MITOSIS BASED SPECULATIVE MULTITHREADED ARCHITECTURES

Carlos Madriles Gimeno

Advisors:

Josep Maria Codina

Pedro Marcuello

Antonio González

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona (Spain), 2012

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor en Informàtica

[Roy:]

I've seen things, you people wouldn't believe, hmm.

... attack ships on fire off the shoulder of Orion.

I've watched C Beams glitter in the dark near the Tannhauser Gate.

All those moments, will be lost in time like tears in rain ...

[... time to die ...]

Blade Runner, 1982.

Abstract

In the last decade, industry made a right-hand turn and shifted towards multi-core processor designs, also known as Chip-Multi-Processors (CMPs), in order to provide further performance improvements under a reasonable power budget, design complexity, and validation cost. Over the years, several processor vendors have come out with multi-core chips in their product lines and they have become mainstream, with the number of cores increasing in each processor generation. Multi-core processors improve the performance of applications by exploiting Thread Level Parallelism (TLP) while the Instruction Level Parallelism (ILP) exploited by each individual core is limited. These architectures are very efficient when multiple threads are available for execution. However, single-thread sections of code (single-thread applications and serial sections of parallel applications) pose important constraints on the benefits achieved by parallel execution, as pointed out by Amdahl's law.

Parallel programming, even with the help of recently proposed techniques like transactional memory, has proven to be a very challenging task. On the other hand, automatically partitioning applications into threads may be a straightforward task in regular applications, but becomes much harder for irregular programs, where compilers usually fail to discover sufficient TLP. In this scenario, two main directions have been followed in the research community to take benefit of multi-core platforms: Speculative Multithreading (SpMT) and Non-Speculative Clustered architectures. The former splits a sequential application into speculative threads, while the later partitions the instructions among the cores based on data-dependences but avoid large degree of speculation. Despite the large amount of research on both these approaches, the proposed techniques so far have shown marginal performance improvements.

In this thesis we propose novel schemes to speed-up sequential or lightly threaded applications in multi-core processors that effectively address the main unresolved challenges of previous approaches. In particular, we propose a SpMT architecture, called Mitosis, that leverages a powerful software value prediction technique to manage inter-thread dependences, based on pre-computation slices (p-slices). Thanks to the accuracy and low cost of this technique, Mitosis is able to effectively parallelize applications even in the presence of frequent dependences among threads. We also propose a novel architecture, called Anaphase, that combines the best of SpMT schemes and clustered architectures. Anaphase effectively exploits ILP, TLP and Memory Level Parallelism (MLP), thanks to its unique fine-grain thread decomposition algorithm that adapts to the available parallelism in the application.

Acknowledgements

First of all, I would like to thank my advisor Antonio González. Antonio has always been an inspiring researcher, teacher, and manager for me during all these years, first at UPC and then at Intel Labs Barcelona. I have been blessed with having two more advisors: Pedro Marcuello and Josep Maria Codina, who proved equally indispensable during this period. Their guidance and support cannot be described in few words. This thesis and all the work at Intel Labs would not have been possible without their unconditional support and friendship.

I would also like to thank my friends and colleagues with whom I share or shared work and life at Intel Labs. First, for being my initial colleagues and even mentors when I started working at Intel, I want to specially thank Suso and Pepe. I learned a lot from both of them and I have very good memories of our initial steps at the lab… we really won and had fun! I also want to thank Ronny Ronen for believing in me although my knowledge of caches didn't proved to be very solid. Second, I would like to thank all the team members that have contributed to make this thesis a reality. I have always been very lucky to work in teams of extraordinary people. The Mitosis project would not have been possible without the effort and dedication of Suso, Pedro, Peter Rundberg, and Carlitos. Together we suffered and overcame all the difficulties of a long bumpy path. My special thanks go to Hong Wang, Perry Wang, and Dean Tullsen for many fruitful discussions and inestimable help. Anaphase is also the result of many people effort: Fernando, Josep Maria, Pedro Lopez, Enric, Raul, and Alejandro. Also thanks to Rakesh that contributed with many questions and valuable comments. The final success of the project is the result of their huge effort, perseverance, and passion. I can only say: Thanks, we did it!

With mixed feelings I have to thank Pedro, Josep Maria, Fernando, Llorenç, and later Ramon for the innovative methods they employed for helping me writing this thesis. This might be the most expensive thesis ever but I have to say that was fun and finally worth it.

I would like to express my deepest gratitude to my family, without their commitment and support I would not have been here. To all of them thanks for helping me grow and feel so loved. In special my brother, with whom I have shared and lived many good and bad moments, and that has always been my best friend.

Last, but not least, I would like to thank my girlfriend, Cris, for her patience, comprehension, and support during the large period that took me writing this thesis. She is my main source of inspiration and the ultimate reason for now be writing these words.

Index

Chapter 1

Introduction

Multi-core platforms have become mainstream, and the number of cores is increasing in each processor generation. These architectures are very efficient when multiple threads are available for execution. However, single-thread performance remains of paramount importance since some applications have limited thread-level parallelism (TLP), and even a small part with limited TLP imposes important constraints to the global performance, as explained by Amdahl's law.

This chapter motivates the important challenges on this scenario and introduces the solutions proposed by this thesis to address them. In addition, it describes the extensive prior research conducted in multi-core design and new execution paradigms to boost sequential and ligthly threaded applications in these platforms, and it finally presents and enumerates the main objectives and contributions of this thesis.

1.1 Motivation

The continuous advancement in process technology has allowed the integration of more and faster transistor devices into each new processor generation. As Moore's law [22] states, the number of transistors that can be integrated in the chip roughly doubles every 18 months. Traditionally, computer architects have taken benefit of this increase in transistors budget to design larger and more complex hardware structures with the objective of improving performance. More precisely, during the last decades, two main sources have been exploited to improve processor performance: (i) technology scaling, which allowed a constant increase in clock frequency, and (ii) architecture advancements on superscalar processor designs, which allowed taking more and more benefit of *Instruction Level Parallelism* (ILP).

However, performance improvements over time have come at the price of increasing power consumption and design complexity. This path hit the roadblock circa early 2000, when power consumption, and the associated cost of thermal solutions to dissipate it, reached prohibitive quotas. This motivated to stop increasing clock frequency in order to keep power consumption under reasonable values. Moreover, at that time the exploited ILP by superscalar processors reached a point of diminishing returns due to the increasing difficulty of extracting more parallelism. The increase in design complexity, and the associated area, design, and validation costs, did not pay off the returns in performance.

Given this scenario, industry made a right-hand turn and shifted towards multi-core designs, also known as Chip-Multi-Processors (CMPs) [71] (i.e. integrate multiple processor cores in a chip), in order to provide further performance improvements under a reasonable power budget, design complexity, and validation cost. Over the years, several processor vendors have come out with multi-core chips in their product lines [39][42][49][67] and they have become mainstream, with the number of cores increasing in each processor generation. In fact, using an extrapolation of Moore's law, the number of cores roughly doubles every 18 months [1].

Current multi-core processors strongly rely on the software for taking full advantage of the parallel hardware. On these processors, performance gains come from executing multiple threads (i.e. instruction streams) in parallel, exploiting *Thread Level Parallelism* (TLP). This introduces important implications in the whole software stack (i.e. applications, programming model, compilers, and operating systems) in order to be able to feed the multi-core system with enough threads. For applications which are inherently parallel or in multi-programmed workloads (i.e. server workloads, multimedia, and gaming software), finding multiple threads available for execution is not an issue. Unfortunately, this is not the case for many other scenarios, where the sequential characteristics of many applications, the difficulty of parallel

programming, and current ineffective parallelization techniques, may limit the exploitable parallelism, and thus the overall performance achieved by multi-core systems.

For instance, there are many applications which are written to run on single-core platforms, such as legacy applications, as well as many current programs whose algorithms are either sequential or whose parallel version performs poorer than the sequential one. These applications nowadays do not benefit from having multiple cores; in fact, if cores are made simpler to accommodate a larger number of them in the chip, they may even be penalized.

In addition, for those applications that its parallel version is beneficial, programmers have to deal with the burdens of parallel programming. Parallel programming, even with the help of recently proposed techniques like transactional memory [66], has proven to be a very challenging task. An alternative approach to parallel programming is provided by parallelizing compilers that automatically decompose applications into threads. However, this may be a relatively straightforward task in regular applications but becomes much harder for irregular programs, where parallelizing compilers usually fail to discover sufficient TLP.

Moreover, even in the case of applications that have been successfully parallelized, many of them are lightly threaded, have limited TLP, and present serial regions that are inherently non parallel. As Amdahl's law dictates, even a small serial portion of a parallel application poses important constraints to the scalability and global performance of the application. Figure 1 shows the scalability of a parallel application, the performance speed-up obtained for running it with more cores compared to running it in a single core, when the ratio of sequential code (s) in the application ranges from 0 (totally parallel) to 0.5 (half parallel / half sequential). As can be seen the scalability is very poor even for the case where there is just 10% of sequential code and it is much worse if the parallelization overhead, like the time to create or destroy a thread, is taken into account.

As a result, coming up with schemes to improve the performance of multi-core systems on these common scenarios is of paramount importance. Over the last decade, there have been many research efforts devoted to address the challenge of speeding-up sequential or lightly threaded applications on multi-cores platforms. The vast majority of these works involve multiple cores on the execution of the original sequential code, thus effectively parallelizing the code. Depending on the scope where this parallelization is exploited we can categorize the works into two approaches: (i) ILP oriented, and (ii) TLP oriented.

