load that misses in the second level cache (a candidate for causing a runahead thread). For this case, a particular runahead thread would be useful if the RDIT entry associated to that load has a runahead distance greater than zero, since it indicates the existence of long-latency prefetching (that is, it presents some MLP). Moreover, if the computed useful distance is large enough, the runahead thread can prefetch distant data even further than the instruction window size. In case of a zero runahead distance, however, a runahead thread is not initiated and this thread will be stalled from fetching further instructions

until the load is resolved. As side effect, the remaining processor resources will be available to the other threads.

RDP Operation

In Figure 5.2 we show a flowchart that depicts the main steps related to the RDP mechanism. The first time a static load accesses the RDIT, the table has no distance information for the load. For this reason, when that load reaches the head of the reorder buffer, a runahead thread is activated (i.e., it turns the normal thread into a runahead thread). Since no useful runahead distance information exists for that load yet, the runahead thread is executed until the L2 miss is fully serviced. At the end of the runahead thread, the load that caused entry into runahead execution allocates a new entry in the RDIT and stores its computed useful runahead distance.

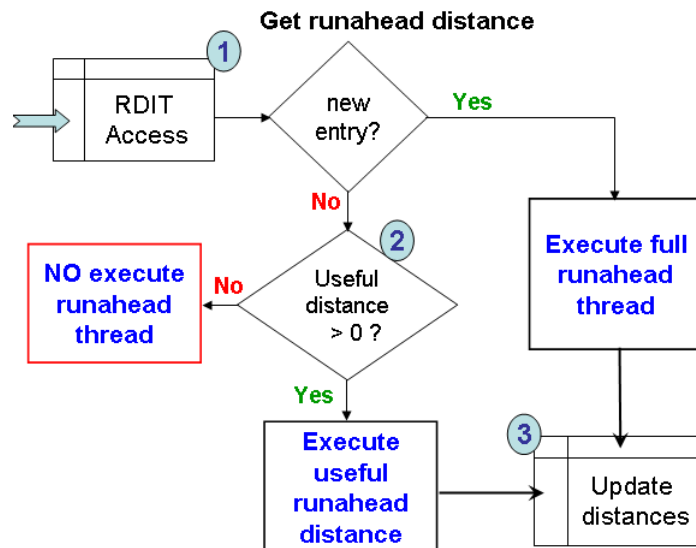


Figure 5.2: Flowchart of Runahead Distance Prediction (RDP)

When the same load misses in the L2 cache again during the normal thread execution, RDIT is accessed to obtain the useful runahead distance associated with it. Based on this useful runahead distance, RDP makes a decision. If the useful runahead distance is zero (i.e., there were no long-latency loads in the last runahead thread created by that load), RDP does not start a runahead execution thread. Otherwise, the processor turns the normal thread into a runahead thread, which executes as many instructions as the useful runahead distance points out for this load. The distance

comparisons to check the termination condition of the runahead thread are performed concurrently at the retirement stage during the runahead speculative pseudo-commit. Once the runahead thread executes as many instructions as the useful runahead distance predicts, the RDIT entry is again updated for that runahead-causing load while the thread is flushed to resume normal execution.

5.1.3 The second approach

RDP is an initial scheme of a runahead distance predictor that guides the runahead thread executions based on the last observed useful runahead distance. This functionality causes the next runahead thread for a given load never exceeds the useful runahead distance computed in the previous runahead execution. So, the useful runahead distance associated with a load instruction either stays the same or monotonically decreases during the different executions of its runahead thread. However, we observe that the useful runahead distance can unexpectedly vary during the execution of the thread. According to our analysis with regard to the useful runahead distance behavior, on average, 58% of times runahead distance changes (up and down). As a result, if the runahead threads initiated by a load change to a larger useful runahead distance in the future, this cannot be detected because there is no way to learn increasing runahead distances in the previous first approach.

Another important case is when the useful runahead distance at any runahead thread for a load results zero (no long-latency load has been detected). After this point, a runahead thread is never initiated due to that load even though runahead periods might become useful in the future. The worst case would be when the runahead distance is zero the first time, since we are banning these loads from initiating a runahead thread in the rest of misses during the execution. There are several cases in which the useful runahead distance becomes larger after having values of zero (16% of cases for MEM workloads on average). If the runahead distance predictor cannot take into account these cases, it would lose performance improvement opportunities by eliminating a large number of overlapped runahead prefetches.

To avoid this unwanted behaviors, we refine our initial approach to include several modifications that make it more accurate and flexible in order to predict the useful runahead distance. Following subsections describe the different refinements applied to this second approach for the useful runahead distance prediction.

Improving the reliability of the useful distance prediction

This second approach maintains the philosophy of the useful runahead distance concept but we introduce a new level of decision to enhance reliability (i.e., confidence) of the predicted runahead distance. Now, instead of recording only one (the last) useful runahead distance in the RDIT, this mechanism works with two runahead distances per load: the useful runahead distance obtained in the *full*-length runahead execution periods and the *last* useful runahead distance. The former (full runahead distance) is updated each time a runahead thread is fully executed whereas the latter (last runahead distance) is always updated per each runahead thread execution. We call this second approach *Runahead Two Distance Predictor* (R2DP) due to this feature.

R2DP uses these two distance values as an indication of how reliable runahead distance information stored for a particular load is. When a load misses in the L2 cache, the R2DP computes the absolute difference between the *last* and *full* distance values and compares the result to a threshold value, called Distance Difference Threshold (DDT). If the distance difference is larger than DDT, R2DP treats the stored distance information as not reliable. In this case, a runahead thread is executed until the L2 miss is fully serviced in order to update the ‘unreliable’ stored distance values (allowing the mechanism to store a possibly higher distance value in the RDIT). Both the *last* and *full* useful distance values are updated at the end of this new full runahead thread execution. On the other hand, if the difference is smaller than DDT, then the stored distance values are considered to be reliable. The *last* useful runahead distance value is used to decide whether or not and how long the runahead thread should be executed. Using this enhancement, R2DP can correct the useful runahead distance value if the last distance becomes significantly different than the previous ones.

We examine several different threshold values for DDT. Although all these thresholds resulted in better performance and efficiency than the initial RDP proposal, we found that a threshold value of 64 obtains the best tradeoff between performance and extra work reduction. We will show results about this analysis in section 5.2.1.

Deciding whenever starting a runahead thread

As we described, RDP decides to start a runahead thread if the value of the useful runahead thread is not zero. If it is zero, the thread is not turn into a runahead mode. However, there are also useful runahead distances whose values are so small

that starting a runahead thread is not necessary. For instance, executing 10 runahead instructions to issue only one long-latency load. All the process to create the checkpoint, start the runahead mode and restore the context state would not be worthwhile in this case. Generally, the normal execution advance in the reorder buffer is enough to capture these close loads with less complexity.

For this second approach, we propose to start a runahead thread when the predicted useful runahead distance is above a particular activation threshold. We select this threshold based on a study about the number of instructions that can be executed from the time a candidate runahead load is detected to the time the load reaches the head of ROB. The extreme values for this study are between 0 and 113 instructions on average for the different threads in our SMT model. Thus, we set the threshold value in 32 instructions based on the global median for this study results. Therefore, if the useful distance is below this value, it means that the possible (near) MLP can be exploited without entering runahead execution, since other possible loads would be issued in the short period till the long-latency load reaches the head of the ROB.

Avoiding incorrect zero distances

Finally, we focus on resolving the limitation of runahead threads predicted with unsettled useful runahead distance value of zero. Certainly, many runahead periods caused by a load do not provide any prefetching benefits, i.e., their useful runahead distance values are zero. However, there is also certain variability in the usefulness of some runahead threads, with useful runahead periods (with non-zero useful distance values) interleave with useless runahead periods (with zero useful distance values). Our study shows that 38% on average of runahead threads have a zero distance value at least in one runahead execution, and then, 26% of them increase their distance value later. According to this study, using the RDP approach, the useful runahead distance value can be set to zero incorrectly, thereby making it impossible to find a larger useful runahead distance even though one might exist later.

To avoid this problem, we propose an additional modification based on a simple heuristic built on the previous improvement. This heuristic consists of the following: while the computed useful runahead distance results lower than the previous starting runahead thread condition (e.g. a distance less than 32) in the last N times for a load instruction, R2DP discards that prediction and the processor initiates full runahead execution for that load instruction. At the end of this full runahead execution,

the runahead distances (full and last) are updated with the new obtained distance. Therefore, R2DP can update the RDIT information with larger useful runahead distance in case it happens. Nevertheless, note that we enforce a limit of N attempts for this process, because if we always enable full runahead execution, we will ignore when the runahead periods are truly useless, diminishing the capability of this approach to reduce the useless extra work.

After performing an exploration of the design space, that we show in Section 5.2.1, our experiments show that a value of $N=3$ performs well in terms of performance and efficiency. To implement this heuristic, we simply add a 2-bit counter per-entry in RDIT to perform the countdown from 3. This complementary heuristic essentially provides an additional confidence mechanism that determines whether or not a small distance is reliable.

Operation of R2DP

Figure 5.3 shows a flow diagram that summarizes the R2DP mechanism procedure of this second approach. The flowchart shows the different steps from the RDIT access to the runahead distance update process once runahead execution ends. As before, the first time a static load misses in the L2 cache, RDIT has no useful information. For this reason, the corresponding runahead thread is executed until the L2 miss is fully serviced. RDIT updates both full and last useful distance fields with the observed useful runahead distance in this case.

Later, when that load misses in the L2 cache during normal thread execution once again, RDIT is accessed to decide about the execution of the possible runahead thread based on the information associated with it (1). Firstly, R2DP checks whether the recorded distances are reliable (2). To do this, R2DP computes the difference between the *full* and the *last* runahead distance values. If the difference is larger than the DDT (too much distance between them), a runahead thread is executed until this L2-miss load is fully serviced since they do not fulfill the two-distance reliability condition ($full - last < DDT$). Both *last* and *full* useful distance values will be updated again at the end of this particular full-length runahead thread execution.

On the other hand, if the last and full distance difference is small, then the distance values are considered to be reliable. In this case, the *last* runahead distance value is used as the current *useful runahead distance* to decide whether or not to start the runahead thread and to control how long the runahead thread should be executed

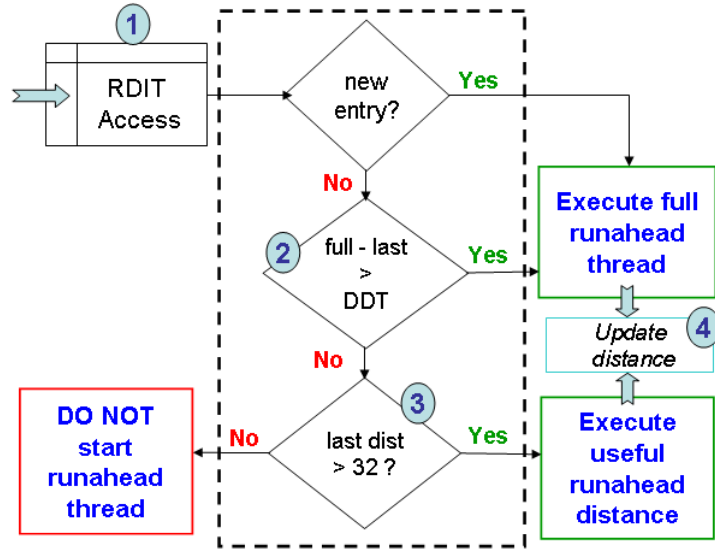


Figure 5.3: Flowchart of Runahead Two Distance Prediction (R2DP)

(3). R2DP decides to start a runahead thread if the value of the last useful runahead distance is above the activation threshold. If so, R2DP turns the normal thread into a runahead thread to execute as many instructions as the last useful runahead distance indicates.

Finally, once the runahead thread executes as many instructions as R2DP predicts, the mechanism updates the corresponding runahead distance fields in the RDIT (4) for the runahead-causing load and the thread is restored back to resume normal execution.

Using all described refinements in the runahead distance predictor scheme, this new approach is able to, first control possible variation of the runahead distances and, second, to enhance the reliability (i.e., confidence) of the computed runahead distance.

5.1.4 Implementation issues related to runahead distance techniques

The new required structures are simple and present a cost-effective hardware with regular behavior for both the first and the second proposal. Basically, in both techniques, the processor needs to track the last-issued L2 miss load in each runahead thread execution to compute the useful runahead distance. To this end, we introduce the use of two new components per hardware context: the *total-runahead-distance counter* and the *useful-runahead-distance register*. The first one is incremented per each instruc-

tion pseudo-retired during a runahead period. The second one holds the last-observed useful runahead distance for a particular thread execution. Since we consider a maximum distance of 1024 instructions, there is enough using 10 bits per each one to store such information. Thus, when a runahead thread starts, both registers are set to zero. Then, whenever a long-latency load is encountered during runahead execution, the total-runahead-distance counter value is copied into the useful-runahead-distance register as Figure 5.4 shows. This procedure continues until the runahead thread ends and then, the last value of useful-runahead-distance register is used to update the RDIT.

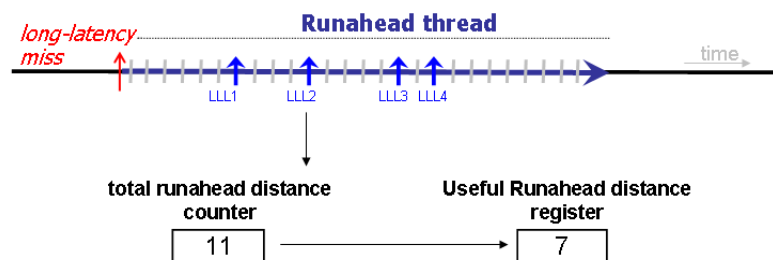


Figure 5.4: total-runahead-distance counter and useful-runahead-distance register functionality

Aside from the operation process of each approach, they only differ on the hardware requirements of the Runahead Distance Information Table (RDIT). So, according with the design of each proposal, the content of each RDIT entry are different. For the RDP approach, each RDIT entry simply consists of three fields: the load tag, the thread identifier and the useful runahead distance (23 bits in total per entry).

In the case of R2DP, each entry of RDIT is extended to store the following fields: the tag, the thread identifier, the two runahead distances: the *last* and the *full* useful runahead distances and 2-bit counter for the zero distance heuristic (35 bits per entry).

After performing a design exploration with different RDIT sizes and associativity, the final RDIT configuration consists of a 4-way table of 2048 entries in total shared by all threads. This setup results in the best tradeoff between the performance, prediction accuracy and aliasing rates. Therefore, the final size of RDIT is different depending on the corresponding approach, RDP or R2DP. For RDP the total size is 46Kbits (5,75KB) whereas for R2DP the total RDIT size is 70 Kbits (8.75KB). In both configurations, the total size is much less than 1% of the L2 cache area.

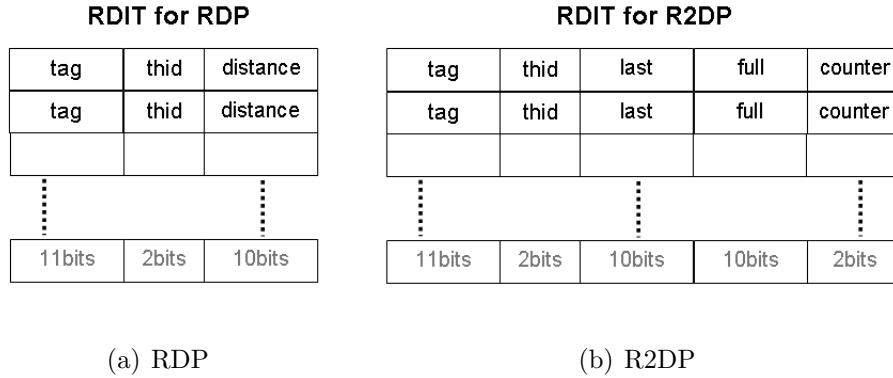


Figure 5.5: Runahead Distance Information Table configurations

We use CACTI [69] to determine the access time to the RDIT in each case. According to CACTI results, the RDIT access/update time fits comfortably in the cycle time of the reference SMT processor. Besides, we want to note that the access and update of RDIT are out of the critical path of execution. First, RDIT is accessed when a load misses in the L2 cache, which is a relatively infrequent event, and the required runahead distance prediction is not needed till the load reaches the head of the ROB. Second, RDIT is updated when the processor terminates a runahead thread. So, RDIT update latency can therefore be overlapped with the pipeline flush at the end of runahead thread. Therefore, RDIT does not need to be tightly integrated into the processing core since runahead threads execution are not sensitive to RDIT access latency. In addition, as the useful runahead distance can depends on the program path leading to the L2-miss load instruction, the RDIT is indexed by an XOR function of the program counter of the load and 2 bits from the two previous conditional branches history.

5.2 Evaluation of runahead distance mechanisms

In this section, we evaluate the runahead distance predictor mechanisms previously described when they are used to control the runahead thread efficiency. The experimental evaluations and results presented in this section show how perform the runahead distance predictor proposals compared to the RaT mechanism in terms of performance, extra work and energy efficiency.

5.2.1 Analysis of refinements applied to R2DP technique

Firstly, we show the different studies related to the refinements introduced in the R2DP approach to obtain its final configuration before showing the performance and energy results about the runahead distance proposals.

Distance difference threshold

The value of the DDT fixes the reliability margin of the two runahead distances for a particular load when the R2DP is applied. Figure 5.6 shows the relation of performance improvement and extra work reduction of the different evaluated DDT values for R2DP with regard to RaT mechanism. The curve of performance represents the average performance throughput difference for all workloads of the different R2DP configurations compared to RaT. The curve of extra work depicts the percentage of speculative instruction reduction for the same experiments compared to RaT as well. We evaluate the DDT with values that range from 0 to 128. A DDT=0 is a very strict threshold in which only when the last and full runahead distance are exactly the same, they are considered reliable. Larger values involve a more flexible condition in this sense, therefore allowing runahead threads with different variability in terms of performance and extra work depending on computed runahead distances. For comparative purposes, we also show in the figure the RDP results (first marks on the left).

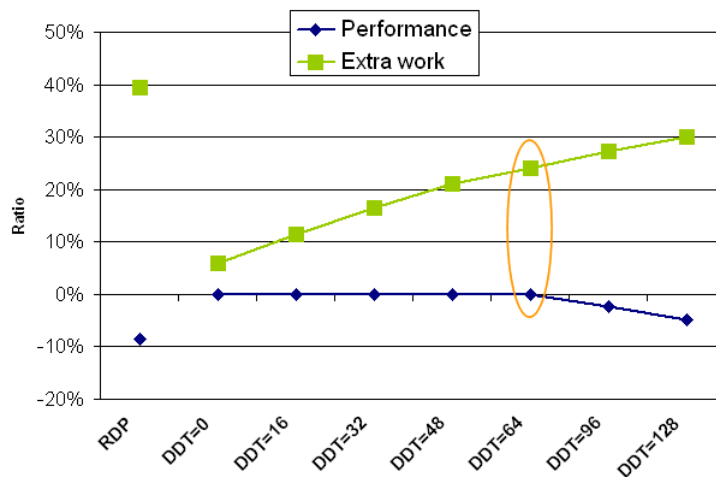


Figure 5.6: Performance (throughput) and extra work (speculative instructions) ratio with regard to RaT for different DDT values

All the tested thresholds for the R2DP result in better performance than the first RDP proposal as Figure 5.6 shows. When the DDT is increased, the extra work is reduced, since the obtained useful distances fulfill easier the DDT condition for larger values. So, executed runahead threads become shorter due to this less strict control that allows more differences among full and last useful runahead distances. However, for DDT values larger than 64, the R2DP starts to lose performance with regard to RaT. If the threshold is bigger, the difference between full and last useful distances can be bigger as well. In these cases, the last useful runahead distance can be decremented with larger margins similar to RDP technique. This results in inaccurate useful runahead distances which cause the R2DP technique reduces the prefetching opportunities for the long-latency loads that cannot be overlapped as in the case of RDP. This generates more runahead threads due to future L2 cache misses that could not be avoided by useful previous prefetches which impact on the performance. Therefore, in order to keep the performance of the original RaT mechanism, we chose the DDT=64 as the threshold value with the best tradeoff between performance and efficiency based on the results for these experiments.