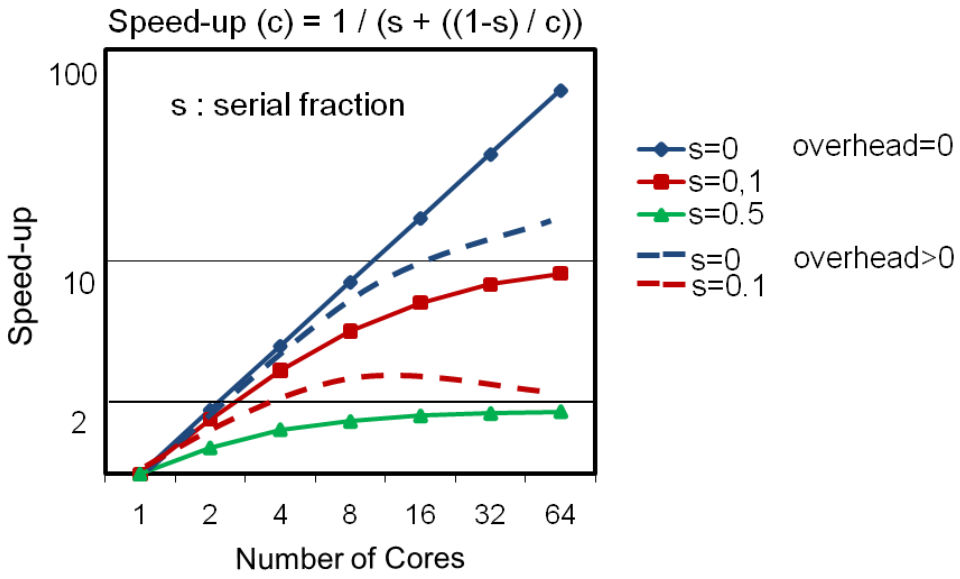


Figure 1. Parallelization performance according to Amdahl's law

ILP oriented approaches draw on ideas from the early proposals for clustered architectures. In these approaches, the dynamic instruction stream of a single-thread application is dynamically decomposed into several cores that run in cooperative mode, virtually creating a fused core [36]. The larger amount of resources effectively boosts the execution of the single-thread application but the parallelism that can be exploited is constrained within the instruction window of the fused core.

On the other hand, in TLP oriented approaches like *Speculative Multithreading* (SpMT) [3][63][89][96][106] the exploitable parallelism is not constrained to a small window. SpMT approaches decompose an application into several speculative threads using the help of hardware, software, or both. The resulting threads can run in parallel in multiple cores even in the presence of unknown or infrequent dependences among them. If the speculation turns out correct, additional performance is obtained, otherwise the hardware provides mechanisms to discard the work done under the misspecualtion and restore the prior correct state.

Despite the large amount of research on Speculative Multithreading, the proposed techniques so far have shown marginal performance improvements. This is mainly due to unresolved fundamental challenges, like dealing with inter-thread dependences and thread decomposition, which may constrain the exploitable parallelism and the overall performance gains. Therefore, new approaches to address these fundamental challenges are required.

This thesis proposes novel schemes to speed-up sequential or lightly threaded applications in current and future multi-core processors that effectively address the aforementioned challenges. The proposed schemes rely on both speculative multithreading techniques and the concept of fine-grain decomposition of ILP oriented approaches. In particular, two schemes are proposed: the Mitosis architecture [30][54] and the Anaphase architecture [55][56]. In a nutshell, the main novelties of these schemes are:

- On the Mitosis architecture, we propose a powerful software value prediction technique to manage inter-thread dependences, based on pre-computation slices (*p-slices*). This technique accurately predicts inter-thread dependences with low overhead. In addition, we propose a general thread decomposition scheme at compile time that is able to identify the most effective points in the code to generate speculative threads.
- On the Anaphase architecture, we propose a fine-grain thread decomposition scheme that at compile time decomposes an application into speculative threads at instruction granularity. Anaphase combines the best of pure Speculative Multithreading schemes, like Mitosis, and the concept of fusing cores. It effectively exploits ILP, TLP and *Memory Level Parallelism* (MLP), thanks to its unique decomposition algorithm.

As it will be presented in this thesis, the proposed schemes effectively speed-up sequential applications on multi-core systems. They implement a cost-effective hardware support on top of conventional multi-core processors that incurs on very little hardware complexity. Moreover, thanks to their unique techniques for dealing with inter-thread dependences and thread decomposition, they achieve significant performance improvements on irregular applications for which traditional SpMT schemes fail.

This chapter is organized as follows. In Section 1.1 we have presented the scenario, the challenges that this thesis envisions solving, and we have briefly summarized our contributions. In Section 1.2 we discuss the different designs for multi-core systems proposed by both industry and academia, which tradeoff TLP and ILP exploitation. In Section 1.3 we describe in detail the main research efforts that have been taken in speeding-up sequential applications on multi-core platforms; and their various pros and cons. In Section 1.4 we enumerate the main objectives and contributions of this thesis, and finally we describe the organization of this document in Section 1.5.

1.2 Multi-core Systems

Chip-Multi-Processors (CMPs) integrate multiple cores on-die to execute parallel, or multi-programmed, workloads. *Thread Level Parallelism* (TLP) is exploited by executing multiple threads in multiple cores, whereas *Instruction Level Parallelism* (ILP) is exploited inside each core.

The design of a CMP able to efficiently exploit TLP and ILP is not straight-forward. The size and type of cores integrated in the chip have important implications on the amount of TLP and ILP that can be exploited. Industry and academia have explored different alternative in the design spectrum of CMPs, moving into two directions: (i) complexity and number of cores, and (ii) homogeneity / heterogeneity of cores.

1.2.1 Complexity and Number of Cores

Multi-core architectures based on a large number of simple cores are very effective to exploit TLP but their performance is compromised for lightly threaded applications. Each core is usually an in-order, small issue width processor. These multi-core designs are particularly interesting for running workloads where the overall throughput is more important than the performance of each individual thread. Typically, this is the case for server and High-Performance-Computing (HPC) workloads, where due to the large amount of available threads it is more beneficial to execute as many as possible in parallel than to execute faster a few of them. Examples of CMPs introduced by different vendors that fit into this category are Sun Piranha [6], Sun Niagara [42], and the prototype Intel® Polaris [99].

By contrast, multi-core architectures based on big and complex cores, like IBM Power6 [49] and Intel® Core™ Duo [67], have fewer of them because of area and power constraints (notice that integrating a large number of complex cores into the same chip is infeasible due to these reasons). They have limited TLP capabilities but are more effective to exploit ILP. Usually, each core is an out-of-order (OoO), wide issue, superscalar processor. These systems are better suited for the general-purpose segment, where single-thread performance is very important but multi-programmed environments and multi-threaded applications, like multimedia and games, are demanding more TLP capabilities to gain additional performance. On this scenario, academia and later processor vendors have adopt the idea of using the otherwise idle resources of big cores for simultaneously executing multiple threads. Simultaneous Multithreading (SMT) [97] and Hyperthreading [64] are examples of this technique.

This thesis focuses on the second type of CMPs because single-thread performance is one of their key selling points. However, for the sake of completeness, many experiments presented in this thesis have been performed with different CMP designs.

1.2.2 Homogeneity / Heterogeneity of Cores

An alternative characteristic of a CMP design is whether the cores integrated in the chip are exact replicas of each other, or are of different configurations or even of different types. Following this design axis, multi-core systems are classified as homogenous or heterogeneous, respectively.

Examples of CMPs introduced by different vendors with a homogenous design are Intel® Core™ Duo [67], Sun Niagara [42], and IBM Power6 [49]. On the other side, there have been less heterogeneous designs introduced in the market. One example of a successful CMP with a heterogeneous design has been the IBM Cell [39].

Heterogeneous multi-cores consist of cores of various sizes and complexity on the same die; from small and simple cores to big and complex cores. There are two main flavors of heterogeneous designs: (i) cores with the same ISA [46], and (ii) one or multiple main cores, coupled with multiple accelerators (with different ISA) [39].

A big benefit of heterogeneous CMPs is the fact that they are able to adapt to workloads that have different execution requirements and behaviors. For instance, for the second flavor of heterogeneous designs, due to the presence of specific accelerators they may be very good at multimedia and gaming workloads. In some designs, they may also be well suited to execute single-thread applications because the processor may incorporate a big and complex core that would perform better than the one present in a homogenous CMP.

However, heterogeneous CMPs are very complex to design and validate. Moreover, they place serious implications on the software side, specifically on the Operating System and the parallel programming model. This is due to the fact that the heterogeneity of the underlying hardware is often exposed to the software with the objective of maximizing the utilization of the processor and the overall performance of the system. This introduces more complexity on the whole software stack. At the end of the day, all these handicaps make them difficult to success in the general-purpose market.

Homogenous CMPs have become the first choice of the main processor vendors and it is likely that future CMP systems will keep following this model. Homogenous CMPs are easier to design and validate than heterogeneous ones. As opposed to heterogeneous systems, they are more suitable for workloads

composed of threads that have similar execution requirements and behavior. In addition, these kinds of systems are much easier to program than heterogeneous ones because they do not require changes in the Operating System or specific parallel programming models.

On the down side, homogenous CMPs are not able to adapt to threads that have different demands on terms of execution resources. Moreover, as previously pointed out in Section 1.1, they do not work very well for executing single-thread applications. In the presence of a single thread, a lot of execution resources available in the processor are idle and do not contribute to the thread execution.

Many research efforts, which will be discussed in the following section, have come out with ideas to use these additional execution resources to boost single-thread performance. This thesis, sum-up to these works and proposes different techniques to speed-up single and lightly threaded applications in a homogenous CMP system.

1.3 Boosting Sequential Execution on Multi-cores

As has been pointed out in previous sections, sequential and lightly threaded applications represent very common workloads in today's general-purpose segment, and very likely in future's as well. Over the years, a lot of research effort has been put into speeding-up these workloads in multi-core processors. As a result, researchers have come up with a broad scope of techniques to boost sequential execution on multi-core processors.

First set of techniques are under the umbrella of *Automatic Parallelization* [10]. They rely on the compiler to automatically parallelize a sequential application. These techniques have proved to be successful for regular numerical applications, but far from optimal for non-numerical, irregular applications. In these later applications the compiler usually fails to discover a significant amount of thread-level parallelism.

In order to overcome the constraints of conservative *Automatic Parallelization*, *Thread-Level Speculation (TLS)* techniques, also known as *Speculative Multithreading (SpMT)*, attempt to speed-up the execution of applications through speculative thread-level parallelism. Threads are speculative in the sense that they may be data and control dependent on previous threads (that have not completed) and therefore their execution may be wrong and may require re-execution. There are two main strategies to exploit speculative thread-level parallelism:

1) Parallelize applications into speculative parallel threads [89], each of which contributes by executing a part of the original application.

2) Use helper threads [19] to reduce the execution time of high-latency instructions/events through beneficial side effects.

Finally, another set of techniques that could be classified as *Adaptive CMP Architectures*, build on the idea of designing a CMP that try to adapt the resources of the chip to the available ILP in the running application. A particular incarnation of these techniques is the idea of combining multiple cores working in cooperative mode to execute a single thread, virtually creating a bigger fused core [36].

In the following subsections we describe all these approaches in more detail.

1.3.1 Automatic Parallelization

Parallel programming is hard and error-prone. The programmer has to parallelize the work, perform the data placement, deal with thread synchronization and, of course, debug the applications (the non-determinism induced by the different activity on different cores makes this particularly hard). Although there has been extensive work on making synchronization easier or less painful, like Transactional Memory techniques [34][47] and Speculative Lock Elision [79], and on allowing deterministic replays for debugging purposes [35][68], devising parallel algorithms and dealing with data placement is still cumbersome.

A convenient alternative to parallel programming is offered by parallelizing compilers [1][7][10][51]. Parallelizing compilers are given sequential applications, which they try to parallelize by deducing that specific program segments do not contain any data dependences. Despite many years of research, parallelizing compilers still fail to parallelize all but the most trivial and regular applications. In fact as pointed out in [95] Intel®'s state-of-the-art compiler (*icc*) actually gets a slowdown for most of the irregular applications it tries to parallelize. The main reason for this is that either the compiler cannot *statically* guarantee that specific code segments do not have any dependences (e.g., memory disambiguation in the presence of pointers is challenging) or they cannot find large enough sections where this holds so that their coverage (i.e. dynamic portion of the application that benefits from the technique) is large enough to produce benefits.

1.3.2 Speculative Multithreading

A step beyond conventional automatic parallelization techniques is provided by systems that support Thread-Level Speculation (TLS), also known as Speculative Multithreading (SpMT). The main feature of these systems is that threads are generated without having to consider all possible cross-thread data

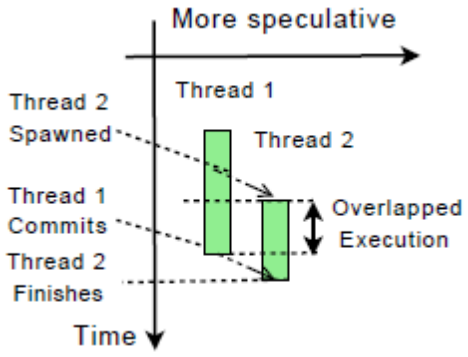


Figure 2. Speculative Multithreading Execution Model

dependences. The resulting threads are speculative in the sense that may be data and control dependent on other threads, and their correct execution and commit are not guaranteed. Therefore, in this model, we can optimistically parallelize sequential sections of code, because we suspect that they may be independent (or predictably dependent), and then we provide mechanisms for checking whether that was true after the fact and re-execute the code if needed. This model effectively breaks the constraint of conventional parallelizing techniques and is able to expose much more thread-level parallelism.

SpMT systems as other parallelization techniques need to guarantee the original sequential semantics of the program that has been speculatively parallelized. Since unattended execution of speculative threads may violate the sequential semantics of the original program, the data dependencies and the order among the threads need to be verified at runtime before the threads can update the architectural state. In order to do that, SpMT architectures need mechanisms to buffer the speculative state of the threads, and ways to verify the correctness of the speculation. In case of misspeculation, support for rollback to a safe state is also necessary.

Figure 2 shows a simplified representation of a common SpMT execution model. In this model, the control flow of the original sequential application imposes a total order on the threads. At any time during execution, the earliest thread in program order is *non-speculative* while the others are *speculative*. The terms *predecessor* and *successor* are used to relate threads in this total order. A speculative thread is created when a *spawn* action (i.e. a thread light-fork operation) is executed and it starts executing in a free *thread unit* (in a multi-core processor a thread unit could be a core).

Stores from speculative threads generate unsafe *versions* of variables that are stored in some sort of *speculative buffer*. If a speculative thread overflows its speculative buffer it must stall and wait to become non-speculative. Loads from speculative threads are provided with potentially incorrect versions. As

execution proceeds, the system tracks memory references to identify any inter-thread data dependence violation. For instance, a store from a thread to a memory location previously loaded by a more speculative thread would produce a dependence violation. If a dependence violation is found, the offending thread must be *squashed* (i.e. kill the thread and flush the *thread unit* where it is running), along with its successors, thus reverting the state back to a safe position from which threads can be re-executed.

Threads commit in sequential order. This means that a thread must wait to become the oldest one (i.e., the non-speculative thread) to commit. When the non-speculative thread reaches the starting point of the following speculative thread, this speculative thread becomes non-speculative and is allowed to *commit*. When it commits the values it generated can be moved to architectural storage (usually main memory or some shared higher-level cache). After committing, the previous non-speculative thread dies and its thread unit is free to start executing a new speculative thread.

During the last decade Speculative Multithreading has generated a lot of interest in the research community. Many works have been published covering a broad scope of the design space. Even though there have been very significant breakthroughs on this area, there are still many challenges that need to be addressed to make SpMT systems competitive. This thesis proposes solutions to these challenges (see Section 1.4). We can summarize the SpMT design space along following four dimensions:

1) *Thread Partitioning*: What type of threads is extracted from the sequential application.
2) *Inter-thread Dependences Management*: What techniques are employed to handle data dependences among threads.
3) *Hardware vs. Software*: Whether the SpMT system is implemented only in hardware, only in software, or in combination of hardware and software.
4) *Execution Model Support*: What kind of support is provided by the SpMT system to assist in execution model tasks like: speculative state buffering, inter-thread dependence violation detection, violation recovery, etc…

In this section, we discuss the most relevant prior art in Speculative Multithreading, following these four dimensions, and we point out the main design decisions adopted in this thesis.

1.3.2.1 Thread Partitioning

One of the key design decisions for a SpMT system is what type of speculative threads is extracted from a sequential application. In other words, how sequential code is partitioned into speculative threads. As one

can imagine, the choice of a good thread partitioning scheme is critical for the overall performance of the speculatively parallelize application.

Several criterions are traditionally followed to partition an application into speculative threads: (i) number of data dependences among the threads, (ii) workload balance, (iii) size of the threads, and (iv) parallel coverage.

The *number of data dependences* among the threads has a direct impact on the amount of overhead that is required to manage them (i.e. through synchronization or value prediction; see Section 1.3.2.2 for a detailed discussion), and thus on the time the code is effectively running in parallel. In addition, a higher number of inter-thread dependences may increase the probability of misspeculating due to a data or control dependence violation. On the other hand, w*orkload balance* refers to the desirable property of all the threads doing the same amount of work in order to have similar execution time. Otherwise, resources could be wasted due to idle thread units. *Thread size*, is another important factor to take into account when decomposing an application into speculative threads. Very small threads may result on no performance gains or even slowdown due to very little work done in parallel compared to the overhead of spawning and committing threads. At the same time, large threads may overrun the speculative state buffer, which may imply stalling the threads and loosing performance. Finally, it is desirable to have a thread partitioning scheme that ensures a good *coverage* on the dynamic speculatively parallelized code. As previously shown in Figure 1, just a small portion of sequential code may significantly limit the overall performance of the parallelized application.

Several studies propose schemes to extract speculative threads from well-known program constructs such as loops [16][23][33][53][60][90], sub-routines [5], and modules [101] (procedures, functions, methods, etc.). The rationale behind this choice is that typically loop or sub-routines favor several of the mentioned criterions. Loop iterations lend themselves well to the workload balance as usually loop iterations carry out similar amount of computation and hence are similar in execution time. In addition, they usually represent the dominant part of the dynamic execution stream for most of the applications, which ensures a good parallel coverage for many applications. On the other hand, spawning speculative threads at sub-routines continuation could be more suitable than loop iterations in terms of number of inter-thread dependences. The computation in the sub-routine and after the sub-routine have usually very less dependences (i.e. mainly the return value of the sub-routine), but are not very workload balanced by nature and may result on lower parallel coverage than loops.

An alternative approach investigated by some researchers is to not partition based on any program constructs but directly analyzing the control-flow graph and data-dependence graph of a piece of code, looking for threads that satisfy all the mentioned criterions. Pioneer works in the area of SpMT, like the Expandable Split Window Paradigm [27] and the follow-up works, the Multiscalar processor [89][100], and the Min-Cut decomposition algorithm [37] use this approach. A more complex scheme, proposed by Marcuello [58], that follows the same approach is based on profiling information and exploits the idea of high control independence between basic blocks. Any combination of basic blocks is considered a candidate spawning pair, which defines a speculative thread. Those spawning pairs that have high control independence and satisfy the aforementioned criterions are selected. A different approach is taken by the MEM-slicing algorithm [20], which instead of spawning threads at points of the program with high control independence, spawns threads at memory access instructions.

Another important aspect is the granularity at which partition in threads happen. Traditional Speculative Multithreading schemes, like the ones mentioned in this section [5][16][37][58][98][106], decompose sequential codes into large chunks of consecutive instructions. In the case of loops, most of the previous schemes spread the iterations into different threads, i.e. one or more loop iterations are included in the same thread. A limited number of works consider the decomposition of the code inside the iterations by assigning several basic blocks into the same thread, while others assign strongly connected components to different threads. In the case of sub-routines, all previous works considering the decomposition at function calls shred the code in such a way that consecutive functions are dynamically executed in parallel. Finally, all works considering general control flows shred the whole application in such a way that consecutive chunks of basic blocks are assigned to different threads for its parallel execution. All these partitioning schemes can be defined as coarse-grain.