Control heuristic to manage the distances with zero value

The initial RDP technique has the drawback that when a useful runahead distance value is zero for a long-latency load, possible subsequent runahead threads for that load would never be initiated again even though runahead periods might become useful in the future. We resolve this shortcoming for small useful runahead distances with a control heuristic implemented in the R2DP technique. Unless the runahead distance has been lower than starting runahead thread condition (e.g. a distance less than 32) in the last N times for a load instruction, we initiate a full runahead execution for that load instruction. With this heuristic, R2DP updates the useful runahead distance with new larger distance in case it exits.

We evaluate different values of N to obtain a configuration that performs well in terms of performance and efficiency. Selecting a particular value for this heuristic depends on the balance between performance and extra work. Figure 5.7 shows the ratio of performance and extra work with regard to RaT mechanism after performing an exploration of the design space for values of N from 0 to 3. These experiments show that R2DP with a value of $N=3$ for this heuristic is the only configuration that achieves speculative reduction (25%) without performance loss.

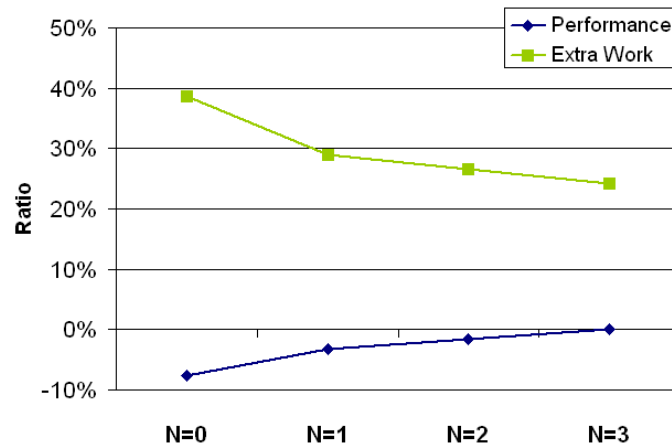


Figure 5.7: Performance (throughput) and extra work (speculative instructions) ratio with regard to RaT for different values of N for the zero distance control heuristic for R2DP

In Figure 5.8, we show the percentages of accesses to RDIT which result in a useful runahead distance of zero for the RDP and R2DP with the different values of N evaluated before. This represents the percentage of runahead threads not initiated due to zero-value distances for each technique. Increasing the value of N, R2DP considers less reliable more small useful distances causing more full runahead thread executions. So, R2DP reduces the possibility that a useful runahead thread will not be executed for greater values of N. From this figure, we note that RDP and R2DP with N=0 are not the same and then they have different results due to the reliability distance condition of the latter ($full - last < DDT$). If the full distance is greater than DDT and the last distance becomes zero, they do not fulfill the two-distance condition and then, the distances will be updated according to the R2DP functionality.

For N=3, R2DP executes 20% on average more runahead threads than they would not be started with RDP, thereby reducing the times that RDP completely eliminates the execution of a runahead thread incorrectly for a particular load. These data show as R2DP with this additional heuristic is able to better tolerate dynamic fluctuations in the useful runahead distance for different instances of a load instruction. Consequently, R2DP would predict more effective runahead distances to avoid performance degradation compared to RaT as we demonstrate with these experiments and the performance results in the next section.

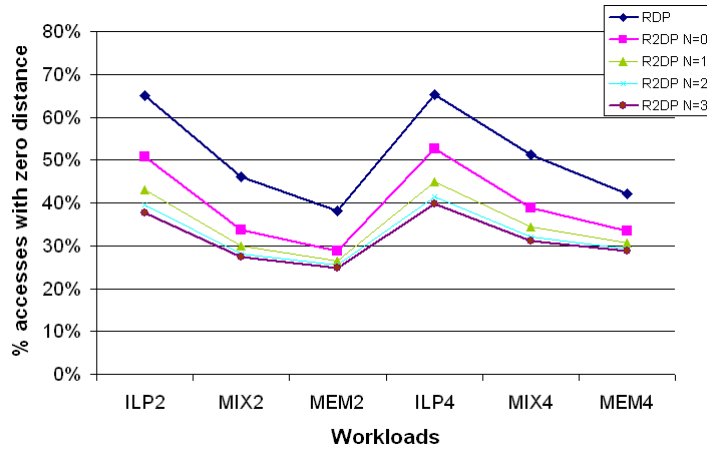


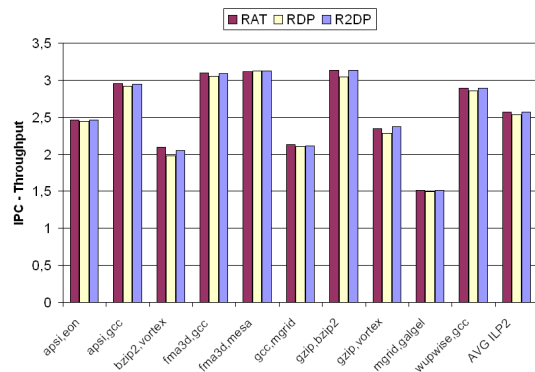
Figure 5.8: Percentage of runahead threads not started due to zero distance value

5.2.2 Performance and extra work evaluation

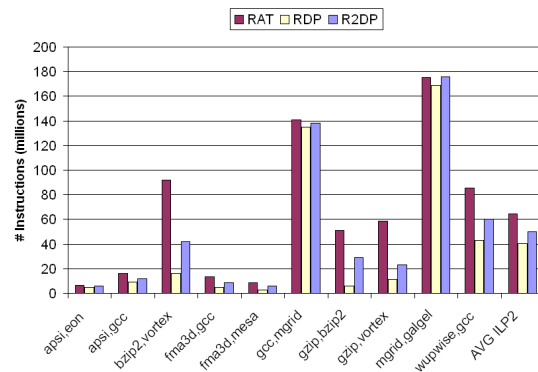
Figures 5.9 and 5.10 show the performance and extra executed instructions per each category and workload individually for 2-thread and 4-thread workloads. We show the results for RaT, the first proposal with a single distance (RDP), and the second approach of Runahead Two distance predictor (R2DP) with the best configurations obtained in the previous section. All graphics on the left column (5.9(a), 5.9(c), 5.9(e), 5.10(a), 5.10(c), 5.10(e)) show the performance in function of the IPC throughput. Graphics on the right column (5.9(b), 5.9(d), 5.9(f), 5.10(b), 5.10(d), 5.10(f)) show the total speculative instruction executed by runahead threads when each one of the different mechanism is used¹.

All performance figures show that R2DP gets similar throughput results to RaT whereas RDP suffers from performance loss compared to RaT (9% slowdown on average). This performance loss for RDP technique is more remarkable in MIX and MEM workloads as the corresponding figures show. For instance, RDP results in 7.9% performance loss for MIX4 and 12.6% for MEM4 compared to RaT. On the other hand, R2DP provides better performance than RDP for all workloads, since R2DP is more conservative from performance point of view because of the procedures to check the distance accuracy. R2DP outperforms RDP by 1.5%, 15.9% and 17.2% for ILP, MIX

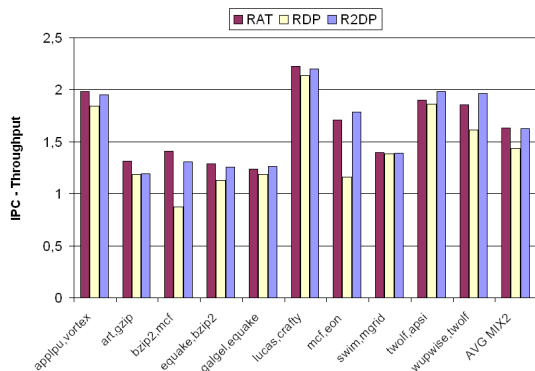
¹ Note that for clearness purposes, y-axis scale is different among figures.



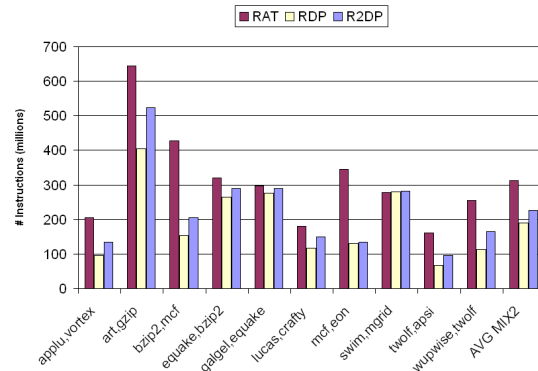
(a) Throughput ILP2



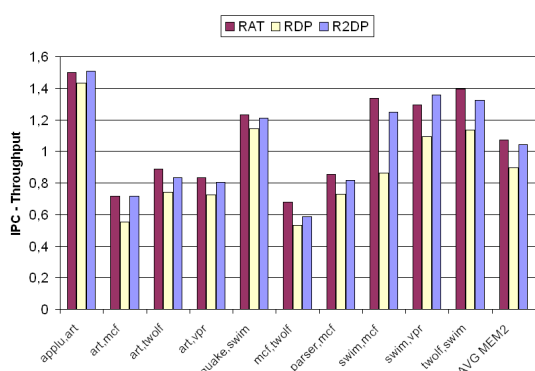
(b) Extra instructions ILP2



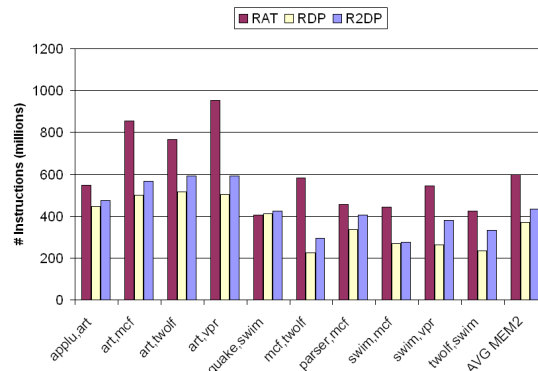
(c) Throughput MIX2



(d) Extra instructions MIX2

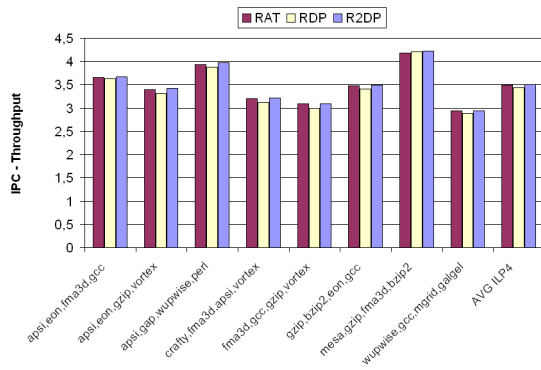


(e) Throughput MEM2

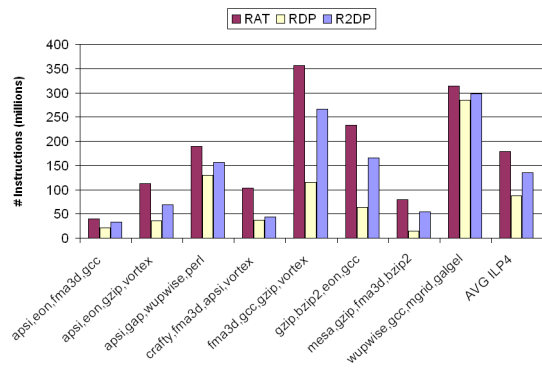


(f) Extra instructions MEM2

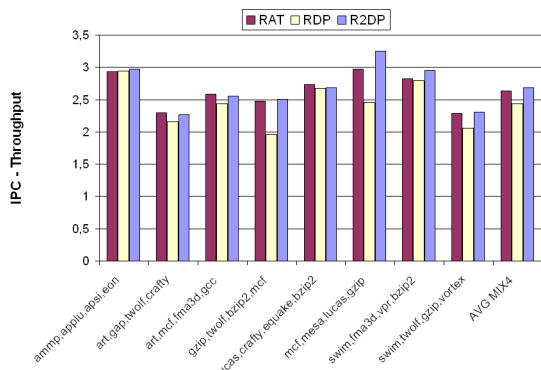
Figure 5.9: Performance throughput and speculative executed instructions for 2-thread workloads



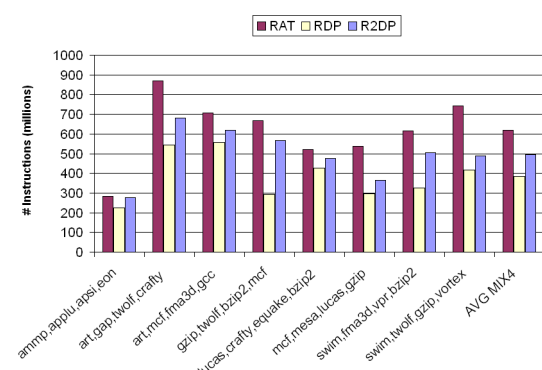
(a) Throughput ILP4



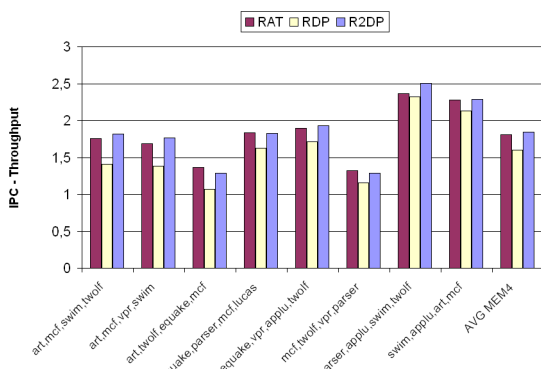
(b) Extra instructions ILP4



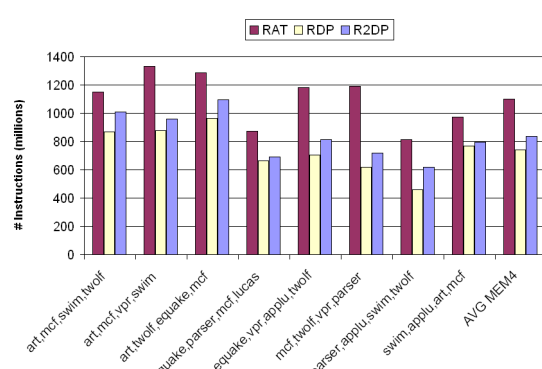
(c) Throughput MIX4



(d) Extra instructions MIX4



(e) Throughput MEM4



(f) Extra instructions MEM4

Figure 5.10: Performance throughput and speculative executed instructions for 4-thread workloads

and MEM 2-thread workloads and by 2.1%, 11.3% and 16.0% respectively in the case of 4-thread workloads.

However, using RDP technique, the runahead threads generate the least increment in the speculative executed instructions among the three mechanism evaluated. With RDP, the extra instructions are reduced for all 2 and 4 thread workloads, getting the greatest reduction in the case of MEM workloads. In the memory-bound workloads, RDP eliminates around 227 millions of speculative runahead instructions for MEM2 workloads and 359 millions of instructions for MEM4 workloads. This high reduction is due to RDP eliminates many speculative runahead instructions by always predicting the last useful runahead distance as the useful runahead distance for each runahead thread. Nevertheless, as this predicted runahead distance can be decremented and it cannot increased again due to previously explained reasons, RDP also entails performance loss due to the reduction of overlapping prefetches per runahead thread.

Although R2DP executes more extra instructions than RDP according to results showed in the different figures from the left column (b,d,f), R2DP also significantly reduces the extra executed instructions of runahead threads for the different workloads. Following with the MEM workloads, R2DP eliminates 165 and 221 millions of instructions on average for MEM2 and MEM4 workloads respectively compared to RaT. Opposite to RDP, this extra work reduction comes with a negligible impact on throughput performance as it is showed on the performance figures.

To summarize the different extra speculative work, Figure 5.11 shows the ratio of speculative executed instructions of RDP and R2DP normalized to RaT instruction count according to the previous figure results. Controlling the runahead threads by the runahead distance prediction mechanisms effectively reduces the increase in the number of speculative instructions from 33% (MEM4) to 56% (ILP4) with the RDP technique and to 28% on average with the R2DP technique. We want to note that the reduction for ILP workloads results the biggest ratio as this figure shows although the greatest reduction regarding the amount of instructions is for MEM workloads (see Figures 5.9(f) and 5.10(f)).

5.2.3 Energy efficiency

We quantify the power consumption and performance-energy balance to evaluate the efficiency of the proposed techniques in relation to RaT. We include in our extended

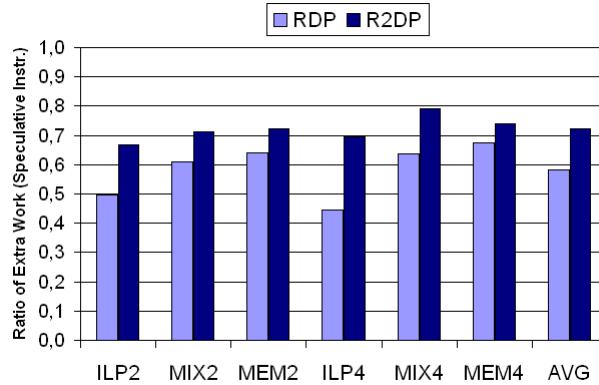


Figure 5.11: Ratio of speculative executed instructions per runahead thread (normalized to RaT mechanism)

version of Watch the power consumption and activity counts for the new hardware used, such as the RDIT, to take the energy consumption statistics.

To this end, we use CACTI to get an estimation of the power consumption. For instance, the basic power for the RDIT is 1.19 watts for R2DP configuration, which is very small regarding the total processor power consumption (less than 1%) and it is much less than cache power consumption (4,7W for Icache and 9,4W for Dcache according to our results).

Figure 5.12 shows the average power consumption of the SMT processor per each workload category in function of the mechanism applied. We show results for ICOUNT, RaT and the two techniques proposed in this chapter (RDP and R2DP). This figure quantifies the actual impact of the extra work in terms of energy and allows us measure the power savings of the different techniques. Overall, both RDP and R2DP reduce the processor power consumption compared to RaT for all workload categories. As showed in the previous section with the extra work, the biggest impact on power is also produced across MIX and MEM workloads. RDP reduces the power consumption by 17% compared to RaT, although this reduction comes with a negative performance degradation of 9% as we saw before. Likewise, R2DP reduces the power consumption for all workloads as well, with 12% reduction on average compared to RaT (up to 22% for MEM2 workloads) but in this case without performance loss.