Such coarse-grain decomposition may constraint the benefits of the SpMT paradigm. This is particularly true when facing hard to parallelize codes, where coarse grain decomposition may introduce too many dependencies among threads. This may end up limiting the parallelism in this codes and harming performance. Instead, parallelizing applications at instruction granularity provides more flexibility and thus it has more potential to further exploit TLP. Ranjan (Fg-STP [81]) has recently proposed a thread partitioning scheme at instruction granularity, which can be seen as a hybrid approach between SpMT and an Adaptive CMP Architecture (see Section 1.3.4).

In this thesis we sustain that a thread partitioning model not constrained to program structures (loops, routines, modules, …), like the one based on high control independence [58], is crucial for

achieving good parallel coverage in many irregular applications. On the other hand, many hard to parallelize codes require fine-grain thread decomposition techniques in order to extract speculative threads that satisfy the criterions presented in this section. The Speculative Multithreading schemes proposed in this thesis build based on these two pillars.

1.3.2.2 Inter-thread Dependences Management

Another important design decision for a SpMT system is how data dependences among speculative threads are handled. It is well known that the way inter-thread data dependences are managed strongly affects the performance of speculative multithreaded processors and its design complexity [63].

So far researchers have proposed to deal with inter-thread dependences using one or a combination of the following schemes: (i) *speculate no dependencies* [82], (ii) *explicit synchronization* [89][91][96][103], and (iii) *value prediction* [3][62][72][106].

The first scheme leverages the main characteristic of SpMT systems: speculating on inter-thread data dependences. It assumes that speculative threads do not have any dependence among them and relies on the support already present in any SpMT system for detecting and recovering from a data dependence violation. In that case, no special mechanism is needed to manage inter-thread dependences.

Although the great advantage of SpMT is that synchronization between threads can be avoided, there are some dependencies that speculating on them may introduce too many misspeculations. One possible way to deal with these dependences is through explicit synchronization. Often data dependences causing squashes at execution time can be statically determined possibly with the use of profiling. This observation is exploited in [103] by introducing synchronization instructions when dependences can be identified at compile time. Misspeculation is avoided by inserting wait and signal primitives, forcing uses to wait until the producing instruction in the predecessor thread has executed. Similar schemes are used in other compiler-based approaches like the Multiscalar [89] and the Superthreaded [96]. These works propose some techniques to reduce the synchronization latency like identifying the last writer, or reordering the code to compute the dependent value earlier.

Managing inter-thread dependences with explicit synchronization is relatively cheap in terms of design complexity but synchronizing too many dependencies may constrain the exploitable parallelism and degrade performance. This is especially true for many sequential applications that are highly control intensive and have many memory pointer accesses. In that case, detecting the right dependences to synchronize can be very challenging and synchronizing all the dependences is a too conservative approach that may limit the parallelization benefits.

A powerful alternative investigated by many researchers is value prediction. Value prediction of dependences can significantly increase parallelism, especially for register values that are quite predictable [62]. Note however that not all dependences are easy to predict, in particular memory values are harder to predict [17][91]. In addition, speculative multithreading architectures are particularly sensitive to value prediction accuracy, because multiple values must typically be predicted correctly for the thread to be useful. We can broadly categorize value prediction schemes into two categories: *Hardware Value Prediction* and *Software Value Prediction*.

Hardware Value Prediction techniques [52][62][88] use hardware structures to capture a variety of run time information like value patterns and context information, in order to predict the dependent values. While hardware value prediction have shown higher accuracy in exploiting ILP in programs, their accuracy in the context of SpMT is limited to very small loops, usually inner loops [62]. The accuracy of these predictors is highly reliant on the context information, so as the thread size grows their accuracy usually decreases.

On the other hand, *Software Value Prediction* has the advantage that no big hardware structures are required for doing the prediction and that complex patterns can be captured, at expenses of inserting additional instructions in the code. Although it lacks dynamic information, which is usually gathered using profiling anyway, the analysis done at compile time allows implementing more complex schemes that can use high-level information and from a broader scope. The POSH [53] compiler is an example of compiler that uses software value prediction to predict inter-thread data dependences. Its main drawback is that it is limited to induction variables. A more generic scheme that uses additional compiler support for doing software value prediction is proposed in [50]. In this scheme, any detected inter-thread dependence at compile time is candidate for value prediction. The compiler helps to determine critical and predictable values, using selective value profiling, and insert the minimal code to predict them.

A different approach to predict the input values of speculative threads, based on software value prediction, is proposed in Master/Slave Speculative Parallelization (MSSP) [106]. MSSP defines a distilled program, a subset of the instructions of the parallelized program, to compute/predict the inputs of all the speculative threads. The prediction is potentially more accurate since the computation of these values is directly derived from the original code. Additionally, it can encapsulate multiple control flows that contribute to the computation of the input values. Note that some hardware value predictors can also do that [62], but they require additional hardware to predict the control flow of the parent thread. The

main drawback of this scheme is the large amount of additional activity, or overhead, that is required to predict the input values of the threads, which may significantly increase the overall energy of the system.

Predicting inter-thread dependences with high accuracy and low overhead is very critical to the overall performance and efficiency of a SpMT system. In this thesis, we investigate on models that use novel software value prediction techniques to deal with inter-thread dependences, we propose several optimizations to reduce the prediction overhead as well as mechanisms to combine and select the most appropriate scheme to manage each of the dependences.

1.3.2.3 Hardware vs. Software

Based on the amount of hardware and software involved on the execution model of a SpMT system, we can classify them in: (i) *hardware only* approaches, (ii) *software only* approaches, and (iii) *hybrid* approaches.

Hardware only approaches have the advantage that can leverage dynamic information and can exploit speculative thread level parallelism and speed-up sequential applications without having to recompile them. For some legacy applications, whose source code may not be available, this is an important value. On the other hand, these platforms incur on very high hardware complexity and suffer from low quality speculative threads. This is due to the fact that hardware has a limited view of the dynamic instruction stream, usually constrained to the window of in-flight instructions, which impacts on the quality of the threads the hardware can extract. Examples of SpMT systems that implement a purely hardware based scheme are: the Dynamic Multithreaded Processor [3], the Implicitly-Multithreaded Processor [77], and the Clustered Speculative Multithreaded Processor [60].

On the other hand, *software only* approaches do not add hardware complexity to SpMT systems and benefit from higher quality speculative threads due to much better thread partitioning schemes, at expenses of having to recompile the programs. The fact that all the analysis is done at compilation time, and hence lacks the runtime characteristics of the program, introduces some challenges. Even though dynamic information (i.e. control path, data dependences, data values, …) is usually gathered using profiling, software only systems are not necessarily very good at adapting to the runtime change in the program behavior. However, the main drawback of this approach is that provides low performance due to high software overheads in terms of additional memory space and computation activity to buffer speculative state and track misspeculations. Examples of this kind of systems where the software is entirely responsible for carrying out all the SpMT execution model tasks on stock multi-core processors are: S-TLS [85], LRPD Test [82], SpLIP [69], and Copy or Discard [94].

It is important to notice that both software and hardware are particularly good at doing certain SpMT tasks and complement each other. As mentioned, to extract good quality speculative threads software is definitely better placed than hardware since it has the picture of the whole program and can afford complex thread partitioning schemes. Whereas, the hardware can buffer speculative state and track dependence violations with much less overhead compared to software methods. This insight motivates the software/hardware co-designed approach taken by many of the SpMT proposals in the research literature. Examples of these SpMT *hybrid* systems where thread partitioning is performed by the software (compiler) and the hardware is responsible for the rest of tasks are: Multiscalar [89], Superthreaded [96], SPSM [24], MSSP [106], I-ACOMA [44], STAMPede [90], Hydra [33], Atlas [20], and Pinot [70].

We sustain that the hybrid software/hardware approach is more suitable for SpMT given the benefits in terms of performance and hardware complexity of such a design. Having a SpMT system with a powerful thread partitioning scheme and efficient hardware support is crucial for its success. Due to the aforementioned reasons, in this thesis all the systems we propose follow a hybrid software/hardware approach.

1.3.2.4 Execution Model Support

Finally, we can differentiate among SpMT systems by the kind of hardware support to assist in the execution model main tasks: speculative state buffering, inter-thread dependence violation detection, and violation recovery.

With regard to speculative state buffering, state produced by speculative threads must be carefully handled so as not to compromise the correct execution of the application. In particular, since some of the speculative threads may be incorrect, their state cannot be merged with that of the safe state of the program. Consequently, the state of each thread is stored separately and if a violation is detected the state generated by the misspeculated thread is discarded. Otherwise, at thread commit the state is allowed to propagate to the safe architectural state of the program. In the case of registers, some works propose specific register files to store the speculative register state [9][44]. In the literature, more attention has been put on different hardware designs for storing the speculative memory state. While proposals like [3] [29][33][104] introduce special buffers to store the speculative modifications, others including the Speculative Versioning Cache (SVC) [32], the Multi-Value Cache [61], and [45] use the existing local caches for the purpose. A detailed taxonomy of approaches to buffer and manage the speculative memory state on SpMT processors is presented in [31]. Moreover, some optimizations to increase the scalability of the memory subsystem of a SpMT multi-core processor are proposed in [78].