Giving all evaluated results up to now, we have seen that RDP scheme provides more power savings than R2DP but the former degrades the performance regarding

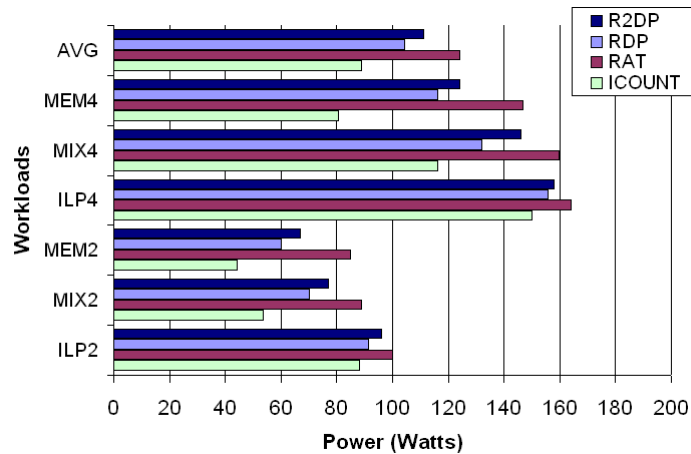


Figure 5.12: Average power consumption for the different mechanisms

the latter. To see which technique is the most efficient, we evaluate the energy-delay² (ED^2), a metric that provides an idea of how efficiently the instructions are executed, relating the performance to the power consumption.

Figure 5.13 shows the ED^2 ratio of RDP and R2DP compared to energy-delay² result of RaT for each workload category. The general observation looking Figure 5.13 is that R2DP actually is the most efficient technique in terms of ED^2 ratio since it has the lower bar for each kind of workload for both 2-threads and 4-threads. On average, R2DP provides 8.7% better ED^2 than RaT for 2-thread workloads. The result is higher for 4 threads, since R2DP provides the best average improving ED^2 by 11.8% over RaT. On the other hand, RDP energy efficiency is slightly better than RaT only in the case of ILP workloads. For the rest of workloads, both MIX and MEM, the ED^2 ratio is worse, resulting in 19% and 10.3% ED^2 ratio degradation for 2-thread and 4-thread workloads respectively compared to RaT. So, the ratio between performance and the power consumption obtained by RDP is not as good as the ratio obtained with the RaT mechanism.

Since the energy-delay² product gives more importance to delay than to energy reduction, the processor performance using the different schemes is very important when considering its energy efficiency. In this sense, R2DP that performs close to RaT is more efficient since its performance difference is minimal while reducing the wasted energy of the original mechanism. Therefore, R2DP enhances the efficiency of RaT

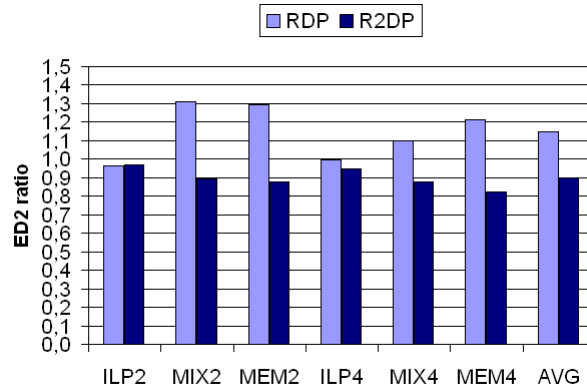


Figure 5.13: Ratio of energy-delay² product compared to RaT

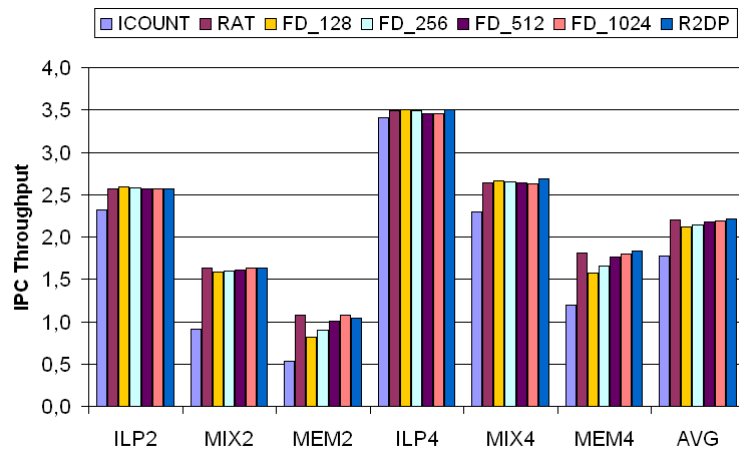
while is effective at retaining the performance benefits of runahead threads, making the executed runahead threads more efficient.

5.2.4 Evaluation of dynamic adaptivity of runahead distance

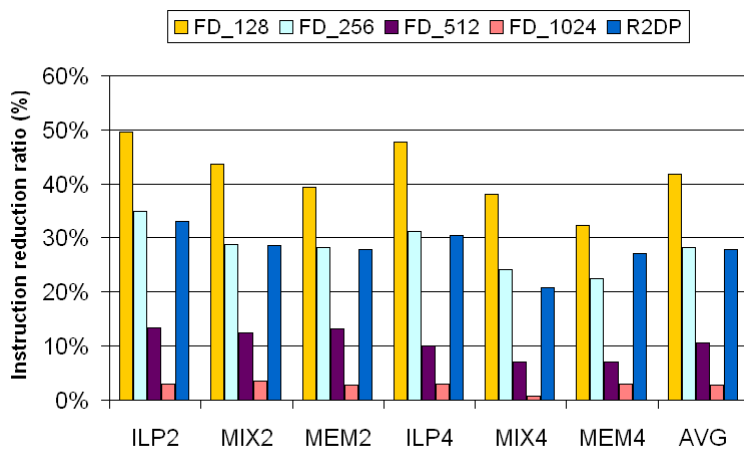
To complete the evaluation results, we evaluate the effect of dynamically adapting the runahead execution according to the useful runahead distance. We compare the R2DP technique, which is able to adapt the runahead distance of runahead threads dynamically based on run-time information, to a simple static policy that always uses a fixed runahead distance of N instructions per each runahead thread execution. We label this mechanism as the fixed distance (FD) for these experiments. So, when a load misses in the L2 cache and turn the thread into a runahead thread, FD _{N} executes N instructions in that runahead thread.

Figure 5.14 compares the performance and extra instructions with R2DP and FD _{N} versions. We experiment with values for N of 128, 256, 512 and 1024 instructions. As this figure shows, R2DP has higher performance than the fixed distance mechanism for all static distances tested on average. FD technique with small distances provides small performance gains for ILP workloads whereas large distances provide better performance in the case of MEM workloads. As the fixed distance increases, both the performance and the extra instruction executed increase (less instruction reduction), as expected.

We remark two major observations regarding Figure 5.14. First, R2PD performs better than the fixed distance mechanism FD₁₀₂₄, which results in significantly higher



(a) Throughput



(b) Percentage of speculative executed instruction reduction compared to RaT

Figure 5.14: Dynamic versus fixed runahead distance mechanisms

number of extra instructions than R2DP (42.3% more instructions). In fact, FD_1024 executes almost as many extra instructions as the non-controlled RaT mechanism (only 2.7% less instructions). Second, R2DP executes approximately the same number of extra instructions as the fixed distance mechanism FD_256 (around 28% instruction reduction for both), which performs worse than R2DP (7% less performance on average than R2DP, but up to 22% for MEM2 workloads). With these results, we show that R2DP technique tunes dynamically the different runahead distances in order to efficiently exploit the MLP for every runahead thread obtaining the best performance and

extra work reduction. Therefore, the dynamic prediction of runahead distance provides a better tradeoff in performance and executed instructions for runahead threads than using a statically set, fixed runahead distance.

5.3 Analysis of the approaches

In this section we show several data to explain the features and particular behavior of the RDP and R2DP to support previous section results. An analysis of the distance prediction accuracy and characteristics about the controlled runahead threads when using these proposals are provided. This provides insights into how each technique manages the executed runahead instructions according with their different approach.

5.3.1 Runahead distance prediction accuracy

We evaluate the prediction accuracy of the useful runahead distance predictors by RDP and R2DP. To evaluate whether the particular predictor can accurately predict the runahead distance, we quantify the probability to predict the ideal useful runahead distance to exploit the maximum MLP. That is, a runahead distance far enough that captures the maximum long-latency loads with the minimum runahead instructions per each runahead thread. This *ideal* runahead distance represents the optimum distance, according to our useful distance definition, calculated for each executed runahead thread using the original RaT mechanism. Therefore, we consider a useful runahead distance prediction as a misprediction if the predicted distance applying a particular technique (RDP or R2DP) is smaller than the ideal useful distance at the end of a runahead thread (i.e., the maximum available MLP is not fully exposed by the distance predictor). Otherwise, a prediction is classified as a correct prediction if the predicted distance is at least as large as the ideal useful runahead distance.

Figure 5.15 shows the average accuracy of the Runahead distance techniques for the different categories of workloads. This data shows the ability of RDP and R2DP for predicting whether a long-latency load is going to expose enough MLP through a given runahead thread in function of the useful runahead distance. The average runahead distance prediction accuracy for RDP is 67% whereas for R2DP is 83.4%. For the R2DP, the accuracy is higher due to the different refinements that improves the runahead distance learning reducing the RDP technique mispredictions. In case

of RDP, this lower accuracy results in smaller runahead threads but also it leads to performance loss, since runahead threads will be ended or not started although there is MLP to be exploited. This also causes execute more runahead threads as we will show later.

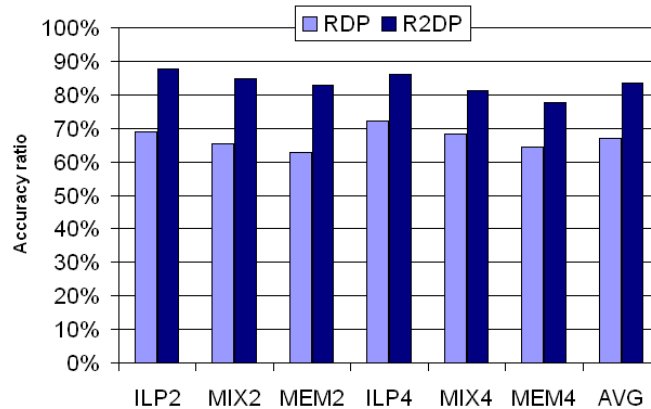


Figure 5.15: Accuracy of RDP techniques for predicting the ideal runahead distance

5.3.2 Predicted useful runahead distance

Figure 5.16 shows the average useful runahead distance predicted by RDP and R2DP for the different workloads. For comparison purposes, we also show the average optimum runahead distance in function of the ideal runahead distance of RaT as we described in the previous section. As we can observe, the R2DP average useful distances are larger than RDP ones in each category of workload, that is, 205 versus 123 on average, thereby closer to optimum distance. RDP predicts lower runahead distances than R2DP since the useful distance computation for each long-latency load monotonically decreases because of RDP functionality. RDP always predicts the last runahead distance without employing any confidence mechanism.

The better prediction accuracy of R2DP results in close useful runahead distance to optimum distances as Figure 5.16 shows. R2DP is more accurate than RDP predicting the useful runahead distance because it employs confidence mechanisms to restore the runahead distance when it is not reliable. Hence, R2DP increases the predicted useful runahead distance per runahead thread with regard to RDP, in order to get better

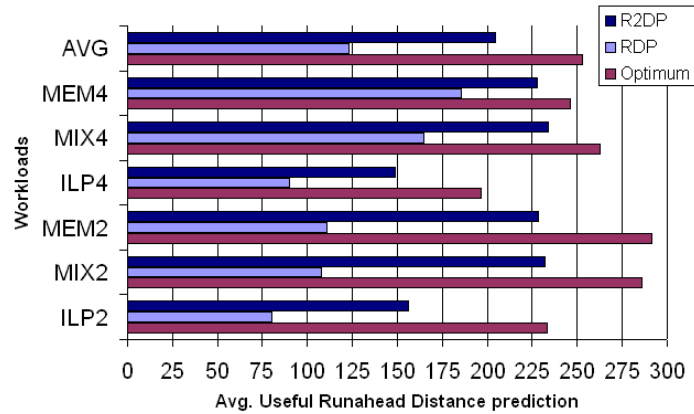


Figure 5.16: Predicted averages useful distances for RDP and R2DP mechanisms

performance. The predicted runahead distances are usually larger for MIX and MEM workloads (more than 225 instructions) having more impact for these workloads as we later see.

5.3.3 Length of runahead threads

We have shown the prediction accuracy and predicted distances of the two approaches, so we also show the runahead executed instructions in the controlled runahead threads to complete part of this analysis. Figure 5.17 shows the average length of runahead threads for the original RaT mechanism and the two approaches of this chapter. This length is measured as total number of instructions executed per runahead thread on average. So, this data is relative to the total executed runahead instructions and the number of runahead thread activations (for instance, ILP workloads execute fewer runahead threads than MIX or MEM workloads). For comparison purposes, we also show a striped part for RaT bars which indicates the average ideal useful distance for the executed runahead threads.

From this figure, runahead threads controlled by RDP and R2DP have smaller lengths than RaT. The average length reduction per runahead thread for 2-thread workload is 462 instructions for RDP and 282 instructions for R2DP, whereas for 4-thread workloads this reduction is 341 and 201 instructions respectively. Therefore, RDP causes runahead threads to execute the lowest number of speculative instructions, resulting in runahead threads with 202 executed instructions on average opposite to

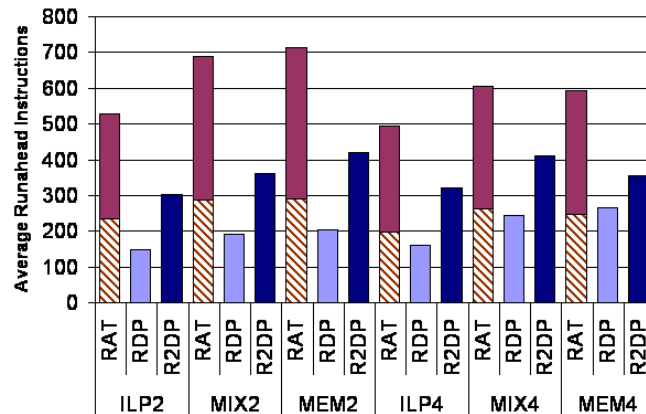


Figure 5.17: Average number of instructions executed per runahead thread in function of used mechanism (RaT, RDP and R2DP)

604 of RaT. R2DP, with an average of 362 instructions, still reduces by 37% the average runahead instructions per thread related to RaT.

Likewise, looking at the ideal computed useful runahead distances (striped part) provides information about how well each technique manages the executed runahead threads to get the optimum performance and control the useless instruction execution. As the figure shows, RDP bars are under this optimum distance of RaT for each category of workload while the R2DP bars are over. So, RDP effectively reduces the speculative runahead instructions but RDP mispredictions produce that the predicted distances are below of the appropriate useful runahead distances for getting similar performance to the original RaT. It eliminates a large number of runahead threads and therefore it eliminates the prefetching benefits provided by many useful runahead threads.

Using R2DP, the length of runahead threads is over to the length of ideal useful runahead distance, although close to this. Therefore, these data show that R2DP results in smaller runahead threads than RaT alone, but they are more efficient ones, that is, runahead threads with fewer executed instructions than RaT with similar performance improvement.

5.3.4 Number of runahead threads

Figure 5.18 shows the number of runahead threads executed per each technique (RAT, RDP and R2DP) on average for each kind of workload. R2DP increases 11% the amount of runahead threads for all workloads compared to RaT on average, specially for MIX and MEM workloads, with 13% and 12% more runahead thread executions respectively. In the case of ILP, the increase is smaller (9%) since these workloads have a low ratio of long latency misses. For RDP technique, the ratio is greater than R2DP with 41% more runahead threads than RaT. However, although there are more runahead thread executions for these techniques, the length of runahead threads controlled by the runahead distance prediction techniques is lower than baseline RaT as we show in the previous section.

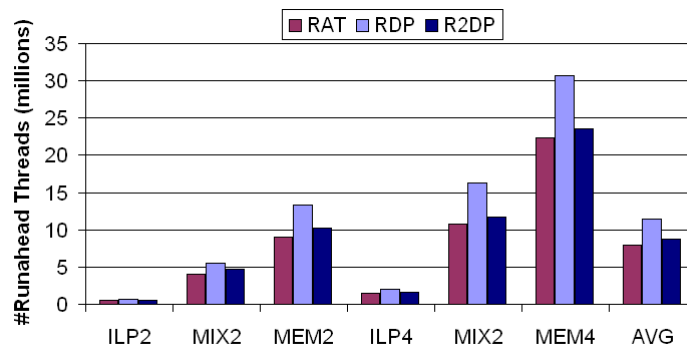


Figure 5.18: Number of runahead threads executed per each mechanism

5.3.5 Distribution of controlled runahead threads

We finally show in this last section several experiments that give insightful details about how the most-efficient R2DP approach works. Figure 5.19 illustrates a particular analysis of the runahead thread distribution in terms of the different R2DP decisions during 2-thread workload executions. In this figure, each bar is broken up in three parts (from top to bottom): the percentage of runahead threads fully executed, the runahead threads limited by a predicted useful runahead distance and, finally, the percentage of possible runahead threads that were not started (due to their predicted useful distances do not fulfill the activation threshold).

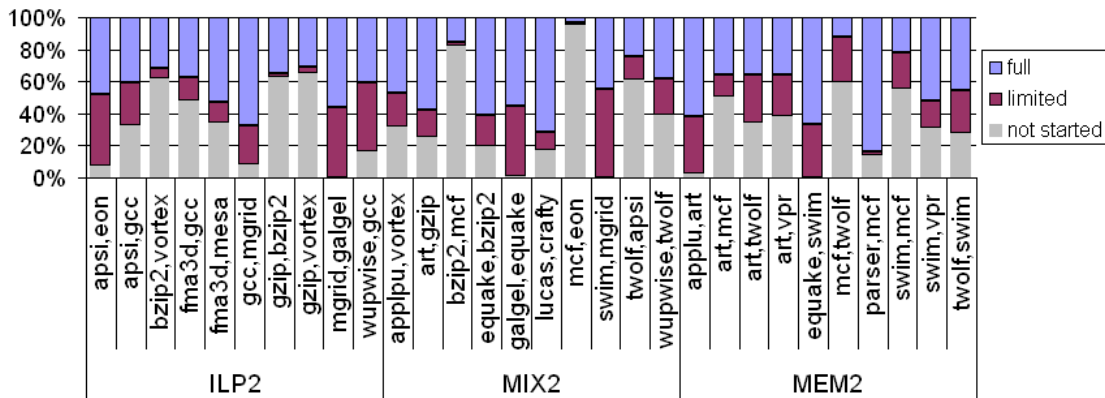


Figure 5.19: Runahead thread executions breakdown

This figure indicates how R2DP manages the runahead threads in the different workloads and how many times a runahead thread is controlled or not according to the predicted useful distances. For example, if we analyze how many times our technique eliminates a runahead thread completely for a particular workload, we can observe there are workloads which present a high percentage. These are the cases of *bzip2-mcf* and *mcf-eon*. As *mcf* is a benchmark with a huge number of dependent loads. This feature causes invalid runahead loads that do not issue prefetches which are not taken into account for useful distance computation, thereby reducing the useful distance value for the corresponding runahead threads. On average, the percentage of not initiated runahead threads due to small distances for 2-thread workloads is 34% (in the case of four threads this percentage is 37%). This ratio represents the useless runahead threads detected and avoided.

On the other hand, there are workloads which have a high ratio of runahead threads limited by the corresponding useful runahead distance. Such examples are *apsi-eon*, *mgrid-galgel*, *galgel-equake*, *swim-mgrid* or *applu-art* with around 40% of runahead threads which R2DP controls their execution according to the useful runahead distances. This percentage is 22.5% for overall 2-thread workloads. This ratio addresses the elimination of the useless part of executed runahead threads and then, it contributes to reduce the speculatively executed useless instructions and their energy consumption as we have already showed in the previous section.

However, there are many loads for some kind of programs (specially computing intensive threads) that have isolated long-latency misses. In addition, these loads

are not re-offender, generating a runahead thread just once. Therefore, the runahead thread is always fully executed the first and single time (for instance *gcc,mgrid* or *lucas,crafty*). For these cases, the useful distance information is rarely used in the future for the same load, so it is difficult to control and avoid more useless runahead executions.

To complete the previous study, Figure 5.20 shows a histogram of how many runahead threads execute between N and M instructions (for ranges of 32 instructions) for particular *art,gzip* workload using RaT and R2DP technique². The bar for R2DP between 0-32 instructions indicates the number of runahead threads not started for this workload because the runahead distance is not higher than 32.

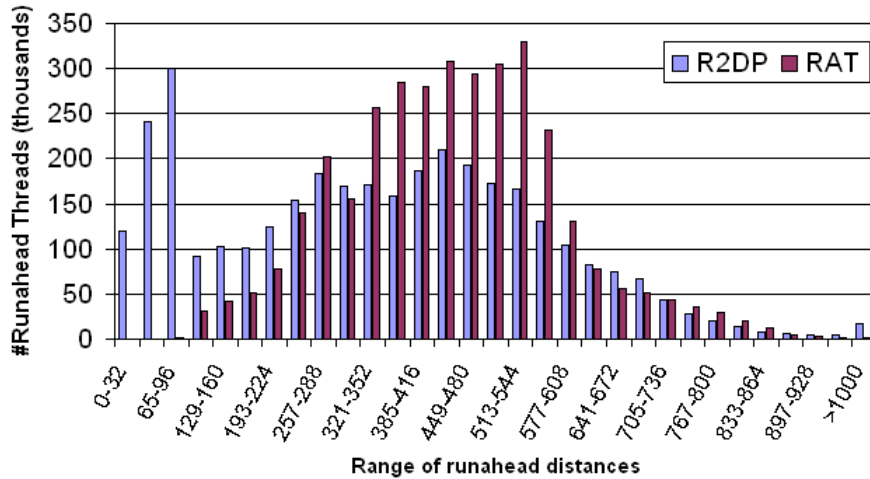


Figure 5.20: Runahead threads distance histogram for R2DP and RaT

As we can observe, R2DP reduces the number of runahead threads with larger distances compared to RaT. This reduction is concentrated in the central distribution of this figure, in which the higher number of runahead threads are. This distribution is shifted towards runahead threads with smaller runahead distances. Thus, the Figure shows as R2DP effectively eliminates the useless part of larger inefficient runahead threads causing more efficient runahead threads in the range of smaller distances.

²Although we only show one example, the rest of workloads follow a similar trend.

5.4 Summary

In this chapter, we have developed and evaluated two fine-grain approaches to make runahead threads more efficient. To improve that energy efficiency, these approaches focus on predicting the maximum MLP achievable by a particular runahead thread while at the same time reducing the extra useless speculative work.

The two mechanisms described are similar designs that aim to improve the efficiency of runahead threads with low hardware cost and complexity. Both schemes are based on the *useful runahead distance*, a concept that indicates how far a thread should run ahead such the speculative runahead execution is efficient. One approach is called Runahead Distance Prediction (RDP) and the other Runahead Two Distance Predictor (R2DP). Aside from the own operative process of each approach, the general scheme has to main actions. First, it predicts *whether or not* a thread should employ runahead execution, (i.e. whether or not runahead execution is useful) to avoid the execution of useless runahead threads. Second, it predicts *how long* the thread should execute in runahead mode to reduce the unnecessary speculative instructions execute at the end of useful runahead threads. Among both approaches, R2DP distance prediction scheme is a better attempt at capturing the useful runahead distance in a more accurate way.

Limiting the runahead execution of a thread by this distance prediction not only avoids unnecessary speculative execution but also reduces the executed instructions, thereby the resource requirements of runahead threads. We have evaluated both RDP and R2DP in terms of performance improvement, reduction of extra instructions, and energy efficiency. Although both approaches effectively reduce the speculative extra instructions, RDP by 42% and R2DP by 28%, results have shown that R2DP is more energy-efficient than RaT reducing the power consumption by 12% on average without affecting its performance. Therefore, R2DP provides not only high performance but also an efficiency aware way of managing runahead threads in SMT processors.

Related Work

Both Simultaneous Multithreading and Runahead execution are well-known micro-architectural models focused on overcoming the superscalar processor limitations. The former exploits the thread-level parallelism to improve the performance throughput whereas the later provides an alternative to building large instruction windows to tolerate long-latency operations. Nevertheless, they are two different and clearly separate proposals that have not been considered together before to this work.

In this chapter we describe the related work that involve SMT and Runahead research lines. We cover simultaneous multithreading mechanisms, thread-base speculative techniques and the Runahead background close to the scope of this thesis. We describe the functionality and benefits of the most relevant approaches and discuss the differences, advantages or disadvantages in relation to Runahead Threads and the techniques proposed in this dissertation.

6.1 Simultaneous Multithreading

Simultaneous Multithreading [53][76][84] emerges as a solution to superscalar processor limitations to increase the performance through exploiting thread level parallelism and tolerate long main memory latencies better. A simultaneous multithreaded processor has the ability of a single physical processor to simultaneously dispatch instructions from more than one hardware thread context. The processor maintains a list of active threads and decides which instructions from those threads to issue into the pipeline.

Depending on the level of sharing, threads use exclusively some machine resources, like the reorder buffer, or they share others resources like the issue queues, the functional units, and the physical registers. Shared resources are dynamically allocated between threads competing for them. This dynamic resource distribution among threads determines not only the final processor performance, but also the performance of individual threads. If a single thread monopolizes most of the resources, it will run almost at its full speed, but the other threads will suffer from a slowdown because of resource unavailability. Therefore, the design of an SMT processor requires additional resource policies that determine how the resources should be shared.

Although software approaches exist that try to reduce the interference in SMT shared resources, like compiler techniques [42][54] or operating system techniques [57], in this thesis we mainly focus on hardware techniques. SMT processors topic is an intensive research line, and different and new SMT techniques and policies have been proposed as this thesis was being developed. The different researches in literature in this line include instruction fetch policies, dynamic resource allocation mechanisms and memory-level parallelism aware techniques. We describe the different hardware proposals for each category in the next sections. We will provide a detailed quantitative comparison of the mechanisms from this thesis with the main resource control policies described in this section in Chapter 7.

6.1.1 Instruction fetch policies

The use of shared as well as partitioned resources in an SMT processor can be indirectly controlled by instruction fetching mechanisms. An instruction fetch (I-fetch) policy determines which threads feed the processor pipeline with new instructions and which ones are left out. This decision indirectly affects how shared resources are allocated in function of front-end instruction advance. Various fetching policies have been proposed in the literature to provide the best supply of instruction mixes from multiple threads for building the most efficient execution schedules.

Initial fetch policies, which were presented by Tullsen *et al.* [75], are based on several simple heuristics for the thread selection process that assign different priority to each thread. These techniques focus on keeping the maximum fetch bandwidth (maximize the performance throughput) by fetching from multiple threads each cycle. From this work, Round-Robin (RR) policy fetches instructions from all threads alternatively,

disregarding the resource use of each thread. Others I-fetch policies in the same work [75] attempt to improve the round-robin priority policy using feedback from other parts of the processor. Among these other policies proposed, the ICOUNT policy presents the best performance results, specially for threads with high instruction level parallelism (ILP). ICOUNT prioritizes threads with fewer instructions in the pre-issue stages of the pipeline (decode, rename and the issue queues). This policy tries to achieve three purposes: (1) to prevent a single thread from clogging the issue queues, (2) to give highest priority to threads that are moving instructions through the front-end most efficiently, and (3) to maximize the parallelism in the queues from a mix of instructions, thereby improving processor throughput.

However, previous policies have difficulties to deal with threads that present many long-latency memory operations (e.g. a load that misses in the L2 cache). When this situation happens, the RR or ICOUNT policies do not realize that a thread can be blocked on an L2 cache miss and will not make forward progress for many cycles. These threads with a high L2 miss rate allocate an excessive share of pipeline resources stalling the other threads (for instance, the processor can run out of registers). In consequence, the total performance throughput suffers from a serious slowdown. Therefore, several techniques built on top of ICOUNT are proposed to alleviate this problem and prevent resource monopolization.

STALL [74] prevents the thread from fetching further instructions if an L2 cache miss is predicted. When this happens, *STALL* technique stops the offending thread temporarily. This stalled thread neither enter new instructions in the pipeline nor compete for resources. Nevertheless, the main drawback of the stall policy is that the L2 miss detection mechanism may be too late and a thread may already hold many resources until the long latency load is solved. For this reason, authors extend *STALL* policy and also propose the *FLUSH* mechanism. *FLUSH* [74] minimizes this problem by flushing instructions from the offending thread as long as of the long latency operation is detected. In this case, *FLUSH* makes the shared allocated resources available for other threads. Cazorla *et al.* proposed *FLUSH++* [6], which combines the advantages of *STALL* and *FLUSH*. *Flush++* is based on the fact that *STALL* performs better than *FLUSH* for workloads that do not require many resources (few long-latency loads) and that *FLUSH* performs better than *STALL* for workloads that require many resources (many long-latency loads). This policy analyzes the cache behavior and number of threads to select between *FLUSH* or *STALL* trying to get the best benefit of each

technique. In addition, FLUSH++ use an additional thread running policy, named *continue oldest thread* -COT. This enhancement always attempts to leave the oldest stalled thread running. That is, there are N hardware contexts in execution, but N-1 of them are already stalled because of STALL or FLUSH actions. If the only thread running experiences an L2 miss, it is flushed and stalled, but then, the thread that was first stalled is activated and continues its execution.

Other mechanisms focus their attention on L1 Dcache misses instead of the L2 cache ones. The key idea is to prevent the negative effects before they occur, instead of waiting until an L2 miss is produced, when probably some harm has already been done. In this way, two mechanisms are proposed to reduce clogs in the issue queues caused by loads that miss in the L1 data cache [24]. The first mechanism, Data Gating (DG) stalls a thread to avoid fetching from threads that experience an L1 data miss. The second one, Predictive Data Gating (PDG), is more aggressive than DG and adds an L1 data cache miss predictor. PDG prevents a thread from fetching instructions as soon as a cache miss is predicted. The main problem of the DG and PDG policies arises when there are few threads, because the exposed parallelism and the pressure on resources are low. Consequently, to stall a thread every time it has an L1 data miss may cause an under-utilization of the shared resources. Dwarn policy [8] also uses L1 data cache misses as indicators of a possible L2 miss. But, Dwarn does not stall the thread due to L1 misses as previous techniques DG and PDG do. In this technique, threads experiencing an L1 data cache miss are given lower fetch priority than threads with no data cache misses. That is, Dwarn uses indirect indicators (data L1 cache misses) of potential resource abuse by a given thread in order to adapt the pressure on resources reducing resource under-use.

Nevertheless, these techniques would punish the offending thread excessively at the cost of increasing later its re-start latency or even the released resources may not be used by the other threads.

6.1.2 Dynamic resource allocation techniques

While all the I-fetch mechanisms are effective to various degrees, they never control explicitly the per-thread resource utilization. They only make decisions based on static criteria or events, like L2 misses, to give priorities, stall or flush the threads. I-fetch techniques do not exploit dynamic architectural information about the resource alloca-

tion and they do not exercise direct control over how resources are distributed among threads¹. Therefore, even if I-fetch techniques try to prevent resource monopolization, they can sometimes cause resource under-utilization if other threads do not require the deallocated resources. To go one step forward in SMT resources policies, the research community have also proposed dynamic resource allocation and control policies. This kind of techniques carry out a more fine-grained dynamic control over SMT resources. Information about resource utilization or performance impact is considered. This direct control allows a better use of resources, reducing under-utilization, but requiring additional hardware components (like additional counters) and the logic to implement the corresponding resource sharing model.

DCRA [7] technique presents a dynamic resource sharing algorithm. This technique directly monitors the usage of resources by each thread during the course of their executions, trying to guarantee that all threads get a fair amount of the critical shared resources in function of the different resource requirements. To do this, DCRA first classifies threads according to the amount of resources they require. This classification provides a view of the demand that threads have of each resource. Next, based on the previous classification, DCRA determines how each resource should be distributed among threads assigning a resource usage limit per thread type. Each cycle, DCRA directly monitors the resources usage per thread and dynamically changes the resource limits. In contrast to the previous methods that directly stall or flush threads which have cache misses, DCRA firstly attempts to help these threads by providing more resources to them (if such resources are available). However, when DCRA detects that a thread is exceeding its assigned allocation limit, it immediately stalls that thread until it no longer exceeds its allocation assignment.

Choi *et al.* propose *Hill Climbing* (HC) [11] as other dynamic approach for SMT resource distribution that uses performance feedback to address the resource sharing. Instead of monitoring the resource indicators each cycle, Hill Climbing observes the impact that resource distribution decisions have on performance at runtime, and feeds this information back to the resource control mechanism to improve future decisions. This approach employs a learning-based algorithm (hill-climbing) that varies the resource allocation of multiple threads towards the best distribution of resources. This learning-based algorithm starts from equal partitioning, and then, it moves an equal amount of resources from all the other threads to a preferred thread by evaluating

¹Only, the ICOUNT policy takes into account the occupancy of the issue queues.

each certain time (epoch) the resource distribution. This approach tries to adapt the best resource partitioning following the gradient descent functions derived from SMT performance metrics. Like DCRA, when a thread exceeds its assigned resource sharing pool, it is stalled until that utilization decreases. Nevertheless, some implementations of this technique require off-line single-thread executions to get the IPC of individual programs. This is computationally expensive because of the exhaustive trials depending on the real system workloads. In addition, it requires some kind of support for external inputs to provide that information.

6.1.3 MLP-aware policies

The Memory-Level Parallelism (MLP)[31] indicates the number of memory operations that are outstanding in parallel. Previous research has shown that microarchitecture features and parameters have a profound impact on achievable MLP and that exploiting MLP is an effective approach for improving the performance of memory bounded applications [12][37][51]. Programs with high MLP tend to have bursts (or clusters) of long-latency loads in the dynamic instruction stream in several phases of program execution [12]. Therefore, this feature makes the exploitation of MLP feasible, for instance using phase-based prediction techniques.

Specifically for SMT, a recently proposed set of policies [27] take into account the available memory-level parallelism (MLP) presents in the executed programs to control the threads. These MLP-aware techniques are the most related to our work, since they are also aware of the memory-level parallelism for improving previous SMT fetch policies. These MLP fetch policies continue fetching instructions up to the point where the maximum available MLP for the given ROB size can be achieved. The key idea is a thread that overlaps multiple independent long-latency loads (e.g. a memory-intensive thread) has a high-level of MLP, and therefore this thread should be allowed to allocate as many resources as needed in order to fully expose this available MLP.

For this purpose, the amount of MLP for a given load is predicted. Based on the amount of MLP predicted, a fetch control action is performed over the thread which executes that load. Depending on the particular technique, they decide to (1) fetch stall (MLP-STALL) or flush (MLP-FLUSH) the thread in case there is no MLP (like previous stall or flush instruction fetch policies) or (2) continue allocating resources

for the particular thread for as many instructions as predicted by the MLP predictor. After that, they also stall or flush the thread according to the corresponding technique.

However, these techniques has two sensitive factors that can limit their potential. Firstly, they rely on the MLP predictor accuracy to speculatively execute instructions. Secondly, the number of speculative instructions is limited by the reorder buffer and the hardware setup of the MLP predictor (e.g. the long-latency shift register size). These two factors can reduce the opportunities to improve the performance because it limits the amount of MLP achievable since other distant long-latency loads cannot be exploited.

Most of the presented fetch policies and resource control policies in the previous sections have a determinant common action: they stall or flush threads under determined conditions to prevent resource overuse. In this sense, except MLP-aware techniques, the rest of them generally restrict memory-intensive threads in order to get higher global throughput from fast threads. Although these policies avoid memory-intensive thread monopolization, the performance of fast threads are better off with the cost of stalling the execution of memory-intensive threads. Consequently, these control actions, either stalling or flushing threads, harm performance opportunities of memory-bound threads to unfairly benefit fast threads. On the contrary, our solution is try to reach the potential performance achievable on SMT processors making the most of shared resources without harming none kind of thread.

6.2 Runahead paradigm

Runahead approach (RA) is a speculative paradigm whose goal is to bring ahead data and instructions into the caches. A first proposal of Runahead was presented and evaluated as a method to improve the data cache performance of a pipelined in-order execution machine [23]. This first mechanism pre-executes instructions under a cache miss on an in-order processor that does not employ any hardware prefetching techniques. It shows to be effective at tolerating the latency of first-level data and instruction cache misses for this type of processors.

Later, Runahead was extended for out-of-order superscalar processor [51] as an alternative to large instruction window processors [1][18][65]. In this scenario, Runahead consists of avoiding the blockage of the instruction window due to long-latency operations, *e.g.* a load that misses in the second level (L2) cache. Instead, the processor

continues executing instructions speculatively, trying to follow the most likely program path until the load that triggered the runahead mode is resolved. The runahead main benefit comes from the pre-execution of these speculative instructions which improves the data and instruction cache efficiency.

Currently, we contribute to extend the paradigm of Runahead to improve the performance of the multithreaded processors, which are nowadays the base for high-performance computing designs. We call this approach *Runahead Threads*(RaT). As we have described in previous chapters, RaT is a valuable solution for both exploiting memory-level parallelism and reducing resource monopolization in SMT processors. We propose a new utilization of the Runahead on SMT processors as a different and speculative policy to improve the performance of memory-intensive threads without penalizing computing-intensive threads. Memory-intensive threads can tolerate better memory latencies and the other threads can proceed without resource clogging problems by using RaT on SMT processors.

Energy-efficiency techniques for Runahead

Previous research [50] shows that Runahead executes significantly more instructions than an out-of-order processor, sometimes without providing any performance benefit. Hence, runahead execution is not efficient for these cases. This work identifies three causes of inefficiency in runahead execution; short, overlapping, and useless runahead periods. In base of this study, several simple techniques to reduce their occurrence and improve the efficiency of runahead execution in single-threaded processors are proposed in that work.