Different hardware schemes are proposed in the literature to detect inter-thread dependence violations, which vary on scalability, precision, and hardware complexity. Some proposals, like the ARB [29] and the MDT [45], define directory-like structures to track memory dependences among speculative threads. These directories are centralized, which may introduce latency and bandwidth issues that limit scalability. GMDT [16] is a distributed implementation of the MDT that addresses the scalability issue. However, the majority of works leverage well-known MSI/MESI cache coherence protocols, which are extended with additional state bits and control logic to track memory dependence violations. Examples of these schemes are the SVC [32] and STAMPede [92]. Others schemes, like the Bulk architecture [14], reduce the amount of additional state bits that are kept by using signatures and bloom filters [8] to detect the memory violations, at expenses of suffering from some false positives. Another important design parameter is the granularity detection is performed at (i.e. byte, word, or line). Smaller is the granularity of detection, better is the performance but higher is the hardware complexity.

Violation recovery is a fairly important architectural component of a SpMT system. The system should be able to restore any speculative change, so that the architectural state remains valid when data dependence violations have occurred. When a violation is detected the speculative memory and register state of the violating thread is invalidated and the thread is restarted from a safe point. Most proposals in the literature perform a complete squash of the violating thread and re-execution from its start. An alternative is to use selective-repair schemes or perform periodic checkpoints to reduce the amount of work that is thrown away on a thread squash. Re-Slice [87] is a recovery mechanism where a forward slice of the thread that misspeculates is collected and executed separately to repair the architectural state. While Re-Slice saves some of the useful work done by the violating threads, it has the extra overhead of collecting the slice. Also as the slice collection is done by the hardware, it incurs significant hardware complexity. Sub-threads [21] is a scheme to take periodic checkpoints. On a violation the thread would rollback to the nearest previous checkpoint rather than to the beginning of the thread. While this scheme is very promising for very large sized threads, it is not suitable for smaller threads due to the high overheads of checkpoints.

Some works propose hardware support for implementing SpMT on Simultaneous Multithreading (SMT) [97] architectures like [59][74]. An important challenge of implementing SpMT in SMT environments is to appropriately partitioning resources between the non-speculative thread and the speculative threads.

The SpMT schemes proposed in this thesis rely on specialized hardware to support their execution models on top of conventional multi-cores systems. For the sake of simplicity and effectiveness, we do

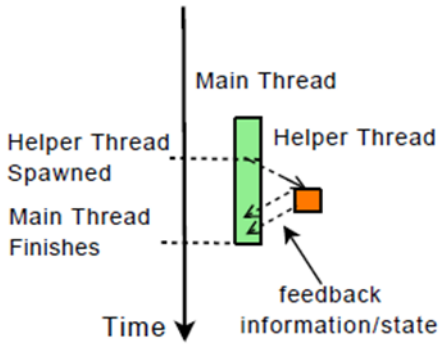


Figure 3. Helper Thread Execution Model

not consider the additional hardware support to allow running speculative threads on SMT architectures. Designing a hardware support with low overhead in terms of area, power, and performance impact, is a must. In this thesis, we explore different cost-effective microarchitecture support with the objective of achieving good performance benefits keeping hardware complexity low.

1.3.3 Helper Threads

Another possible solution to accelerate program execution in multi-core systems without resorting to parallel programs or speculative parallelization is provided by systems that support Helper Threads (HT) [15][19][84].

Under the helper threads approach, also called subordinate microthreads, small threads are run concurrently with a main thread. The purpose of the helper threads is not to directly contribute to the actual program computation, which is still performed in full by the main thread, but to facilitate the execution of the main thread indirectly. Common ways to accelerate the execution of the main thread involve initiating memory requests ahead of time (i.e., prefetching; such that the results are hopefully in the cache by the time they are needed by the main thread) and resolving branches ahead of time.

Usually, the execution of helper threads is speculative in that they may be following some incorrect control flow path and/or producing and consuming incorrect data. In this multithreaded execution model there is no particular ordering among multiple helper threads and all are discarded at the end of their execution. Figure 3 depicts this execution model.

Helper threads can be statically generated by the compiler/programmer, or dynamically generated at runtime. They often consist of slices of instructions from the main thread (e.g., only those instructions directly involved in the computation of some memory address or branch condition). Depending on the

size and complexity of the helper threads it may be possible to keep all their intermediate results in the registers, but it may be necessary to allow for spills to the memory hierarchy, which in turn requires providing storage for speculative versions of data. The compiler marks the main thread with fork-like *spawn* instructions at points where particular helper threads should be initiated [19].

The architectural support required by HT is usually much more simpler than by SpMT parallelization, and consists of three main components: i) a mechanism to allow helper threads to operate in their own context and, possibly, to enforce that speculatively modified data be also kept separate, ii) a mechanism to spawn threads in different cores or contexts, and iii) a mechanism to discard threads when finished.

Note that SpMT schemes (see Section 1.3.2), as helper threads do, may benefit from side-effects like prefetching and branch-prediction training. The SpMT schemes proposed in this thesis (see Section 1.4) are not an exception. Therefore, a comparison of the SpMT schemes proposed in this thesis with helper threads is also included (see Chapter 2 and Chapter 3).

1.3.4 Adaptive CMP Architectures

Some researchers have proposed adaptive multi-core architectures where the characteristics of the architecture dynamically adapt to the parallelism of the applications. One flavor of this approach is the idea of combining or fusing cores [36][48][107] to virtually create a larger and wider issue core with the objective of effectively exploit the instruction-level parallelism (ILP) available in the applications running in the CMP.

In a nutshell, when the application has higher ILP than can be exploited by an individual core, then multiple cores execute the instruction stream together to provide a wider instruction issue width. Whereas, when the application program has low ILP, some of the additional cores on the chip can be switched off to save power. When running a Multi-programmed workload, the cores can effectively act as independent cores.

This approach, a.k.a. Core Fusion [36], can be seen as a natural evolution of clustered microarchitectures [11][18]. In a conventional clustered microarchitecture, instructions are distributed among clusters (an execution unit along with some partitioned processor structures) following a steering mechanism. Steering policies are usually based on data dependences to minimize the data communication between clusters, but other heuristics like workload balance may also be used. Instructions can be assigned to clusters statically at compile time, or dynamically through a steering logic.

Core Fusion involves the task of dynamically distributing instructions in cores rather than clusters. Thus, starting from a single-threaded code, Core Fusion decomposes a dynamic stream of instructions into different hardware contexts. Therefore, Core Fusion exploits the concept of non-speculative multithreading execution where instructions are distributed in a fine-grain fashion. Similar to previously presented techniques, Core Fusion relies on parallelizing techniques but it is limited to the exploitation of ILP inside an instruction window of a typical size, which ultimately constrains the parallelism that can be exploited. A recently proposed scheme, called Fine-Grain Single Thread Partitioning (Fg-STP) [81], alleviates this constrain. Fg-STP is a hardware-only scheme that is able to look for parallelism on a large instruction window and to exploit fine-grain speculative thread execution using two cores.

Other research proposals of adaptive CMP architectures that try to achieve the same goal than Core Fusion are Smart Memories [57], Voltron [107], and TRIPS [86]. In Smart Memories multiple in-order RISC cores are merged to form a VLIW machine. The two configurations are not ISA-compatible, and the VLIW configuration requires specialized compiler support. Similarly, Voltron is a multi-core architecture that thanks to specialized compiler support can exploit hybrid forms of parallelism by organizing its cores as a wide VLIW machine. Finally, TRIPS is a pioneer reconfigurable computing paradigm that aims to meet the demands of a diverse set of applications by splitting ultra-large cores. TRIPS implements a custom ISA and microarchitecture, and relies heavily on compiler support for scheduling instructions to extract ILP.

It is important to remark that the concept of distributing instructions into different cores, virtually creating a fused core, inspires one of the SpMT schemes proposed by this thesis (see Section 1.4). In addition, this thesis presents a comparison of our proposal with Core Fusion (see Chapter 3).

1.4 Thesis Objectives

The main objective of this thesis is to propose novel schemes to speed-up sequential or lightly threaded applications in current and future multi-core processors. To accomplish this objective we rely on both speculative multithreading techniques and the concept of adaptive CMP architectures, which have been introduced in this chapter. These systems leverage the additional resources of multi-core processors to exploit the ILP and TLP available in these applications. Although there have been many research efforts on these areas during the last decade, the proposed techniques so far have shown marginal performance improvements.

In this thesis, we first identify the fundamental challenges of these techniques that may constrain the exploitable parallelism and the overall performance gains, and propose schemes that effectively address them. In a nutshell, main challenges that should be addressed are:

- Achieving high parallelization coverage on irregular application may be a challenge for traditional thread partitioning schemes. A general thread partitioning model not constrained to program structures (loops, routines, modules, …), like the one based on high control independence [58], is crucial for achieving good parallel coverage in many irregular applications.
- Traditional SpMT schemes partitions applications into speculative threads in chunks of consecutive instructions. This coarse-grain thread decomposition may constrain the exploitable parallelism. Many hard to parallelize codes require fine-grain thread decomposition techniques in order to extract speculative threads that adapt to the available parallelism, and have good workload balance with few inter-thread dependences.
- Predicting inter-thread dependences with high accuracy and low overhead is very critical to the overall performance and efficiency of a SpMT system. A mechanism based on software value prediction is key for achieving these objectives, as well as having in place a mechanism to flexibly select the most appropriate scheme to manage each of the dependences.
- Hardware only SpMT architectures may require very complex additional hardware support. The alternative of implementing everything in software may sacrifice too much performance. A hybrid software/hardware approach is more suitable for SpMT given the benefits in terms of performance and hardware complexity of such a design. Moreover, designing a cost-effective microarchitecture support is important for achieving good performance benefits keeping hardware complexity low.

With all these challenges and objectives in mind, this thesis proposes two novel SpMT architectures to boost single-thread performance on multi-core processors: the Mitosis architecture, and the Anaphase architecture.