To eliminate useless runahead periods, they propose a technique with a binary MLP predictor based on two-bit saturating counters. In case there is no MLP to be exploited, a runahead period is not initiated at all. To eliminate short and overlapped periods, the authors presented different threshold-based heuristics. To avoid many short runahead periods, this work [50] suggests keeping track of the number of cycles each L2 miss spends waiting for data from memory. Thus, the processor is only allowed to enter runahead when this count crosses a threshold (rather than when the load reaches the head of the ROB). Overlapping runahead periods are another source of inefficiency. Two runahead periods overlap if some of the instructions executed during the first period are re-executed during the second. To avoid this inefficiency, Mutlu *et al.* proposed beginning a new runahead interval only when it will not overlap a previous

runahead interval. To achieve this, the processor counts the number n of instructions pseudo-retired during runahead mode and saves that value in a register on return to normal mode. In normal mode, it also records the number of instructions i fetched since the last runahead period. When an L2 miss reaches the head of the ROB in normal mode, the processor enters runahead mode only if i is greater than n .

These single-thread *efficient runahead* techniques are different from our Runahead distance prediction because they solely try to predict and *eliminate* runahead periods that are ineffective (short, overlapping, and useless runahead periods). However, if a runahead period is predicted to be effective (i.e., not short, not overlapping, and not useless) then the processor stays in runahead mode until the L2 miss that caused entry into runahead mode is serviced. For this reason, even if runahead execution does not provide any benefits beyond some point in runahead mode (that we called useful runahead distance), these techniques continue speculatively executing instructions in runahead mode until the L2 miss that caused runahead is serviced.

Another technique [17] combines the advantages of both MLP-aware flush and RaT mechanisms. This paper proposed MLP-aware runahead threads to reduce the number of useless runahead periods. In case the MLP predictor predicts there is far-distance MLP to be exploited, the long-latency thread enters runahead execution. If not, the MLP-aware flush policy is applied to free allocated resources while exposing short-distance MLP. Whereas this proposal predicts the MLP only in order to decide whether the thread goes into full runahead execution (far-distance MLP) or the thread is flushed/stalled (short-distance MLP), the novelty and difference of our proposal is that we also predict *how long* a thread should stay in runahead by the useful runahead distance prediction. For this very reason, we will show as our mechanism is able to eliminate more useless speculative instructions (even in useful runahead periods) that cannot be eliminated by these previous techniques. In Chapter 7, we provide a detailed qualitative and quantitative comparison of our proposals to efficient runahead execution and find that our proposal provides significantly higher performance and energy efficiency for Runahead Threads.

6.3 Thread-based speculative techniques

So far, we have only discussed hardware thread scheduling techniques focus on resolving the related resource sharing problems on SMT processors. Otherwise, there are other

techniques that employ the multiple hardware contexts of these processors to speedup the execution of single-thread programs. These mechanisms take profit of the hardware support on a multithreaded processor for spawning speculative threads and obtain some performance improvements.

Runahead Threads share some aspects with these thread-based techniques present at literature. RaT catches the essence of being a speculative mechanism but it is inspired by the goal of improving the thread performance and reducing the resource monopolization at the same time on SMT processors. The idea regarding perform speculative pre-execution is close to the idea of our proposed mechanism although using a different approach with important differences as we explain below.

These speculative thread-based techniques [9][22][55][80] consist on executing a subset of the original program instructions on separate threads (helper or assisted threads) in parallel with the main computation thread with the purpose of to improve the performance of this last one (by generating prefetches or resolving branches early).

The slipstream processor [55] runs shortened advanced program (A-stream) ahead of the main program (R-stream) by dynamically skipping computation non-essential for correct forward progress. The A-stream runs faster as a result, but it is speculative. Simultaneous Subordinate Microthreading (SSMT) [9] was proposed as an attempt to improve performance of a single thread by having multiple microthreads (sequences of microcode) that do useful works such as prefetching data or training the branch predictor. Assisted Execution [22] uses lightweight threads (known as nanothreads) that share idle fetch and execution resources on a multithreaded processor. These assisted threads mainly run code ahead of the primary thread to accelerate the memory accesses triggering cache misses earlier or gather performance statistics on the main thread.

While these previous approaches directly spawn helper threads under some event of the main thread execution, a number of other proposals investigate the creation of pre-execution threads, that we call slice-based helper threads. Slice processors, proposed by Moshovos, *et al.* [49], Speculative Data-Driven Multithreading (DDMT) [56], Execution Based Prediction [85] and Dynamic Speculative Precomputation (DSP) technique [14] are examples of these approaches. These techniques set the possible points to spawn threads via offline or online analysis and dynamically generate sequences of code (also known as precomputation slices) to accelerate the main thread.

Overall, most of benefit of these speculative thread-based techniques comes from the top few targets, prefetching loads or branch prediction. The main disadvantages of these techniques are: (1) they require a potential sophistication of slice creation to construct the helper threads, (2) they employ several contexts and processor resources to enhance the performance of a single main thread, (3) most of them require a complex design and are bounded by slice size, window size, or instruction types and, (4) some of these slice-based helper thread techniques require the construction of efficient code slices and the insertion of special instructions in the code which success relies on the effectiveness of the compiler.

In contrast with these techniques, RaT neither requires to active a separate thread context nor relies on compiler support. RaT mechanism does not spawn new threads nor insert specific instructions to switch normal threads into runahead threads. Our approach uses the same thread context, and it does not require a separate context to take benefit of prefetching. During long-latency memory access periods, we switch the offending thread into a light speculative thread that uses as few resources as possible to improve its performance by prefetching and reducing the resource clogging through the same hardware context.

Overall Mechanism Comparison

During this dissertation, we present Runahead Threads as a high-performance mechanism to resolve the SMT problems and evolve the initial approach with complementary techniques to reduce its useless speculative execution improving its energy efficiency. In this chapter we want to give a quantitative overall overview of all this research work. We evaluate in detail the performance, fairness and efficiency of the different runahead thread-based mechanisms compared to a large range of state-of-the-art SMT techniques. The objective of this chapter is to present a survey that reports which mechanisms fit best depending on the scenario constrains: high-performance or power consumption, or even both. In addition, this overall evaluation represents the first complete study that includes all these techniques at the literature.

7.1 State-of-the-art SMT mechanisms

In this section, we provide a detailed comparison of RaT and state-of-the-art SMT techniques. This comparison is performed with the most relevant prior work in the scope of SMT resource policies presented in Chapter 6. All of them are also compared to the reference SMT baseline with the ICOUNT policy used during this dissertation. Regarding runahead thread-based mechanisms, we show the results for original RaT, code semantic-aware runahead threads with loop stall and post-subroutine usefulness techniques (RaT-LS+PSU) and the efficient runahead threads with the Runahead two distance prediction mechanism (RaT-R2DP).

7.1.1 Performance evaluation

According to our SMT mechanism classification in Chapter 6, we group the performance evaluation in three categories:

- instruction fetch (I-fetch) policies,
- dynamic resource allocation mechanisms,
- memory-level parallelism (MLP) aware techniques.

The experiments are performed following the evaluation methodology and performance metrics described in Chapter 2. Due to the extension of all experiments, we do not show the individual performance results for every workload.

RaT vs. I-fetch policies

An I-fetch policy determines which thread is going to fetch instructions in a given cycle. For this group of techniques, we select three most-common I-fetch schemes for handling long-latency loads: *STALL*, *FLUSH* and *FLUSH++*. Figure 7.1 summarizes the average throughput of these techniques including the baseline ICOUNT and RaT mechanisms for the different types of workloads. In general, we can observe as RaT mechanisms performs better than all I-fetch techniques for all workload categories both for 2 threads and 4 threads workloads, improving significantly the performance for MEM workloads. The average performance is very close among our different RaT approaches.

Overall, regarding the I-fetch policies, *FLUSH++* slightly outperforms *FLUSH*, and *FLUSH* outperforms *STALL*. *STALL* is less aggressive than *FLUSH*, since it does not re-execute instructions. This favors the performance in case of ILP workloads which allocate resources by less time. However, its performance results are worse especially for MEM workloads compared to *FLUSH*. *FLUSH++* extracts the best of *STALL* and *FLUSH* to outperform both.

Nevertheless, the three different RaT approaches are clearly ahead of all. Our mechanisms have the best throughput for each workload group, mainly for MEM workloads. For instance, RaT performs 74% and 39% better than *FLUSH++* for 2 and 4 memory-intensive threads respectively. The fact that RaT exploits the memory-level parallelism

during runahead thread executions avoids the need for stalling or flushing a thread initially, and thus it does not slowdown the program progress. This drawback causes memory-intensive threads do not achieve better performance in mixed and memory workloads with the evaluated I-fetch policies.

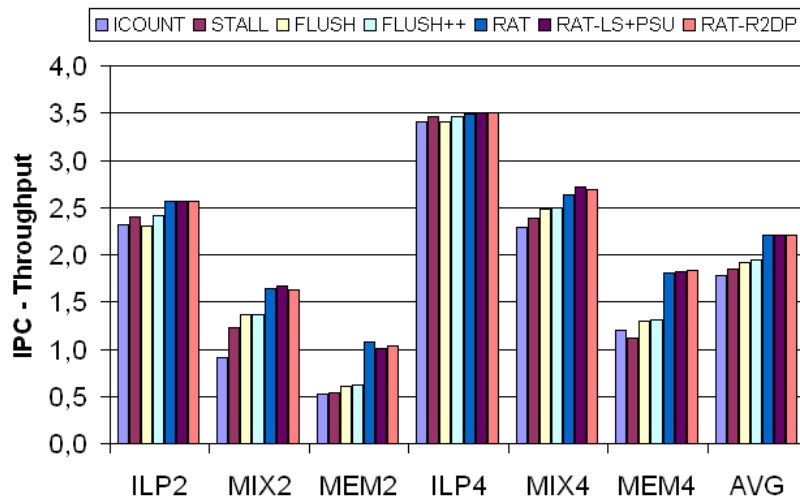


Figure 7.1: Average throughput for RaT mechanisms and the different I-Fetch policies

On the other hand, to evaluate the balance between fairness and throughput, Figure 7.2 compares the hmean metric of the different static I-fetch techniques evaluated here. Again, runahead thread-based mechanisms achieve the best results in terms of hmean, which represents a better performance-fairness among all techniques. Although the hmean improvement for ILP workloads is moderate, for instance RaT performs 8% for ILP2 and 2% for ILP4 better than FLUSH++, this improvement is much more significant for MEM workloads: RaT gets 51% and 37% hmean gain over FLUSH++ (the best of the I-fetch policies) for 2-thread and 4-thread workloads respectively. We also observe that the hmean among I-fetch techniques presents little difference in the case of 4-thread workloads, being close to ICOUNT results. Even, FLUSH suffers 3% hmean degradation for MEM workloads. These results show that RaT provides good performance improvement as well as fairness, mainly for memory intensive threads presented in MIX and MEM workloads.

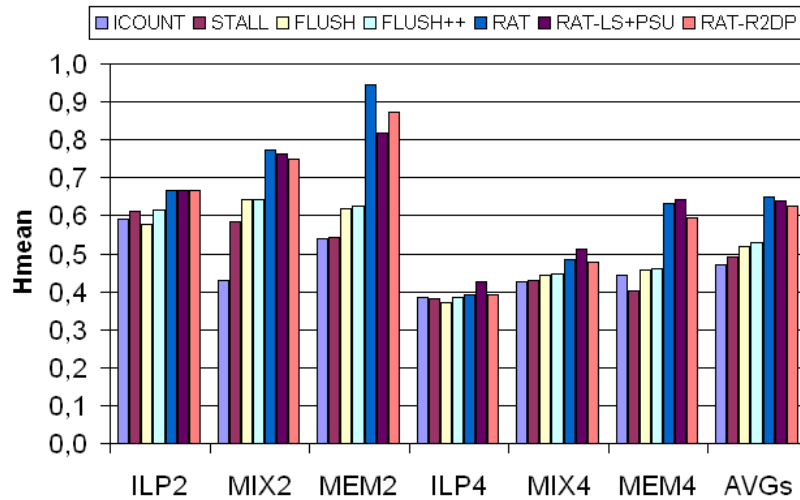


Figure 7.2: Average hmean for RaT mechanisms and the different I-Flush policies

RaT vs. Dynamic allocation techniques

We compare the RaT approaches with two dynamic resource partition policies: DCRA [7] that makes resource scheduling decisions based on resource utilization and, Hill-Climbing (HC) [11] that follows a dynamic partitioning strategy guided by performance. Regarding HillClimbing mechanism, we implement the approach guided by the performance function based on the throughput (named Hill-Thru in [11]). The other two approaches, that use weighted speedup and harmonic mean metrics as feedback measures, need the use of the IPC of each benchmark in single thread mode as an external input. These heuristics for HillClimbing require a machine that supports the capability to introduce this feedback by some way (e.g. from a costly sampling or external inputs). Besides, this issue implies the repetition of the single program execution to guide the mechanism, which at the same time is highly dependent on the variability of the program characteristics.

Figure 7.3 shows the average throughput for the baseline ICOUNT, DCRA, HC and RaT mechanisms. We can see that all techniques perform better than the baseline ICOUNT. For ILP4 workloads DCRA, HC and RaT techniques obtain similar performance but like previous static policies comparison, RaT outperforms both DCRA and HC in the rest of workloads due to its benefits for the memory-intensive threads. For MIX workloads, DCRA and HC performs well due to the dynamic resource scheduling among fast and slow threads. On the contrary, DCRA and HC do not work well

on MEM workloads, threads with high data cache miss rates and low baseline performance. Allocating more slots of resources to such threads in a normal execution is not enough to improve their performance without exploiting the MLP.

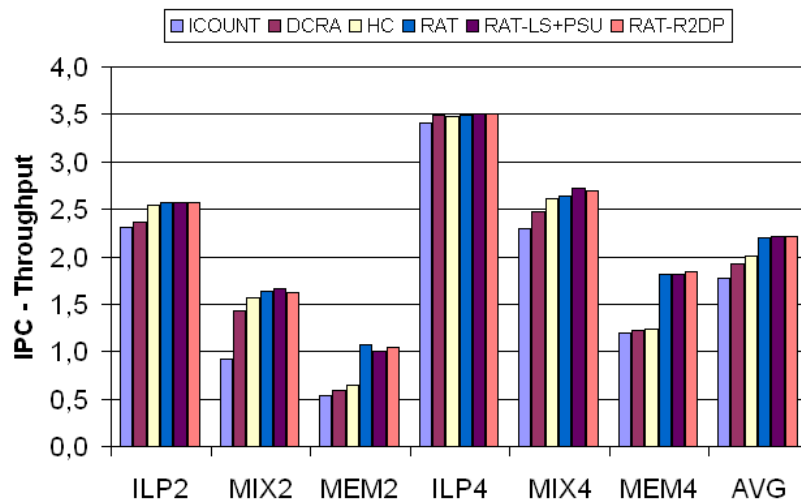


Figure 7.3: Average throughput for RaT and the different resource control policies

RaT provides an average performance gain of 27% and 20% compared to DCRA and HC respectively. This overall performance gain is achieved non-uniformly across the different workload groups. RaT excels again for all MEM workloads compared to these dynamic policies, both for 2-thread and 4-thread contexts as Figure 7.3 shows. Furthermore, the performance gains are larger for the 2-thread workloads (36% and 24% compared to DCRA and HC respectively) than for the 4-thread workloads (19% and 16% respectively).

In the case of hmean evaluation, which results are shown in Figure 7.4, RaT achieves better balanced throughput and fairness than the dynamic control policies in all workload categories. RaT outperforms DCRA by 31% and 19% for the 2-thread and 4-thread workloads and by 29% and 33% respectively compared to HillClimbing. This is a positive result given the diversity of our workload sets and the differences between the techniques evaluated. Likewise, RaT results are also good for MEM workloads, 53% better than DCRA and 55% better than HillClimbing, but RaT mechanisms do not neglect the hmean results for the other ILP and MIX workloads either. Remarkable is the fact that while RaT considerably outperforms ICOUNT by 19% for all 4-thread workloads, HillClimbing performs worse than ICOUNT in all 4-thread categories with

an average 10% of hmean lost. As HC makes the resource scheduling based on improving the performance, it favours high performance threads opposite to slow ones. This factor entails performance improvement differences among these threads, reducing the fairness of this technique when there are workloads with more threads.

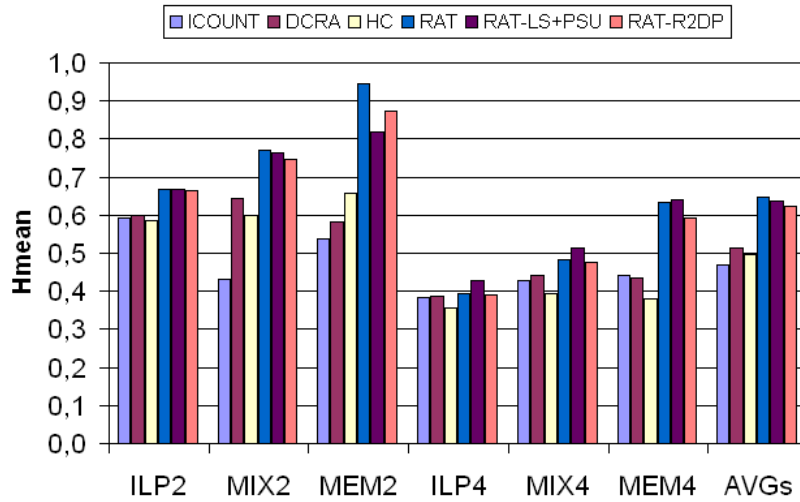


Figure 7.4: Average hmean for RaT and the different resource control policies

RaT vs. MLP-aware policies

To finalize the performance comparative with all the different state-of-the-art techniques in function of their category, in this section we evaluate RaT mechanisms versus other recent MLP-aware techniques. These techniques work like the STALL and FLUSH policies to handle long-latency loads but taking into account the nearby possible memory-level parallelism (MLP).

Figure 7.5 shows the throughput of MLP-STALL and MLP-FLUSH techniques together with RaT mechanisms for each of the different workloads. According to this figure, RaT-based mechanisms again get the best overall throughput. On average, RaT outperforms both MLP-aware techniques across all categories, 28% better throughput than MLP-STALL and 19% better than MLP-FLUSH.

Being also an MLP-aware technique, such MLP-STALL and MLP-FLUSH, run-ahead thread-based mechanisms considerably provide better performance in MEM workloads. Thus, RaT has a throughput improvement of 71% over MLP-STALL and 57% over MLP-FLUSH for MEM2 workloads and 42% and 32% respectively for MEM4

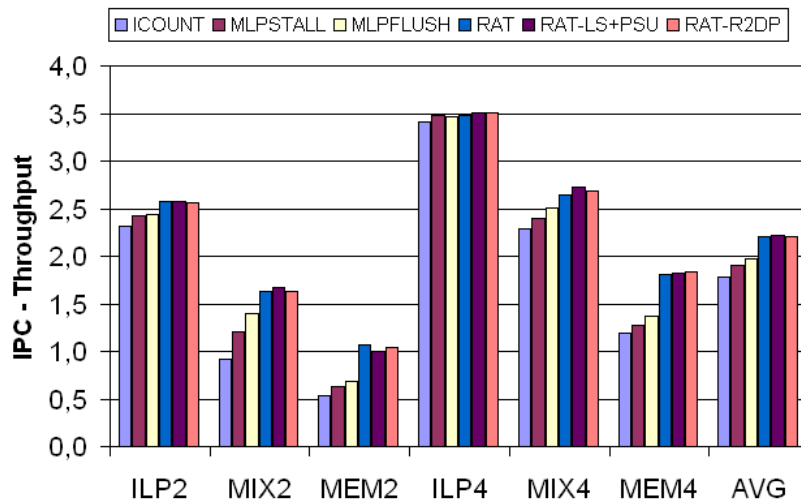


Figure 7.5: Average throughput for RaT and the different MLP-aware techniques

workloads. As explained in Chapter 6, these MLP-aware techniques are limited by the reorder buffer size in the amount of MLP they can exploit because a thread that misses in the L2 cache cannot execute more instructions than the reorder buffer size permits. In contrast, RaT can execute more instructions from a thread than the reorder buffer size permits in the shadow of an L2 miss because the speculative nature of runahead execution.

In Figure 7.6, we show the results regarding the hmean metric. On average, MLP-FLUSH is better than MLP-STALL but RaT outperforms both techniques improving MLP-STALL by 22% and MLP-FLUSH by 16%. For 2-threads, the difference in hmean is more significant, being RaT 6%, 12% and 34% better than MLP-FLUSH for ILP2, MIX2 and MEM2 workloads respectively. In the case of 4 threads, hmean improvements are 3%, 5% and 28% respectively when RaT mechanism is used.

All these results prove that it is preferable to exploit the memory-level parallelism to strictly limit the resources or stall the threads. Saving the penalty of a long-latency memory access has a bigger performance impact than the cycles of penalty due to resource conflicts. In addition, if we alleviate the former, we ease the latter. Therefore, although RaT mechanisms do not have any knowledge about the direct resource allocation among threads as other policies have, this speculative approach lets threads use a properly amount of resources to improve the performance while avoiding any possible resource monopolization. That is, RaT provides a right interaction between

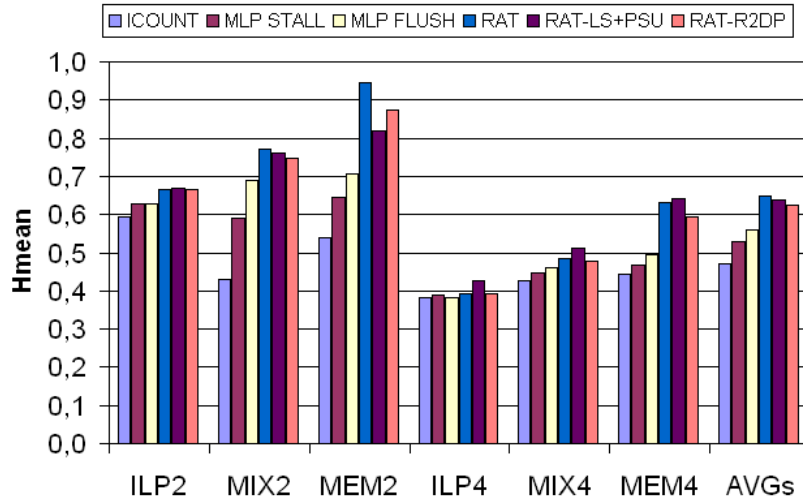


Figure 7.6: Average hmean for RaT and the different MLP-aware techniques

memory-bound threads turned into fast runahead threads and normal threads in order to obtain performance improvements using the available resources.

7.1.2 Extra instruction execution

Up to now, we have evaluated and compared the performance of Runahead Threads and the most representative SMT policies. We have shown in the comparison that the RaT mechanism and its enhancements to be more efficient outperform all previously proposed SMT policies both in terms of throughput and hmean. However, some of these techniques, including RaT, involve speculative execution which translates into a higher number of executed instructions. The flushing and re-execution of these extra instructions from a thread has an energy overhead. Analyzing and comparing the execution of this extra amount of instructions provides an approximation about the additional power consumption.

In addition to RaT mechanisms, among the previously evaluated techniques, the policies that re-execute additional instructions are FLUSH, FLUSH++ and MLP-FLUSH due to the flush action to handle the long-latency loads. These techniques repeat the execution of the instructions issued per each long-latency load until the point at which these instructions are squashed (e.g. when the long-latency load detection point is reached or the MLP prediction limit).

Figure 7.7 shows the average number of instructions executed by each technique according to the category of workloads. For simplicity, we only show in this figure the techniques that really execute a greater amount of additional instructions because the rest of techniques that are not shown execute a ratio of instructions similar to the baseline ICOUNT policy. Therefore, the techniques showed are FLUSH, FLUSH++, MLP-FLUSH and the three RaT approaches. This figure shows as RaT executes the larger number of instructions on average. For ILP workloads, there are few runahead threads activations due to less long-latency misses as shown in Figure 5.18, thereby there are no much speculation (11% more extra instructions). For the MEM workloads this quantity is higher, executing 86% on average more instructions than the baseline ICOUNT. However, using code semantic-aware runahead threads (RAT-LS+PSU) and efficiency-aware runahead threads (RAT-R2DP), this increase of executed instructions in MEM workloads is effectively reduced to 65%.

In the same way for the memory-intensive workloads, FLUSH executes 56% more instructions than ICOUNT whereas FLUSH++ executes 34% and MLP-FLUSH executes 35%. Comparing the results between them, FLUSH++ does less flushes than FLUSH, because FLUSH++ performs stall actions (instead of flushing) in fuction of the “continue oldest thread” heuristic. In the case of MLP-FLUSH, the exploitation of MLP avoids carry out some flush actions as result of long-latency load prefetching. On average, runahead thread efficient techniques are close in terms of executed instructions to FLUSH technique as Figure 7.7 shows but providing better performance.

As complementary data, Figure 7.8 shows the extra work ratio (EW). This ratio is computed as the number of executed instructions with respect to the committed instructions per workload. The higher the value of the EW the higher the number of speculative executed instructions respect to the total useful instructions. If there were not extra work (any type of speculative instructions), the EW ratio would be 1. In addition, this ratio is an approximation to the relation between the number of instructions executed by the baseline ICOUNT and the rest of techniques. For the baseline, excepting the wrong-path executed instructions which are much less in comparison with other speculative instructions (the former represents only a 5% of extra work), the rest of executed instructions are committed and therefore, they are considered useful work. As we can see in Figure 7.8, RaT has an average 1.55 EW ratio, which means this mechanism executes around 55% more instructions compared to the number of useful executed instructions. However, this EW is reduced to 1.42

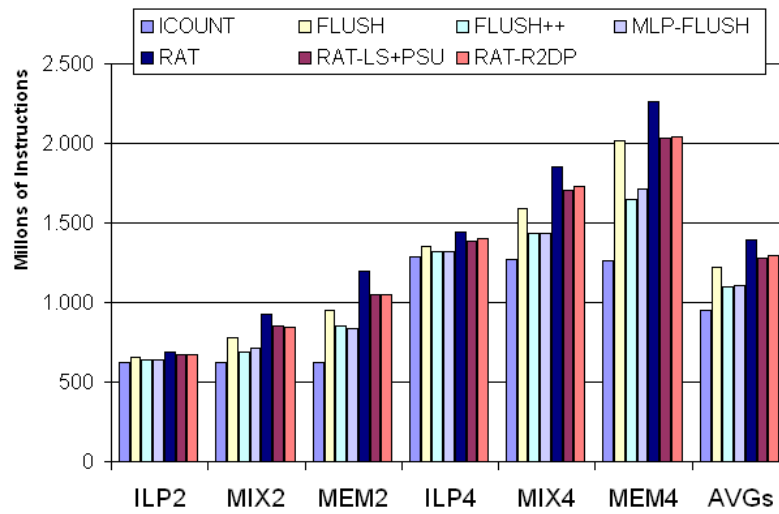


Figure 7.7: Average number of total executed instructions

on average applying our best approaches for runahead thread efficiency. In order of magnitude, from the higher to the lower, the rest of techniques have 1.35 for FLUSH, 1.23 for MLP-FLUSH and 1.22 for FLUSH++ extra work ratio respectively.

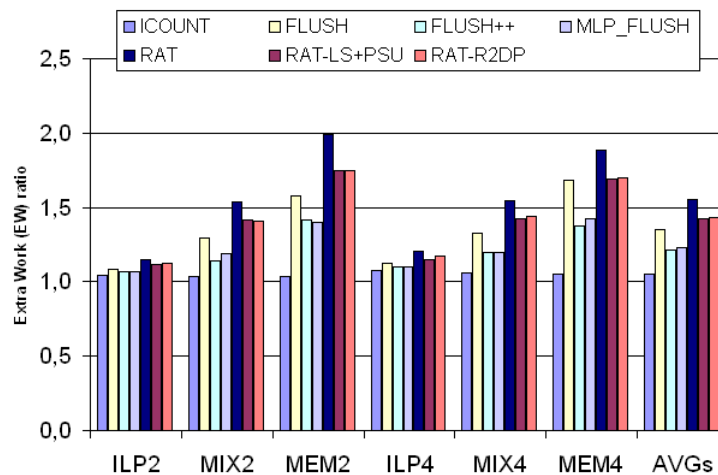


Figure 7.8: Extra work: ratio of executed instructions related to committed ones with regard each mechanism

7.2 Efficient runahead thread techniques

In the previous evaluation, we place the Runahead Thread mechanisms proposed in this dissertation in an overall frame regarding prior work related to all SMT mechanisms. However, the other main proposals of this work focus on techniques for improving the energy-efficiency of RaT reducing its extra work. To evaluate these other mechanisms in the right framework, we compare our two most-efficient techniques applied to RaT mechanism, RaT with LS+PSU and RaT with R2DP technique, with previous work related to make the runahead execution more efficient¹.

Inside the goal of this evaluation, we apply the techniques for efficient runahead execution which were proposed in a single threaded context [50], to our Runahead Thread mechanism. We implement the best performing heuristics from this work to eliminate runahead periods that are ineffective (short, overlapping, and useless runahead periods) in conjunction with RaT. We introduce the static threshold heuristic to eliminate short runahead periods, the full threshold policy to eliminate overlapping runahead periods, and the Runahead cause status table (RCST) to eliminate useless runahead periods. If one of these heuristics predicts a possible runahead thread to be short, overlapping, or useless, then that runahead thread is not initiated to prevent the processor from executing useless runahead instructions. We label in the figures this combination of Runahead Threads with these Single-thread Efficient runahead Techniques as RaT-SET.

Likewise, we also include in this evaluation the MLP-aware runahead thread approach [17] (MLP-RaT) proposed recently during the development of this dissertation. The researchers of this approach also support the important benefits of RaT mechanism in the SMT context and propose to use the MLP predictor of MLP-aware techniques to reduce the number of short runahead threads. In this proposal, a full runahead thread is initiated in case there is far-distance MLP to be exploited. Otherwise, an MLP-aware technique is applied for nearby MLP.

Furthermore, to emphasize the benefits of the proposed efficiency-aware mechanism, we include an aggressive prefetcher in the modeled SMT processor for this evaluation. An accurate hardware prefetcher can anticipate another possible L2 cache misses, specially if the memory accesses follow a strided pattern. This additional prefetching maybe soften the effectiveness of efficient runahead threads since the number of run-

¹We describe these prior energy-efficiency techniques for Runahead in Chapter 6.

head threads could be reduced due to prefetched long-latency loads. Including this prefetcher, we evaluate whether the mechanism still works properly in order to effectively reduce the useless runahead periods beyond the performance benefits coming from just prefetching. Therefore, the SMT processor employs an aggressive hardware prefetcher for all experiments described in this section. This prefetcher is based on a stream-based stride predictor that can detect and generate prefetches for different access streams into the L2 cache (similar to described in [29] and [70]). This prefetcher can bring ahead data into the L2 cache for 16 different access streams for each thread.

Following the line of this work, we perform this comparison with three main evaluation factors: performance, power and energy-delay².

7.2.1 Performance

We evaluate the overall performance of the described high-performance SMT processor model, measuring the IPC throughput and the hmean metric of all runahead-efficient mentioned techniques. The SMT processor includes the prefetcher commented before, which in the case of the baseline with ICOUNT, the use of this prefetcher improves the performance of the SMT processor by 16.6% on average. As representative of the previous state-of-the-art evaluation, we also include the MLP-aware flush policy (MLP-Flush) for comparison purposes.

Figure 7.9 shows the throughput performance of ICOUNT, MLP-Flush, RaT, RaT-SET, MLP-RaT, LS+PSU, and R2DP mechanisms for each category of workloads respectively. This figure demonstrates the performance gains achievable using mechanisms build on top of RaT. Although MLP-Flush improves the throughput over ICOUNT by 12% on average, the rest of mechanisms that support runahead threads provide even higher performance, especially for MIX and MEM workloads. Furthermore, these results show that the benefits of runahead threads and stream prefetching are complementary. R2DP obtains 33% and LS+PSU obtains 34% throughput average speedup over the baseline ICOUNT. Both are the techniques with the closest throughput performance to RaT (35%). RaT-SET and MLP-RaT also get 23% and 28% performance improvement respectively. The lower performance improvement of RaT-SET comes from the fact that this approach eliminates many useless runahead periods as well as useful runahead periods because of its binary prediction. In addition, the heuristics involved are not as fine-grained as the distance prediction considering

both the MLP and the useless work. In other words, depending on the decisions of heuristics using RaT-SET, the processor either executes the whole runahead thread fully until the L2 miss returns or it does not even enter runahead thread. So, RaT-SET eliminates extra work avoiding full runahead periods, but it also eliminates useful runahead periods degrading performance by reducing prefetching benefits. For MLP-RaT approach, the binary prediction in function of the MLP also selects either to execute or not a full runahead thread.

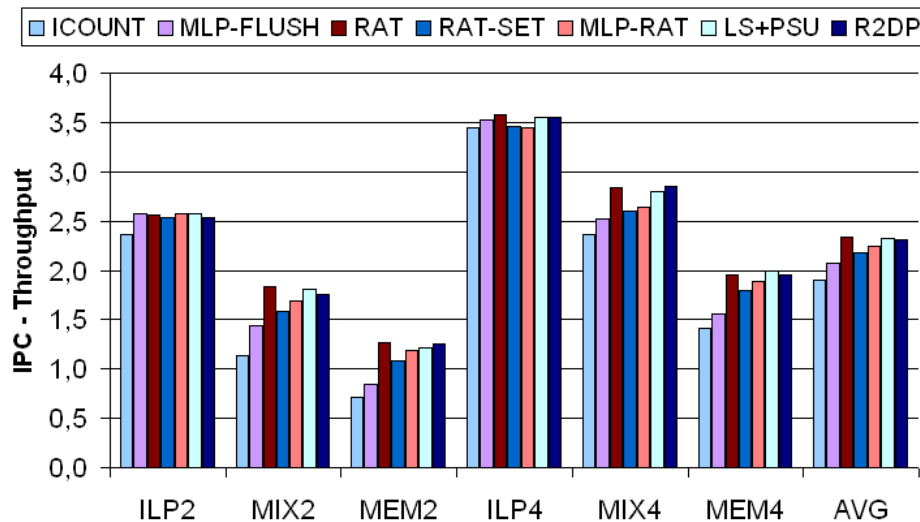


Figure 7.9: Throughput of our proposals versus efficient runahead and MLP-aware SMT fetch policies

On the other hand, the runahead distance prediction mechanism, R2DP, considers that decision as a particular case when the predicted distance is small. For this case, the processor does not activate the runahead thread like the previous two techniques do (RaT-SET and MLP-RaT). However, R2DP is more fine-grained with the executed runahead threads. The processor executes a runahead thread until the predicted useful runahead distance, that is, until the runahead threads stops being useful. Therefore, according of these issues and from the results of Figure 7.9, R2DP effectively preserves the throughput performance provided by runahead threads compared to the evaluated efficient runahead techniques.

Figure 7.10 shows the corresponding results for the hmean metric. As this figure shows, RaT-SET, MLP-RaT, LS+PSU, and R2DP obtain lower hmean performance

overall results compared to RaT. RaT provides 31% and 14% hmean speedup over ICOUNT for 2- and 4-thread workloads respectively. Next, MLP-RaT and R2DP have overall close results with 4% and 5% hmean inferior ratio respectively. Ls+PSU and RaT-SET present around 6% lower hmean compared to RaT. The ratio of useless runahead thread mispredictions in each technique degrades sometimes the individual IPC of memory-intensive threads compared to the original RaT mechanism, which reduces the hmean ratio. These hmean results confirm that runahead-based mechanisms present good throughput/fairness balance, only suffering slightly slowdowns among them, but significant outperforming the baseline processor.

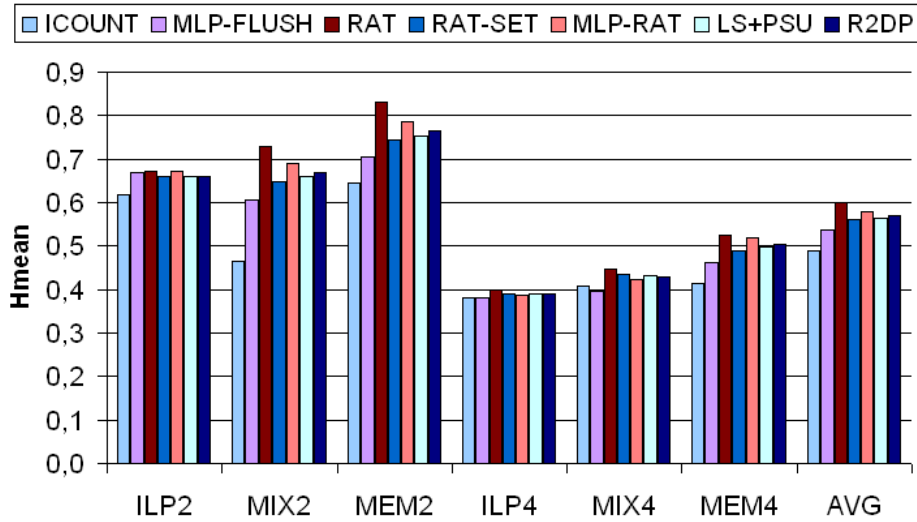


Figure 7.10: Hmean of our proposals versus efficient runahead and MLP-aware SMT fetch policies

7.2.2 Energy efficiency

After analyzing the performance results, we now study the implications of using the different techniques in terms of energy efficiency. We quantify the extra work reduction, the power consumption and performance-energy balance to show a detailed evaluation of the energy efficiency of the analyzed techniques.

Figure 7.11 shows the ratio of speculative executed instructions of runahead threads for the different techniques normalized to original RaT count (note we only show the mechanisms that incorporate runahead threads). According to this figure, MLP-RaT

reduces the speculative work by 10% (extra flushed instructions also count for speculative instructions) whereas applying the single-thread efficient techniques (RaT-SET) this reduction is 31%. As we said, in both mechanisms (MLP-RaT and RaT-SET), if a runahead thread is predicted to be executed, the processor fully executes the runahead thread until the L2 miss that caused enter into runahead mode is solved. For this reason, even if runahead execution does not provide any benefits beyond some point in the runahead thread, these mechanisms continue speculatively executing instructions.

On contrast, the reduction in speculative instructions using R2DP comes from eliminating both the complete useless runahead threads (as other techniques also do) and the useless runahead instructions at the end of useful runahead threads (thanks to the fine-grain runahead distance prediction). As result, R2DP achieves the highest speculative instruction reduction, with 44% on average according to the Figure 7.11. The combination of code semantic-aware techniques (LS+PSU) also achieves a good reduction with 37%.