The Mitosis architecture [30][54] is a hybrid software/hardware SpMT system. Its main distinguishing feature is that it is able to parallelize irregular applications and achieve good performance even in the presence of frequent dependences among threads. This is achieved thanks to: (i) a general thread partitioning scheme implemented in the Mitosis compiler that is able to place *spawning pairs* [58] in any point of the program and identify the most effective points to spawn speculative threads, and (ii) a

powerful software value prediction technique to manage inter-thread dependences. This value prediction technique is based on predicting/computing the thread input values via software through a piece of code, the pre-computation slice (*p-slice*), that is generated at compile time and added at the beginning of each thread. The accuracy of a *p-slice* is higher than other prediction schemes because it is constructed from the original program instructions. An important observation is that hardware support for recovery may allow the compiler to apply aggressive optimization techniques in order to significantly reduce the cost of *p-slices*, even sacrificing correctness for spare cases. In addition, the Mitosis hardware support contains a number of novel features for the execution and validation of *p-slices*.

The Anaphase architecture [55][56] is a novel hybrid software/hardware SpMT system that combines the best of pure Speculative Multithreading schemes, like Mitosis, and the concept of fusing cores introduced in adaptive CMP architectures like Core Fusion [36]. In contrast to traditional SpMT schemes, the main distinguishing feature of this novel scheme is that it effectively exploits ILP, TLP and Memory Level Parallelism (MLP), thanks to its unique and powerful decomposition algorithm that decomposes an application into threads at instruction granularity. The flexibility that this fine-grain decomposition provides has many benefits:

- Resulting speculative threads can adapt to the available parallelism of the application. In case of abundant ILP, Anaphase can create a partition where near instructions are interleaved among different threads. Whereas, if TLP is abundant, Anaphase can distribute instructions that are far away in the dynamic stream into different threads.
- The decomposition algorithm implements specific heuristics to: (i) minimize the dependences among the resulting threads, (ii) improve the workload balance, and (ii) increase the amount of MLP that can be exploited by taking into account how delinquent loads and their consumers are distributed among threads.

Moreover, the Anaphase architecture implements a cost-effective hardware support on top of a conventional multi-core processor that allows the execution of Anaphase threads with low hardware complexity.

1.4.1 Thesis Contributions

The main contributions of this thesis are:

1) On the Mitosis Architecture:

- A general compilation framework to analyze and partition applications into speculative threads, by inserting *spawning pairs* at any point of a program.
- A model to estimate the benefit of any set of *spawning pairs* for a given SpMT configuration, and a scheme to select the most effective set.
- A novel software value prediction technique to manage inter-thread dependences based on the use of pre-computation slices (*p-slices*) to predict the input values of speculative threads.
- A mechanism to statically build and optimize (in terms of both accuracy and overhead) pre-computation slices. In this area, several aggressive and speculative *p-slice* optimizations are proposed.
- Hardware support for executing Mitosis threads on a regular multi-core processor, with specific support for the execution and validation of *p-slices*.
- A new multi-version register file (MVRF) that manages global register state and inter-core dependences with very little latency overhead.
- Different proposals of a multi-version memory subsystem (MVC and MU[E]SLI) to manage the speculative memory state and inter-threads dependences.

2) On the Anaphase Architecture:
   - A novel SpMT paradigm that is able to effectively exploit ILP, TLP, and MLP, in which code is shred into speculative threads at instruction granularity.
   - An algorithm to perform fine-grain speculative thread decomposition that includes heuristics for minimizing inter-thread dependences, improving workload balance among threads, and increasing the amount of exploitable MLP.
   - A mechanism to statically combine and select the most appropriate scheme to manage an inter-thread dependence. Dependences can be either (i) ignored, (ii) fulfilled through an explicit communication instruction pair, or (iii) fulfilled by a pre-computation slice.
   - A software/hardware technique to dynamically reconstruct the original sequential order of memory instructions that have been arbitrarily assigned to speculative threads.
   - A cost-effective hardware support that allows the execution of Anaphase threads on top of a conventional multi-core processor thanks to a novel module placed in the uncore, named Inter-Core Memory Coherency Module (ICMC), that interfere very little with the cores.

- A software/hardware hybrid scheme to perform register and memory checkpointing that allows taking frequent checkpoint with very little overhead.

1.5 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 presents the first proposal of this thesis in the area of Speculative Multithreading, the Mitosis Architecture, to speed-up sequential applications in multi-core systems. The Mitosis execution model is described, with special emphasis on introducing the concept of pre-computation (*p-slices*) as a powerful way to deal with inter-thread dependences. The components of the Mitosis architecture, the compiler and the hardware support, are extensively discussed. On the compiler side, we describe a novel mechanism to select speculative threads, and we outline how *p-slices* are built, as well as the different *p-slice* optimizations we propose to reduce their overhead. On the hardware side, we discuss the organization of the register file and the memory subsystem to support the Mitosis execution model, focusing on the support introduced for managing *p-slices*. Finally, we evaluate the whole proposal and present the main conclusions that we extract from these studies.
- Chapter 3 describes a novel SpMT scheme proposed in this thesis, the Anaphase Architecture, which addresses several constraints of traditional SpMT schemes. We describe in detail the novel threading model, the Anaphase fine-grain thread decomposition algorithm with special emphasis on the different heuristics proposed in this thesis, and the cost-effective hardware support for running Anaphase threads on top of a conventional multi-core processor. This includes a detailed description of the software/hardware mechanism to reconstruct the original sequential order, the Inter-Core Memory Coherency Module (ICMC), as well as the novel scheme for performing register and memory checkpointing. Several studies for the Anaphase proposal are presented and conclusions are finally drawn.
- Chapter 4 summarizes the main conclusions of this thesis and outlines some future steps and open research areas.

Chapter 2

The Mitosis Architecture

The Mitosis Architecture is a combined hardware-software speculative multithreading approach to speed-up sequential applications in multi-core systems. Thanks to its novel features, Mitosis is able to parallelize irregular applications and achieve good performance even in the presence of frequent dependences among threads.

This chapter describes the Mitosis execution model and the components of the Mitosis architecture, the compiler and the hardware support. It introduces the concept of pre-computation slices (p-slices) as a powerful way to deal with inter-thread dependences. On the compiler side, it presents a novel mechanism to select speculative threads, and it outlines how p-slices are built, as well as the different p-slice optimizations we propose to reduce their overhead. On the hardware side, it discusses the organization of the register file and the memory subsystem to support the Mitosis execution model, focusing on the support introduced for managing p-slices. Finally, it presents the performance studies of the whole proposal and the main conclusions that we extract from these studies.

2.1 Introduction

In Speculative Multithreading (SpMT) architectures [3][5][16][27][33][44][63][89][96], inter-thread dependences are one of the most crucial factors that affects overall performance. This is because in the presence of dependences the threads may not execute independent of each other, which may constrain the exploited parallelism and the achievable performance boost of the system.

It is occasionally possible to partition a program into parallel threads such that there are few or no dependences between them [90]. For SpMT architectures that take this approach, they speculate that there are no dependences among threads, and they recover in the case of a dependence not being met. However, for most programs, it is necessary to generate threads where there are control/data dependences across these partitions to fully exploit available parallelism. The manner in which such dependences are managed critically affects the performance of the speculative multithreaded system.

Previous approaches in the literature have used explicit synchronization [27][96] and hardware value prediction [62] to manage these dependences. Since most of the difficult to parallelize sequential applications have features like highly control intensive code, pointer chasing etc., synchronization might be an overly conservative technique and may lead to overly serialization of the threads. On these cases, hardware value prediction may be a more appropriate scheme to deal with data dependencies between speculative threads.

Unfortunately, the accuracy of most hardware value predictors is highly dependent on the context of the predicted value. In SpMT, for a newly spawned thread, the context between the spawning point and the starting point of the thread is unavailable which leads to lower prediction accuracy. This problem becomes worse when the thread size increases, leading to increase in the amount of lost context information and the amount of values to predict. In order to squeeze every TLP and ILP available in a program, the threading scheme should be able to spawn threads at many different granularities ranging from small threads of few tens of instructions to large threads containing several thousand instructions. Unfortunately, hardware value predictors fail to perform very well at large thread sizes. Therefore, to unveil all the potential of the SpMT model a new method for managing inter-thread dependences is required.

Another important aspect that determines the goodness of a SpMT architecture is the parallelization coverage achieved by its thread partitioning scheme. In other words, how much code of the sequential application is parallelized when it is decomposed into speculative threads. In fact, for achieving good performance it is desirable to have a thread partitioning scheme that provides high parallelization

coverage, with good workload balance among threads, while taking into account other factors like the amount of inter-thread dependences and the thread size. Many SpMT architectures in the literature propose thread partitioning schemes based on high-level program structures, like loops [16][23][33][53][60][90] or subroutines [5]. Although this is a relatively easy and natural scheme to implement, it may not provide the best partition based on the aforementioned criterions. An alternative approach is not to partition based on any program constructs but directly analyzing the control-flow and data-dependence graphs of a piece of code, looking for threads that satisfy all the mentioned criterions. Even though some works in the literature follow this approach [58], there is still room for improvement and better algorithms are required in order to effectively partition irregular applications into speculative threads that provide good performance benefit.

In this chapter we present the Mitosis architecture. Mitosis [30][54] is a hybrid software / hardware SpMT architecture that is able to exploit TLP even on the presence of frequent dependences among threads. This is achieved, thanks to its unique scheme to deal with inter-thread dependences. Mitosis takes a completely new software value prediction based approach for managing both data and control dependences among threads. We find that this scheme produces values earlier than synchronization approaches and more accurately than hardware value predictors. High prediction accuracy is obtained thanks to the fact that the software prediction mechanism uses instructions derived from the original parallelized code.

In the Mitosis execution model, each speculative thread starts with a pre-computation slice (*p-slice*) that computes the thread input values, also known as thread live-ins. Thread live-ins are those register and memory values that are consumed by the speculative thread and may be computed by prior threads still in execution. The *p-slice* typically executes in a fraction of the time of the actual code that produces those live-ins, because it skips over all code that is not specifically computing live-in values.