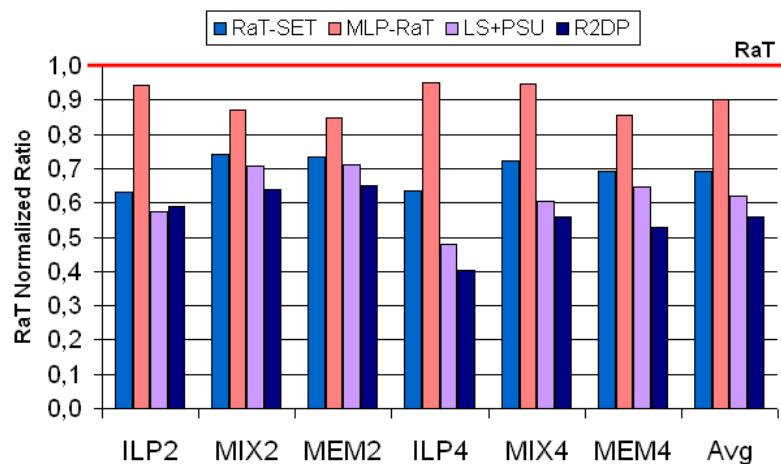


Figure 7.11: Speculative instructions ratio normalized to RaT

Figure 7.12 shows the processor power consumption when each different evaluated techniques (ICOUNT, MLP-Flush, RaT, RaT-SET, MLP-RaT, LS+PSU, and R2DP) is employed to quantify the benefits of its extra work reduction in terms of energy. For this evaluation, we measure the power consumption for all important processor core and memory components using our Wattch model during all workload executions.

All experiments also include the counting and power consumption of the additional hardware in function of the technique that is being evaluated.

At first glance, these power results show that the average power of the processor is correlated with the number of executed instructions, similarly as Annavaram *et al.* work [2] shows. For instance, the average power consumption for 4-thread workloads is higher than those with 2-thread, mainly because the IPC of the former is significantly higher.

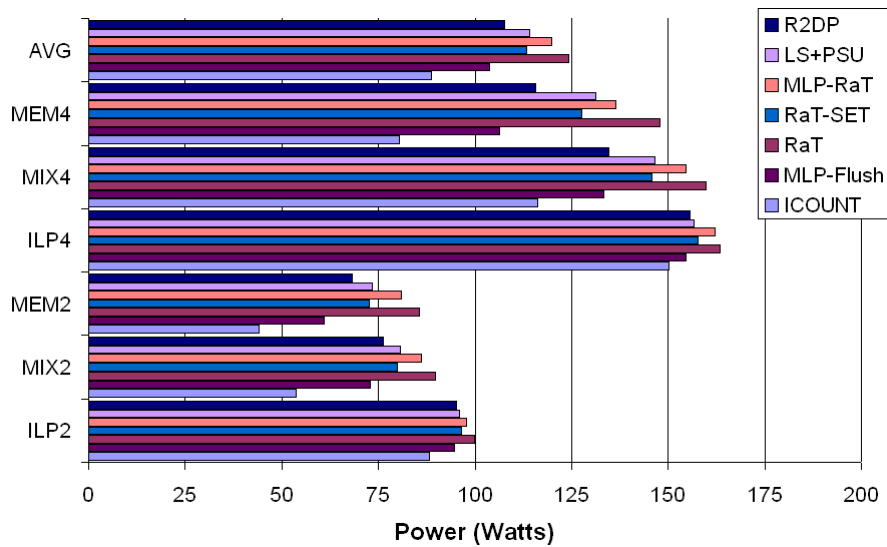


Figure 7.12: Average power consumption

Overall, all techniques consume more power than the baseline SMT with ICOUNT. For instance, MLP-Flush mechanism consumes 22% whereas RaT consumes 51% more power than ICOUNT. Regarding the techniques that control runahead thread executions, LS+PSU and RaT-SET reduce the power requirements by 10% over RaT, although the power reduction for the second one comes with a performance degradation (9% less performance compared to RaT as we have seen in Figure 7.9). MLP-RaT achieves 4% power reduction compared to RaT. Finally, R2DP effectively reduces the power consumption by 14% on average compared to RaT (up to 20% for MEM2 workloads), therefore reducing the power consumption 10% more than MLP-RaT.

In order to compare the final energy efficiency of the different evaluated mechanisms, the energy-delay square (ED^2) metric provides a fair comparison taking into account the energy savings and performance variation together. Figure 7.13 shows the ED^2

relation of the analyzed techniques compared to the non-controlled RaT mechanism. We calculate the ED^2 value of each technique and then, we normalize those results to RaT mechanism result for each bar. As we can observe, MLP-Flush do not provide a good balance between the performance gain and the power consumed (especially for MEM workloads). MLP-Flush has the worst energy efficiency ratio compared to RaT, 1.5 ED^2 ratio normalized to RaT, or what is equivalent to 50% worse than RaT on average.

On the other hand, RaT-SET and MLP-RaT show ED^2 normalized ratio over the original RaT. In spite of these mechanisms reduce the speculative runahead instructions and power consumption (as it just is shown in Figure 5.12), the performance degradation of these techniques compared to RaT, 9% for RaT-SET and 4.5% for MLP-RaT, causes energy-efficiency levels below than RaT. Therefore, RaT-SET and MLP-RaT energy efficiency are lower, with 20% and 12% ED^2 ratio degradation respectively compared to RaT.

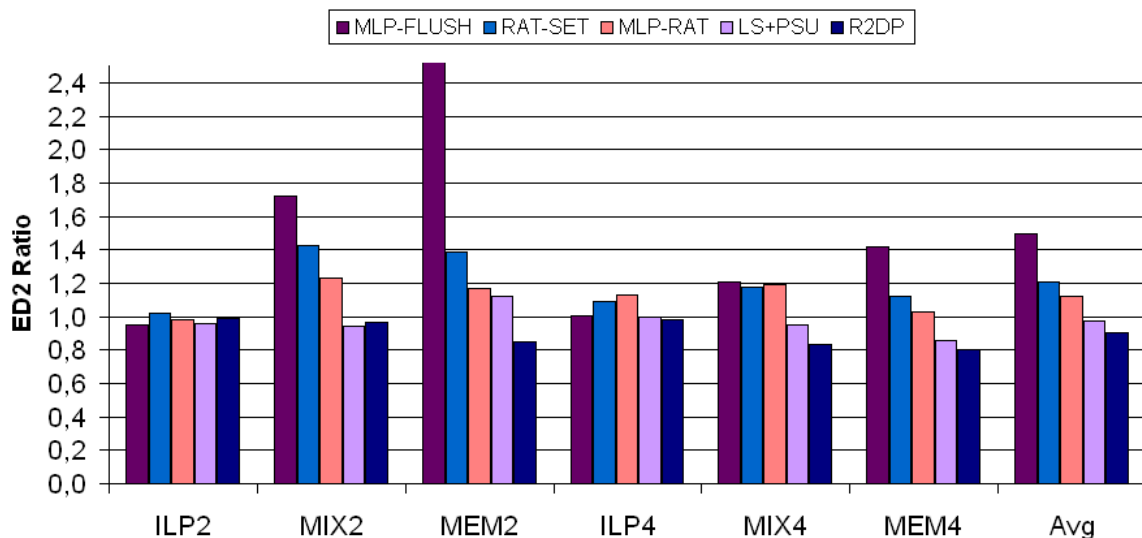


Figure 7.13: Energy-delay² for the evaluated mechanisms compared to RaT

Applying LS+PSU technique, we achieve 3% on average better ED^2 compared to RaT alone. However, Figure 7.13 shows that R2DP provides the best ED^2 product results for the different workloads category, with an average 10% better than RaT. R2DP provides 7% better ED^2 than RaT for 2-thread workloads. The energy efficiency is even better for 4 threads, since R2DP improves ED^2 by 13% over RaT for these workloads.

RaT-SET and MLP-RaT energy efficiency are lower, with 24% and 33% ratio degradation respectively compared to R2DP. As we have shown in this section, this is because R2DP reduces the highest amount of speculative instructions (44% compared to RaT) causing less power consumption while it provides similar performance. Therefore, one important conclusion derived from this evaluation is that SMT processors are more efficient in terms of ED^2 when incorporate RaT combined with R2DP.

7.3 Summary

This chapter provides a comprehensive evaluation of RaT mechanisms of this dissertation with state-of-the-art mechanisms related to SMT processors. Furthermore, we cover all Runahead techniques focused on improving the efficiency applied to the Runahead Threads. All these experiments and empirical analysis presented about this variety of SMT techniques provide support for different concluding remarks.

Figure 7.14 summarizes the overall performance of techniques with the best throughput results per category evaluated in this chapter. We see from the figure that RaT presents the best performance throughput compared to the state-of-the-art.

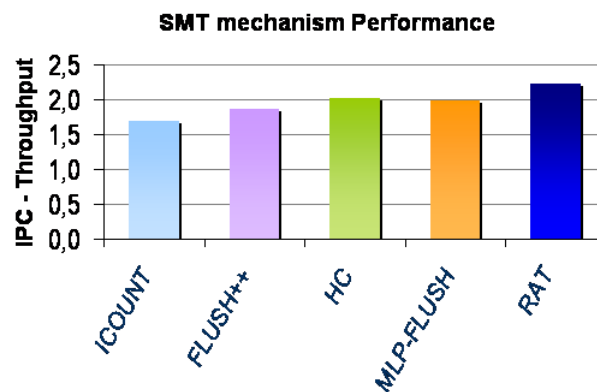


Figure 7.14: Overall performance of SMT state-of-the-art techniques

Figure 7.15 summarizes the ED^2 ratio of RaT-based mechanisms with regard to the baseline SMT processor. The RaT-R2DP technique provides the higher energy-efficiency ratio comparing to the previously proposed efficient Runahead techniques. So, R2DP mechanism executes the most efficient runahead threads which are effective at yielding both high performance and energy efficiency.

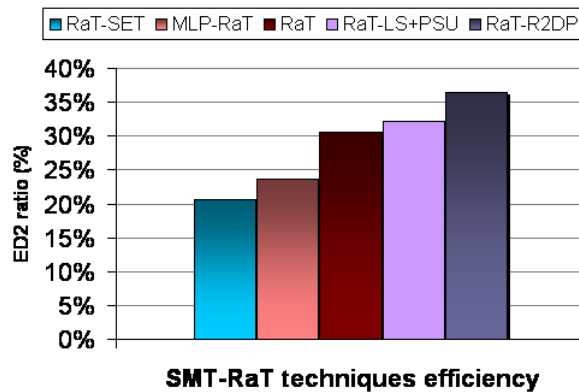


Figure 7.15: Overall efficiency (ED²) ratio compared to SMT baseline

The comparison of SMT mechanisms it is important to understand the fundamental difference in terms of performance, power and energy efficiency between them before we try to consider incorporating in SMT. This exploration shows that Runahead thread-based mechanisms can become an energy-efficient design paradigm in terms of ED² and can provide a 33% performance improvement for a varied mixed of workloads with a power overhead of around 28%. According with the study of this chapter, efficiency-aware runahead threads by R2DP tend to be relatively power efficient compared to prior approaches and equivalent throughput performance to the best RaT mechanism.

Designers of future SMT systems will be increasingly required to optimize for a combination of single-thread performance, total performance throughput, and energy consumption. Therefore, considering all this overall analysis performed, RaT-based mechanisms are suitable techniques that can be implemented for these different final SMT scenarios. In high-performance scenarios, our different RaT proposals are good candidate mechanisms that provides significant improvement both in throughput and hmean. In case of power-aware scenarios, we also provide techniques to address more efficient runahead thread speculation.

Conclusions

In this chapter, we summarize the fundamentals and main contributions of this work and we outline some future lines of research derived from this thesis.

8.1 Goals and conclusions

Nowadays, researches on multithreading topics have gained a lot of interest in the computer architecture community due to new commercial multithreaded and multi-core processors. All these current trends rely on exploiting thread level parallelism (TLP) due to the lack of more instruction level parallelism (ILP) to exploit at typical instruction streams and the bottlenecks of previous single-thread processor generations, such as long-latency cache misses, branch mispredictions and current power constraints.

One of these trends is *Simultaneous Multithreaded* (SMT) processors. The main feature of SMT processors is to execute multiple threads that increase the utilization of the pipeline by sharing many more resources than in other types of processors. Shared resources are the key of simultaneous multithreading, what makes the technique worthwhile. However, this feature also entails important challenges to deal with because threads also compete for resources in the processor core. On the one hand, although certain types and mixes of applications truly benefit from SMT, the different features of threads can unbalance the resource allocation among threads, diminishing the benefit of multithreaded execution. On the other hand, the memory wall problem is still present in these processors. SMT processors alleviate some of the latency prob-

lems arise by DRAM memory's slowness relative to the CPUs. Nevertheless, threads with high cache-miss rates that use large working sets are one of the major pitfalls of SMT processors. These memory-intensive threads tend to use processor and memory resources poorly creating the highest resource contention problems. Memory-intensive threads can clog up shared resources due to long-latency memory operations without making progress on a SMT processor, thereby hindering overall system performance. Alleviating these shortcomings on SMT scenarios is the main point that guides the goal for this thesis.

To accomplish this, the key contribution of this thesis is that we have introduced the paradigm of Runahead execution in the design of multithreaded processors for the first time. **Runahead Threads** (RaT) shows to be a promising mechanism that improves both single-thread performance and multi-thread performance in SMT processors. The original idea to apply RaT is to transform a resource intensive thread (memory-bound thread) into a light-consumer thread by allowing that thread to progress speculatively while it has a long-latency miss outstanding. Therefore, as soon as a thread undergoes a long-latency load, RaT transforms that thread to a *runahead thread*. The main benefits of this simple action performed by RaT is twofold. While being a runahead thread, this thread uses the different shared resources without monopolizing or limiting the available resources for other threads. At the same time, this fast speculative thread issues prefetches that overlap other memory accesses with the main miss, thereby exploiting the memory-level parallelism and improving the performance.

In fact, RaT implies an alternative to prior SMT resource management mechanisms which usually restrict memory-bound threads in order to get higher throughput. By means of Runahead Threads, we contribute to solve simultaneously two shortcomings in the context of SMT processor:

1. RaT alleviates the long-latency load problem on SMT processors by exposing memory-level parallelism (MLP). A thread prefetches data in parallel (if MLP is available) improving its individual performance rather than be stalled on an L2 miss. So, RaT preserves the MLP, and even allows for exploiting far-distance MLP beyond the scope of the reorder buffer by allowing threads to execute speculatively.
2. RaT prevents threads from clogging resources on long-latency loads. RaT ensures that the L2-missing thread recycles faster the shared resources it uses by the

nature of runahead speculative execution. This avoids memory-intensive threads clogging the important processor resources up.

Regarding implementation issues, Runahead Threads adds very little extra hardware cost and complexity to an existing SMT processor. Through a simple checkpoint mechanism and a little additional control logic, we can equip the hardware contexts with the runahead thread capability.

RaT has also been deeply studied. We provide a detailed performance evaluation together with several analysis about the sensitivity of processor parameters and design issues related to RaT. Among the results and conclusions about all this deep study, we remark that SMT processor with RaT performs much better in terms of throughput and Hmean. Additionally, RaT significantly improves the performance of an SMT processor as the memory latency gets longer. Therefore, RaT is a good approach for increasing the tolerance of SMT processors to future long main memory latencies. Furthermore, RaT has an important advantageous feature related to one of the most critical resources of an SMT, because it allows using smaller register files on SMT processors.

The main limitation of RaT though is that runahead threads can execute useless instructions and unnecessarily consume execution resources on the SMT processor (functional unit slots, issue queue slots, reorder buffer entries, etc.) even if there is no prefetching to be exploited. This drawback results in inefficient runahead threads which do not contribute to the performance gain and increase dynamic energy consumption due to the number of extra speculatively executed instructions. Therefore, we also proposed different solutions aimed at this major disadvantage of the initial Runahead Threads mechanism.

In particular, we devise several schemes for executing more efficient runahead threads as well as reduce their power consumption. The result of this research is a set of complementary solutions to enhance RaT in terms of energy requirements and efficiency. These proposals are:

- Code semantic-aware Runahead threads
- Efficiency-aware Runahead threads

Code semantic-aware Runahead threads improve the efficiency of RaT using coarse-grain code semantic analysis at runtime. We provide different techniques that

analyze the usefulness of certain code patterns during runahead thread execution. The code patterns selected to perform that analysis are loops and subroutines, which are chosen based on criteria such as instruction granularity and ease of detection. By means of the proposed coarse-grain analysis, runahead threads oversee the usefulness of loops or subroutines depending on the prefetches opportunities during their executions. Thus, runahead threads decide which of these particular program structures execute depending on the obtained usefulness information, deciding either stall or skip the loop or subroutine executions to reduce the number of useless runahead instructions. Some of the proposed techniques reduce the speculative instruction and wasted energy while achieving similar performance to RaT.

On the other hand, the ***efficiency-aware runahead thread*** proposal is another contribution focused on improving RaT efficiency. This approach is based on a generic technique which covers all runahead thread executions, independently of the executed program characteristics as code semantic-aware runahead threads are. The key idea behind this new scheme is to find out “when” and “how long” a thread should be executed in runahead mode by predicting the *useful runahead distance*. Our results show that the best of these approaches based on the runahead distance prediction significantly reduces the number of extra speculative instructions executed in runahead threads, as well as the power consumption. Likewise, it maintains the performance benefits of the runahead threads, thereby improving the energy-efficiency of SMT processors using the RaT mechanism.

To contrast the RaT mechanisms advantages, a detailed evaluation of the proposals in this dissertation and the state-of-the-art SMT mechanisms is provided. This evaluation represents a complete survey about the SMT mechanisms, including performance results and novel data regarding power consumption as quantitative contribution. This comparison shows that RaT mechanisms perform better than any prior SMT mechanism proposed to manage the resource contention. RaT outperforms on average all techniques for the workloads evaluated in terms of both throughput and hmean, especially in the case of MIX and MEM workloads. In addition, the resulting efficient runahead threads reduces power consumption while maintaining the high performance improvements of RaT, providing better performance and energy balance (measure through the ED² metric) than previous proposals in the field. Overall, the different RaT approaches of this dissertation may be appropriate for scenarios that require a mixture of high-performance and power-efficiency designs.

8.2 Remarks and future directions

During this thesis, we have extended high-performance SMT processors with an energy efficiency evolution of Runahead Thread mechanisms. The evolution of Runahead Threads developed in this research provides not only a high-performance but also an efficient way of managing shared resources in SMT processors in the presence of long-latency memory operations. This dissertation provides a complete initial proposal for general efficient runahead threads.

Nevertheless, the different techniques presented here can be further enhanced or extended opening new research lines. As future work, techniques aimed to reduce the resource utilization by runahead threads can improve the simplicity and resource requirements of this dissertation proposals. One research line could focus on runahead threads that do not use the reorder buffer entries. These resource-aware runahead threads will leave these important resource slots free for instructions belonging to the other normal threads. For instance, introducing small buffers or queues that only keep the needed information related to the correct execution of valid runahead instructions can be a possible solution.

Cross-pollination of Runahead threads and resource management mechanisms can be another starting point for further research. For instance, dynamic resource control mechanisms (e.g. DCRA and HillClimbing) are orthogonal to the mechanism proposed in this thesis in the way that is possible to incorporate an additional resource control mechanism to avoid possible inefficient resource utilization among normal and runahead threads. Logically, handling this new situation requires the adaptation and modification of the policies and a new space of exploration.

Further research is also necessary to determine new ways to improve the effectiveness of proposals focused on the efficiency of runahead threads. Addressing more cost-effective and accurate runahead speculation require additional design trade-offs. In this sense, we propose to extend the code semantic-aware runahead threads to cover other kind of code patterns or identify and classify code sections in function of their memory behavior.

Furthermore, it would be possible to apply some profile-based mechanisms for predicting the runahead distance in order to design an even more accurate runahead distance predictor. The combination of profiling compiler hints with hardware predictors will provide even better performance and efficiency. On the other hand, techniques

to take advantage of performance and power requirement knowledge of the programs being executed in the threads can be an interesting area of research. These possible techniques could make dynamic decisions to adapt the level or aggressiveness of runahead speculation in function of those requirements.

Recent product announcements show a clear trend towards aggressive integration of multiple cores on a single chip. A hot topic research is to analyze new implementation ways of runahead threads in chip-multiprocessors (CMP) or many-core processors. Maybe, decoupled designs with execution and runahead cores can help to improve the performance and energy efficiency of these processors, keeping under control the power consumption of runahead threads. For instance, under peak power consumption conditions, runahead cores can be deactivated in order to get more power savings.

Finally, we want to remark that the obtained results together with the cost-effective design make the RaT mechanisms a promising and appropriate choice on SMT frameworks. For this reason, some industrial work has announced processors that have started to implement Runahead-like mechanisms in current architecture designs. One example is the load lookahead (LLA) execution employed in the IBM POWER6 processor [26] to facilitate load prefetching to the L1 D-cache in its in-order execution. Another commercial example is the ROCK processor [10][71] from Sun Microsystems. ROCK is a chip-multithreading processor (CMT), which contains 16 high-performance cores, each of which can support two threads with speculative out-of-order retirement of instructions. Rock uses a novel checkpoint-based architecture to support automatic hardware scout threads (Sun codename for Runahead threads) under a load miss. The ROCK processor has been implemented and is scheduled to be commercially available soon.

List of Figures

1.1	Different approaches of multithreaded processor	3
1.2	Effects of long-latency misses on SMT processors	8
2.1	Block diagram of SMT processor architecture	16
3.1	A runahead thread is started when a long-latency load is detected . . .	33
3.2	Type of instructions while a runahead thread is executed	35
3.3	Runahead thread recovery	36
3.4	RaT processor architecture	42
3.5	Runahead Threads throughput performance vs. SMT baseline for 2- thread workloads	43
3.6	Runahead Threads throughput performance vs. SMT baseline for 4- thread workloads	44
3.7	Individual thread speedup between baseline and runahead performance for 2-thread workloads	45
3.8	Individual thread speedup between baseline and runahead performance for 4-thread workloads	46
3.9	Hmean of Runahead Threads vs. SMT baseline for 2-thread workloads	47
3.10	Hmean of Runahead Threads vs. SMT baseline for 4-thread workloads	48
3.11	Dcache and L2cache miss rate reduction by RaT	49
3.12	Prefetch improvement percentage by RaT	50
3.13	Resource conflict relation differences between baseline and RaT	51
3.14	Ratio of extra executed instructions for 2- and 4-thread workloads due to Runahead Threads	52
3.15	Total cycles distribution of runahead thread executions for 2-thread workloads	54

3.16	Total cycles distribution of runahead thread executions for 4-thread workloads	54
3.17	Distribution of runahead cycles per thread for MIX2 workloads	55
3.18	RaT performance evolution according to the number of threads per workload	57
3.19	RaT performance with different main memory latencies for 2-thread workloads	59
3.20	RaT performance with different main memory latencies for 4-thread workloads	60
3.21	Performance throughput with different L2 cache sizes for 2-context processor	61
3.22	Performance throughput with different L2 cache sizes for 4-context processor	61
3.23	Performance throughput with different ROB sizes for 2-context processor	62
3.24	Performance throughput with different ROB sizes for 4-context processor	63
3.25	Shared vs. Non-Shared ROB performance	64
3.26	Performance throughput for different number of renaming registers . .	66
3.27	Average physical registers used per cycle between normal and runahead threads	68
3.28	RaT throughput according to underline Icount setup for 2-thread workloads	70
3.29	RaT throughput according to underline Icount setup for 4-thread workloads	70
3.30	Performance of RaT in function of the different thread priority schemes	72
3.31	Evaluating performance of RaT with different checkpointing delays . .	74
3.32	Runahead Threads performance without and with runahead cache for 2-thread workloads	75
3.33	Runahead Threads performance without and with runahead cache for 4-thread workloads	75
4.1	Distribution of executed instructions for 2- and 4-thread workloads . .	81
4.2	Loop detection example	85
4.3	Loop usefulness mechanism example	87
4.4	Hardware requirements of loop control techniques	88

4.5	Subroutine control mechanism	92
4.6	Subroutine control mechanism	93
4.7	Subroutine PSU control technique	95
4.8	Hardware requirements of subroutine control techniques	96
4.9	Backward branches founded during normal threads and runahead threads execution	98
4.10	Loops detected applying our detection loop mechanism for normal threads and runahead threads	98
4.11	Loop usefulness breakdown for 2- and 4-thread workloads	99
4.12	Subroutines detected in the runahead threads	101
4.13	Subroutine usefulness breakdown for 2- and 4-thread workloads	102
4.14	Percentage of loops controlled by LR and LS techniques	103
4.15	Percentage of subroutines controlled by SK and SS techniques	103
4.16	Performance differences when saturated counters are used for loop control techniques	104
4.17	Performance differences when saturated counters are used for subroutine control techniques	105
4.18	Throughput results for the code semantic-aware techniques	106
4.19	Hmean results for the code semantic-aware techniques	106
4.20	Throughput results for code semantic-aware technique mixes	107
4.21	Speculative executed instructions analysis in function of the code semantic-aware techniques	109
4.22	Average extra power reduction	110
5.1	Illustration of runahead thread execution and the concept of <i>useful runahead distance</i>	115
5.2	Flowchart of Runahead Distance Prediction (RDP)	117
5.3	Flowchart of Runahead Two Distance Prediction (R2DP)	122
5.4	total-runahead-distance counter and useful-runahead-distance register functionality	123
5.5	Runahead Distance Information Table configurations	124
5.6	Performance (throughput) and extra work (speculative instructions) ratio with regard to RaT for different DDT values	125

5.7	Performance (throughput) and extra work (speculative instructions) ratio with regard to RaT for different values of N for the zero distance control heuristic for R2DP	127
5.8	Percentage of runahead threads not started due to zero distance value	128
5.9	Performance throughput and speculative executed instructions for 2-thread workloads	129
5.10	Performance throughput and speculative executed instructions for 4-thread workloads	130
5.11	Ratio of speculative executed instructions per runahead thread (normalized to RaT mechanism)	132
5.12	Average power consumption for the different mechanisms	133
5.13	Ratio of energy-delay ² product compared to RaT	134
5.14	Dynamic versus fixed runahead distance mechanisms	135
5.15	Accuracy of RDP techniques for predicting the ideal runahead distance	137
5.16	Predicted averages useful distances for RDP and R2DP mechanisms .	138
5.17	Average number of instructions executed per runahead thread in function of used mechanism (RaT, RDP and R2DP)	139
5.18	Number of runahead threads executed per each mechanism	140
5.19	Runahead thread executions breakdown	141
5.20	Runahead threads distance histogram for R2DP and RaT	142
7.1	Average throughput for RaT mechanisms and the different I-Fetch policies	159
7.2	Average hmean for RaT mechanisms and the different I-Fetch policies .	160
7.3	Average throughput for RaT and the different resource control policies	161
7.4	Average hmean for RaT and the different resource control policies . . .	162
7.5	Average throughput for RaT and the different MLP-aware techniques .	163
7.6	Average hmean for RaT and the different MLP-aware techniques	164
7.7	Average number of total executed instructions	166
7.8	Extra work: ratio of executed instructions related to committed ones with regard each mechanism	166
7.9	Throughput of our proposals versus efficient runahead and MLP-aware SMT fetch policies	169
7.10	Hmean of our proposals versus efficient runahead and MLP-aware SMT fetch policies	170

7.11	Speculative instructions ratio normalized to RaT	171
7.12	Average power consumption	172
7.13	Energy-delay ² for the evaluated mechanisms compared to RaT	173
7.14	Overall performance of SMT state-of-the-art techniques	174
7.15	Overall efficiency (ED ²) ratio compared to SMT baseline	175

List of Tables

2.1	SMT processor baseline configuration	18
2.2	SPEC2000 benchmarks characterization	23
2.3	Workloads of 2 threads used in this thesis	24
2.4	Workloads of 4 threads used in this thesis	25
3.1	RaT performance speedup according to the number of threads	57

Bibliography

- [1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, pages 423–434, Washington, DC, USA, 2003.
- [2] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahls law through epi throttling. In *Proceedings of 32nd International Symposium on Computer Architecture (ISCA)*, pages 298–309, Madison, Wisconsin USA, 2005.
- [3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, Vancouver, British Columbia, Canada, 2000.
- [4] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro journal*, 20(6):26–44, 2000.
- [5] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997.
- [6] F. J. Cazorla, E. Fernández, A. Ramírez, and M. Valero. Improving memory latency aware fetch policies for SMT processors. In *Proceedings of 5th International Symposium High Performance Computing (ISHPC)*, Tokyo-Odaiba, Japan, 2003.
- [7] F. J. Cazorla, A. Ramirez, E. Fernandez, and M. Valero. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, pages 171–182, 2004.
- [8] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dcache Warn: an I-fetch policy to increase smt efficiency. In *Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 74–, 2004.
- [9] R. S. Chappell, J. Stark, S. K. Reinhardt, Y. N. Patt, and S. P. Kim. Simultaneous subordinate microthreading (SSMT). *Proceedings of 26th International Symposium on Computer Architecture (ISCA)*, 1999.

-
- [10] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. ROCK: A high-performance Sparc CMT processor. *IEEE Micro journal*, 29(2):6–16, 2009.
- [11] S. Choi and D. Yeung. Learning-based SMT processor resource distribution via hill-climbing. In *Proceedings of 33rd International Symposium on Computer Architecture (ISCA)*, pages 239–251, Washington, DC, USA, 2006.
- [12] Y. Chou, B. Fahs, and S. G. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of 31st International Symposium on Computer Architecture (ISCA)*, pages 76–89, München, Germany, 2004.
- [13] J. Coke, H. Baliga, N. Cooray, E. Gamsaragan, P. Smith, K. Yoon, J. Abel, and A. Valles. Improvements in the Intel’s Core2 Penryn processor family architecture and microarchitecture. *Intel Technology Journal (ITJ)*, 12(3):179–192, 2008.
- [14] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, 2001.
- [15] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings of 28th International symposium on Computer architecture (ISCA)*, 2001.
- [16] Computer Corporation. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, 2000.
- [17] K. Craeynest, S. Eyerma, and L. Eeckhout. MLP-aware runahead threads in a simultaneous multithreading processor. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 110–124, 2009.
- [18] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramírez, M. Pericas, and M. Valero. Kilo-instruction processors: Overcoming the memory wall. *IEEE Micro journal*, 25(3):48–57, 2005.
- [19] A. Cristal, M. Valero, A. Gonzalez, and J. Llosa. Large virtual ROB’s by processor checkpointing. *Technical Report UPC-DAC-2002-39, Departament d’Arquitectura de Computadors, Universitat Politècnica de Catalunya*, July 2002.
- [20] M. de Alba and D. Kaeli. Path-based hardware loop prediction. In *Proceedings of the International Conference on Control, Virtual Instrumentation and Digital Systems*, Washington, DC, USA, 2002.
- [21] M. R. de Alba and D. R. Kaeli. Runtime predictability of loops. In *Proceedings of the IEEE International Workshop of Workload Characterization (WWC-4)*, Washington, DC, USA, 2001.

-
- [22] M. Dubois and Y. H. Song. Assisted execution. Technical report, University of Southern California, 1998.
- [23] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11st International Conference on Supercomputing (ICS)*, pages 68–75, New York, NY, USA, 1997.
- [24] A. El-Moursy and D. H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, page 31, Washington, DC, USA, 2003.
- [25] J. Emer. Simultaneous multithreading: multiplying Alpha performance. In *Microprocessor Forum*, pages 68–75, 1999.
- [26] H. L. et al. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6), 2007.
- [27] S. Eyerman and L. Eeckhout. A memory-level parallelism aware fetch policy for SMT processors. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA)*, 2007.
- [28] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro journal*, 28(3):42–53, 2008.
- [29] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of 24th Annual International Symposium on Computer Architecture (ISCA)*, pages 133–143, 1997.
- [30] A. Garcia, O. J. Santana, E. Fernandez, P. Medina, and M. Valero. LPA: A first approach to the loop processor architecture. In *Proceedings of 3rd International conference on High-Performance Embedded Architectures and Compilers (HIPEAC)*, pages 273–287, 2008.
- [31] A. Glew. MLP yes! ILP no! In *Wild and Crazy Ideas Session, 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.
- [32] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. Introduction to Intel Core Duo processor architecture. *Intel Technology Journal (ITJ)*, 10(2):89–97, 2006.
- [33] R. Gonçalves, E. Ayguadé, M. Valero, and P. Navaux. Performance evaluation of decoding and dispatching stages in simultaneous multithreaded architectures. *Proceedings of International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2001.
- [34] S. Hily and A. Sez nec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. Technical Report PI-1086, INRIA, 1997.

-
- [35] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *ACM Computer Architecture News (CAN)*, pages 34–42, 1991.
- [36] R. Kalla, B. Sinharoy, and J. Tendler. SMT implementation in POWER 5. *Hot Chips*, 2003.
- [37] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Workshop on Memory Performance Issues (WMPI)*, 2002.
- [38] D. Koufaty and D. T. Marr. Hyperthreading technology in the Netburst microarchitecture. *IEEE Micro journal*, 23(2):56–65, 2003.
- [39] K. Krewell. Fujitsu makes sparc see double. *Microprocessor Report*, 2003.
- [40] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, pages 81–92, 2003.
- [41] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th annual international symposium on Computer architecture (ISCA)*, pages 59–70, Washington, DC, USA, 2002.
- [42] J. L.-J. Lo. *Exploiting thread-level parallelism on simultaneous multithreaded processors*. PhD thesis, University of Washington Computer Science and Engineering, 1998.
- [43] K. Luo, M. Franklin, S. S. Mukherjee, and A. Sez nec. Boosting SMT performance by speculation control. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*, Los Alamitos, CA, 2001. IEEE Computer Society.
- [44] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Tucson, Arizona, USA, 2001.
- [45] D. T. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal (ITJ)*, 6(1), 2002.
- [46] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th annual International symposium on Microarchitecture (MICRO)*, pages 3–14, Los Alamitos, CA, USA, 2002.
- [47] S. A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing frontiers (CF '04)*, page 162, 2004.
- [48] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, (38) April:114–117, 1965.

-
- [49] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: an implementation of operation-based prediction. In *Proceedings of the 15th international conference on Supercomputing (ICS)*, pages 321–334, New York, NY, USA, 2001. ACM.
- [50] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32th annual international symposium on Computer architecture, (ISCA)*, pages 370–381, 2005.
- [51] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, page 129, Washington, DC, USA, 2003.
- [52] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *SIGOPS Operating Systems Review*, 30(5):2–11, 1996.
- [53] G. M. Papadopoulos and K. R. Traub. Multithreading: a revisionist view of dataflow architectures. In *Proceedings of the 18th annual international symposium on Computer architecture (ISCA)*, pages 342–351, New York, NY, USA, 1991. ACM.
- [54] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for SMT processors. Technical report, Department of Computer Science and Engineering, University of Washington, 2000.
- [55] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *Proceedings of the 33th annual International symposium on Microarchitecture (MICRO)*, pages 269–280, 2000.
- [56] A. Roth and G. S. Sohi. Speculative data-driven multithreading. *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.
- [57] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural support for enhanced SMT job scheduling. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 63–73, Washington, DC, USA, 2004.
- [58] J. Sharkey and D. Ponomarev. An l2-miss-driven early register deallocation for SMT processors. In *Proceedings of the 21st annual international conference on Supercomputing (ICS)*, pages 138–147, New York, NY, USA, 2007.
- [59] T. Sherwood and B. Calder. Loop termination prediction. In *Proceedings of the 3rd International Symposium on High Performance Computing (ISHPC)*, pages 73–87, London, UK, 2000. Springer-Verlag.
- [60] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 3–14, Washington, DC, USA, 2001.

- [61] P. Shivakumar, , and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. Technical report, Research Report 2001/2., Western Research Laboratory, 2001.
- [62] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO)*, pages 259–271, 1998.
- [63] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. *ACM SIGARCH Computer Architecture News (CAN)*, 28(5):234–244, 2000.
- [64] SPEC. Standard performance evaluation corporation (SPEC) 2000 benchmark suite. <http://www.spec.org/cpu2000/>.
- [65] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 107–119, New York, NY, USA, 2004.
- [66] Sun Microsystems. *UltraSPARC IV Processor Architecture Overview*, 2004.
- [67] Sun Microsystems. *OpenSPARCTM T1 Microarchitecture Specification*, 2006.
- [68] Sun Microsystems. *OpenSPARCTM T2 Microarchitecture Specification*, 2007.
- [69] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. *Technical Report HPL-2006-86, HP Labs*, 2006.
- [70] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.
- [71] M. Tremblay and S. Chaudhry. Third-generation 65nm 16-core 32-thread plus 32-scout thread CMT SPARC processor. In *International Solid-State Circuits Conference (ISSCC)*, pages 82–83, February 2008.
- [72] J. Tubella and A. Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of 4th International Symposium on High-Performance Computer Architecture, (HPCA)*, pages 14–23, Las Vegas, Nevada, USA, 1998.
- [73] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of International Annual Computer Measurement Group Conference*, pages 819–828, 1996.
- [74] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, Washington, DC, USA, 2001.
- [75] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading

- processor. In *Proceedings of 23rd International Symposium on Computer Architecture (ISCA)*, NY, USA, 1996.
- [76] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of 22nd International Symposium on Computer Architecture (ISCA)*, New York, NY, USA, 1995.
- [77] E. Tune, R. Kumar, D. M. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, pages 183–194, 2004.
- [78] T. Ungerer and U. Sigmund. Evaluating a multithreaded superscalar microprocessor versus a multiprocessor chip. In *Parallel Systems and Algorithms, World Scientific Publication (PASA Workshop)*, pages 147–159, 1996.
- [79] 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. In *Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2006.
- [80] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proceedings of 25th International Symposium on Computer Architecture (ISCA)*, pages 238–249, Washington, DC, USA, 1998.
- [81] C. Webb. Subroutine call/return stack. *IBM Technical Disclosure Bulletins*, 30(11):221–225, 1998.
- [82] O. Wechsler. Inside Intel Core microarchitecture. Technical report, Mobility Microprocessor Architecture, Intel Corporation, 2006.
- [83] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News (CAN)*, 23(1):20–24, 1995.
- [84] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multi-streaming. In *Proceedings of the 3rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 49–58, Manchester, UK, 1995.
- [85] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th annual international symposium on Computer Architecture (ISCA)*, pages 2–13, New York, NY, USA, 2001.