A key observation is that *p-slices* are speculative in nature since they predict the thread input values but correct execution is always guaranteed, even in the case of misprediction, by the underlying Mitosis hardware (as in any SpMT architecture). This observation opens the door to many aggressive optimizations that may be applied to *p-slices* in order to reduce their overhead without sacrificing too much accuracy.

Moreover, for achieving high parallelization coverage with good workload balance among threads, the Mitosis architecture implements a general thread partitioning scheme that is able to generate speculative threads at any point of the program and identify the most effective points to spawn those

threads. The main novelty of the proposed scheme is the use of a model to statically estimate the performance benefit of a set of threads. The model takes into account all the desirable criterions we have previously mentioned like high parallelization coverage, good workload balance among threads, few inter-thread dependences (i.e. small *p-slices*), and appropriate thread size.

On the hardware side, the Mitosis architecture is based on a multi-core processor extended with specific support for running Mitosis speculative threads. The hardware support contains a number of novel features. These include support for the execution and validation of *p-slices*, a new multi-version register file that manages global register state and inter-core dependences with virtually no latency overhead, and a multi-version memory subsystem that uses a replication cache to allow the processor to achieve cache performance similar to what would be seen by the application if it were running in a single core.

Overall, the Mitosis architecture provides the necessary software and hardware elements to address the main constraining issues of previous SpMT systems and achieve good performance improvements for running sequential applications on current and future multi-core platforms.

The rest of the chapter is organized as follows: In Section 2.2 we present the background concepts of Speculative Multithreading and *p-slices*, whereas in Section 2.3 we introduce the execution model used in the Mitosis architecture. In Section 2.4 we describe the Mitosis compiler with special emphasis on the thread selection mechanism and the generation and optimization of *p-slices*. Section 2.5 describes the hardware support implemented in the Mitosis multi-core processor for executing speculative threads including the proposed designs for the register file and the memory system to manage speculative state and handle the execution of *p-slices*. In Section 2.6 we present the experimental evaluation of the Mitosis architecture including the experimental framework used to analyze the proposals made in this work and the results obtained with our proposals. Finally, we conclude this chapter in Section 2.7.

2.2 Background

Before describing the Mitosis architecture, it is worth to introduce some concepts and terminology that is later used on our proposal. In particular, the main concepts that we introduce in this section are the Speculative Multithreaded model that we assume in the Mitosis model, and the idea of pre-computation slices. Both concepts are fundamental ingredients of the Mitosis architecture.

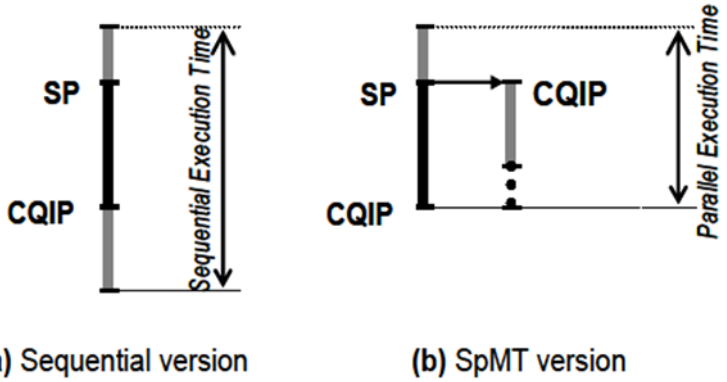


Figure 4. Sequential versus SpMT parallel execution based on spawning pairs

2.2.1 Speculative Multithreading

As described in Section 1.3.2, conventional Speculative Multithreading (SpMT) architectures decompose a sequential application into multiple threads that run in parallel. These threads may be dependent on data and control on previous threads in the program execution order. Speculative threads are ordered according to the control flow of the original sequential application. At any time during execution, the earliest thread in program order is *non-speculative* while the others are *speculative*. A speculative thread is created when a *spawn* action (i.e. a thread light-fork operation) is executed and it starts executing in a free *thread unit* (in a multi-core processor a thread unit could be a core).

We consider a SpMT architecture where a potential speculative thread is defined by a *spawning pair*. A *spawning pair* consists of the point in which the spawning instruction is inserted and the point where the speculative thread will start execution when it is spawned. In this thesis, we follow the terminology presented by Marcuello [58], where the point with the spawn instruction is called the *spawning point* (SP), and the point where the speculative thread starts is called the *control quasi-independent point* (CQIP), so called because it is expected that the future execution flow likely reaches this point. Figure 4 shows a simplified example of a *spawning pair* on the sequential execution (a) and the SpMT execution (b) of the threads defined by this pair.

In our terminology the thread that performs the spawn is called the *spawner* thread, while the spawned speculative thread is called the *spawnee* thread.

2.2.2 Pre-computation Slices

Program Slicing was first introduced by Weiser [102] as a method to abstract program behavior. The Slice of a program is a subset of instructions of the original program which can execute independently and can possibly affect the values computed at a certain point in the program; the point is referred to as the slicing criterion. Slicing criterion in this context can be described as the pair: (*Instruction*, *Variable*). The *Variable* could be one defined by the *Instruction* or one of its inputs. A program slice contains only those instructions from the original program which affect the slicing criterion and ignores the rest. For example program slicing has been very useful in debugging, where the pair consisting of instruction *I* and variable *V* taking a spurious value becomes the slicing criterion. Then the slice includes all those instructions from the original program which when executed in the program order affect the value of the variable *V* at instruction *I*. In any program there are two kinds of instructions which affect the value taken by a variable: instructions which participate in data flow and those which determine the control flow of the program. The slice contains all those instructions that participate in the data computation as well as the control flow computation.

Figure 5 illustrates the concept of program slice. In Figure 5(a), each node shows an instruction in the program and the arrows depict dependencies between them. The tail of the arrow points to the instruction which affects the instruction at the head of the arrow. Node *H* is the instruction and variable comprising the slicing criterion. We isolate all those instructions which affect the behavior of *H* by traversing the graph following its direct and indirect predecessor dependencies. By selecting these instructions we get the graph of Figure 5(b), which shows only the instructions that affect node *H* (i.e. the slice of the original program). The process of computing the program slice is called Slicing.

Program Slices are categorized into two types based on input to the program: Static Slices and Dynamic Slices. Static slices are built from the program using only information that is available statically, i.e. independent of any input to the program. On the other hand, Dynamic Slices are built for a program using a specific input set. The slices proposed by Weiser [102] were static slices. Dynamic slicing was first proposed by Korel and Laski [43]. In Dynamic Slicing, the slicing criterion distinguishes between the different instances of the same static instruction. There have also been proposals to build Hybrid slices, i.e. static slices built using some dynamic run time feedback [25].

Slices can also be classified based on the direction of slices as: Backward Slices and Forward Slices [83]. In the example above, the slice was built by backward traversal of the graph, starting at the slicing criterion. Hence they are called Backward Slices. A forward slice consists of all instructions that

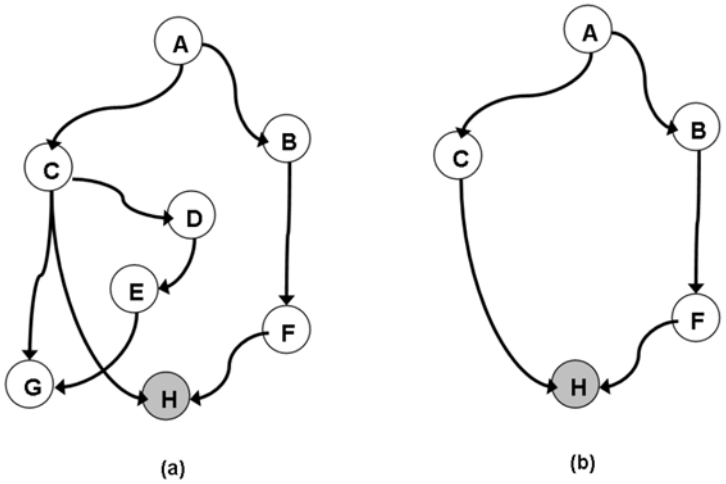


Figure 5. Program Slicing

depend on the slicing criterion. Backward as well as Forward Slices are computed in similar fashion, i.e. traversing the data flow and control flow dependence edges bottom-up in the former case, and top-down in the later.

Program Slicing has been used in many different contexts. Depending on the problem context, the instruction of interest for which the slice is built may vary. Some examples of different applications of program slices are: debugging, program maintenance, and data pre-fetching.

In the Mitosis architecture, we propose a novel use of program slicing. In the next section we describe how program slices are used in the Mitosis execution paradigm and in the following sections we describe a method to construct program slices at compile time and several optimizations we propose to reduce their overhead without sacrificing too much accuracy.

2.3 The Mitosis Execution Model

One of the fundamental aspects of the Mitosis architecture is its ability to exploit TLP even on the presence of frequent dependences among threads. This is achieved, thanks to its unique scheme to deal with inter-thread data and control dependences.

In the Mitosis architecture, we propose a novel use of program slicing as a way to predict/compute the input values of speculative threads, a.k.a. live-ins (values consumed but not produced by the thread). In this context, program slices may be described as *pre-computation slices (p-slices)*, which is the term we would be using throughout this thesis. The *p-slices* considered in Mitosis can be classified as *hybrid static/dynamic* and *backward* slices.

In Mitosis, *p-slices* are generated at compile time and added at the beginning of each thread. A *p-slice* typically executes in a fraction of the time of the actual code that produces those live-ins, because it skips over all code that is not specifically computing live-in values. An important observation is that in SpMT systems, hardware support for recovery may allow to apply aggressive and unsafe optimization techniques to *p-slices* in order to significantly reduce their cost, even sacrificing correctness for spare cases. In that sense, we can say that *p-slices* in the Mitosis architecture are *speculative* in nature.

The use of *p-slices* in Mitosis can be seen as a software-based value prediction mechanism. The main advantages of using *p-slices* compared to other prediction schemes are:

(1) They are potentially more accurate in the prediction of live-ins than a hardware-based predictor, since it is derived from the original code.

(2) They can easily encapsulate multiple control flows that contribute to the prediction of live-ins.

(3) They can accelerate the detection of incorrectly spawned threads (i.e. threads that are spawned but need to be cancelled because the sequential stream is not going to reach the thread starting point).

On the other hand, the use of *p-slices* place important implications on the whole architecture (compiler, microarchitecture, and execution model) that we will detail thorough this chapter. Given its fundamental role, Mitosis can be defined as a *p-slice* based Speculative Multithreading model.

2.3.1 P-slice based Speculative Multithreading Model

The execution model of a *p-slice* based SpMT paradigm, like the one we propose in the Mitosis architecture, is very similar to any conventional SpMT model (see Section 1.3.2). However, the use of *p-slices* to predict the input values of threads requires some specific features in the execution model that we will describe in this section.

In the Mitosis architecture there is always one (and only one) non-speculative thread, which is the only thread allowed to commit its results and guarantees forward progress. All other threads are speculative, and are ordered according to the control flow of the original sequential application. As previously pointed out in this section, Mitosis speculative threads are specified by a SP-CQIP pair, where SP stands for Spawning Point and CQIP stands for Control Quasi-Independent Point [58]. A speculative thread is created when a thread reaches a Spawning Point (SP) and executes a *spawn* instruction, which has been inserted by the Mitosis compiler. Note that in this model any thread can spawn a new thread and that speculative threads can be spawned out of the program order; that is, threads can be created in a

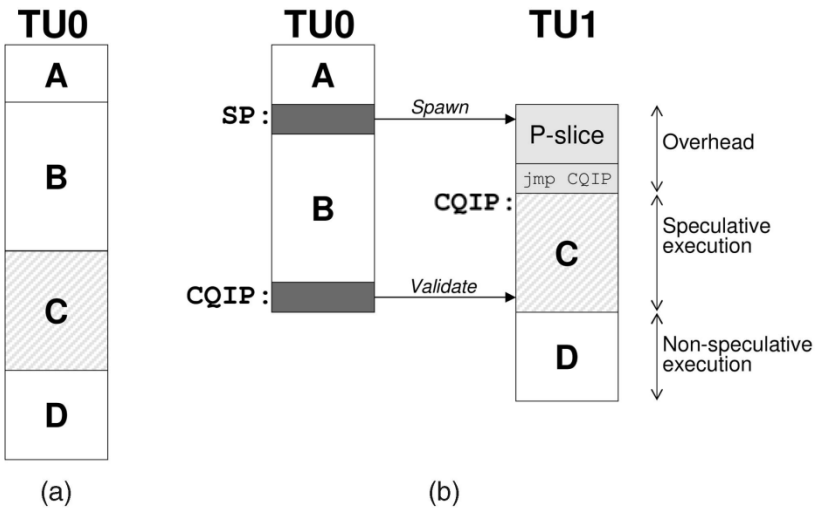


Figure 6. Mitosis execution model. (a) Sequential execution. (b) SpMT execution.

different order than they will be committed. The model allows to spawn a thread less speculative (previous in sequential time) than current running threads, so nested spawning pairs are also supported. The requirements to spawn a thread are: (i) there is a free thread unit (i.e. a core in a multi-core processor), or (ii) there is at least one running thread more speculative (further in sequential time) than the thread to be created. In the latter case, the most speculative thread is cancelled and the freed thread unit is assigned to the spawned thread.

Figure 6 shows a simplified example of the Mitosis execution model using *p-slices* on a dual core multi-core processor: (a) represents the sequential execution of several consecutives regions of code (A, B, C, D), and (b) shows the parallel execution of this code using the Mitosis SpMT paradigm. In the example, the choice of a particular SP and CQIP allow the code in region C (the code following the CQIP) to execute in parallel with the code in region B (the code between the SP and CQIP). When the thread running on thread unit TU0 reaches the SP, it spawns a speculative thread on an idle thread unit TU1. After some initialization, the speculative thread executes the region C starting at CQIP instruction, which is identified as an operand of the *spawn* instruction. In our terminology the thread that performs the spawn is called the *spawner* thread, while the spawned speculative thread is called the *spawnee* thread.

Correct speculative execution of the code following the CQIP requires that the future state (memory and register values) of the processor at the CQIP is correctly predicted. As mentioned, the Mitosis model differs from other SpMT proposals in that each speculative thread includes a pre-computation slice (*p-slice*) that computes the live-ins of the thread. The role of the *p-slice* is therefore to generate the memory and registers values that the *spawnee* thread needs from its previous threads. It

consists of instructions from the program between SP and CQIP which when executed lead to correct computation of all memory and register dependencies with the *spawnee* thread. When a speculative thread is spawned, it first executes the *p-slice* which produces all the dependent values, after which the speculative thread starts executing at the CQIP, thanks to an unconditional branch to this point at the end of the *p-slice*.

In Mitosis, a speculative thread has two operation modes, depending on whether it is executing code from the *p-slice* or the body of the thread. This distinction arises from the fact that the *p-slice* predicts architectural state, whereas the speculative thread body (following the *p-slice*) computes actual architectural state. Therefore, data produced by the *p-slice* must be confirmed, but never committed. In particular, data produced while in *p-slice* mode is stored in a special buffer and will be used as input for the body of the speculative thread. Data produced by the speculative thread is kept in the regular structures of the thread unit (register file and memory) and will be committed once the thread becomes non-speculative. Threads that overflow the speculative buffering space need to be squashed.

Threads commit in sequential order. This means that a thread must wait to become the oldest one (i.e., the non-speculative thread) to commit. When a running thread reaches the CQIP of an active thread, it stops fetching instructions until it becomes non-speculative. Then, a verification process called *Validation* checks that the next speculative thread has been executed with the correct input values, that is the *p-slice* has properly computed the live-ins. If there is a misspeculation, the next speculative thread and all its successors are squashed, and the non-speculative thread continues executing the instructions beyond the CQIP. If the speculation turns out to be correct, the non-speculative thread is committed, and its TU is freed. Moreover, the next speculative thread becomes the non-speculative one: it will either continue executing in non-speculative mode or, if it has already reached another CQIP, immediately proceed to verify its successor. In the case of correct speculation, the parallel overlap between the threads gives the performance benefit of this execution model.

As can be seen from the example in Figure 6, the amount of parallelism exposed by this model is sensitive to the size of B (the distance between SP and CQIP) and the size of the *p-slice*. If the size of the *p-slice* is similar to that of B, so most of the computation of B is consumed in C, then there will be little or no gain from this model. If the *p-slice* is very small relative to B, then the code will execute as if the regions were completely parallel, despite the existence of dependences. This motivates the importance of choosing a proper set of spawning pairs, and applying aggressive optimizations to *p-slices* to reduce their overhead, in order to obtain performance benefit with the Mitosis execution model. This is the job of the Mitosis compiler, which is described in detail in the following section.

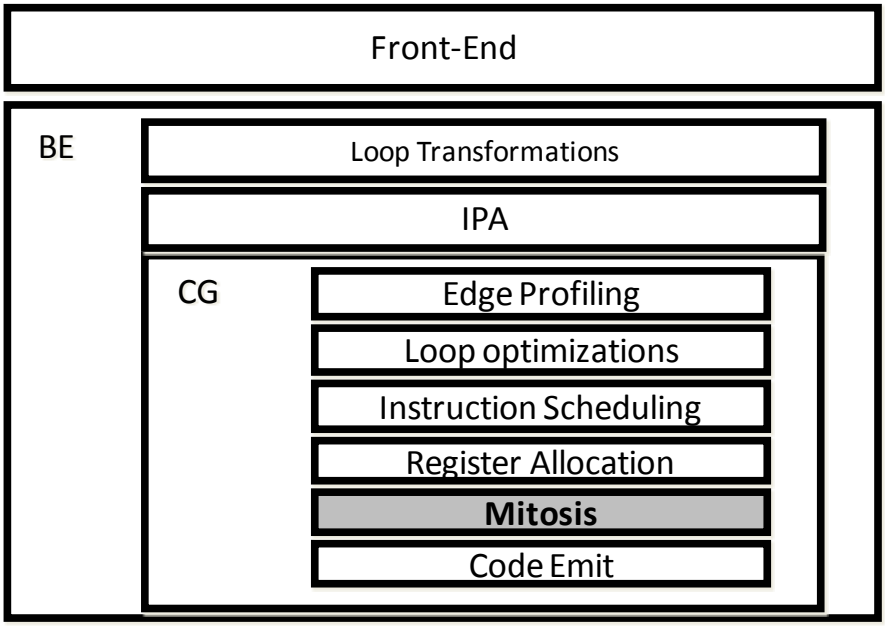


Figure 7. Back-end of a conventional compiler with the Mitosis scheme

2.4 The Mitosis Compiler

As previously mentioned in this chapter (see Section 2.1), the Mitosis SpMT architecture is a hybrid software / hardware system. The software part, which is in charge of automatically parallelizing applications using speculative threads, has been implemented on top of a production compiler (see Section 2.6.1 for more details). Figure 7 shows the back-end of a conventional compiler with the Mitosis scheme implemented in it. As can be seen, the proposed scheme has been implemented in the code generation phase, after optimizations and before emitting code.

The job of the Mitosis compiler is to partition an application into speculative threads by choosing a set of spawning pairs so that thread units (i.e. cores) are efficiently used. The compiler takes into account many parameters to guarantee efficiency: threads and *p-slices* overheads, workload balance, interaction among speculative threads, high parallelization coverage, etc... On the following sections we will discuss all the factors considered by the compiler in further detail.

In order to achieve this objective, the Mitosis (MTS) compiler performs the three following highly coupled tasks:

1) Analyzes, selects, and inserts the best spawning pairs at any point of the program being compiled.

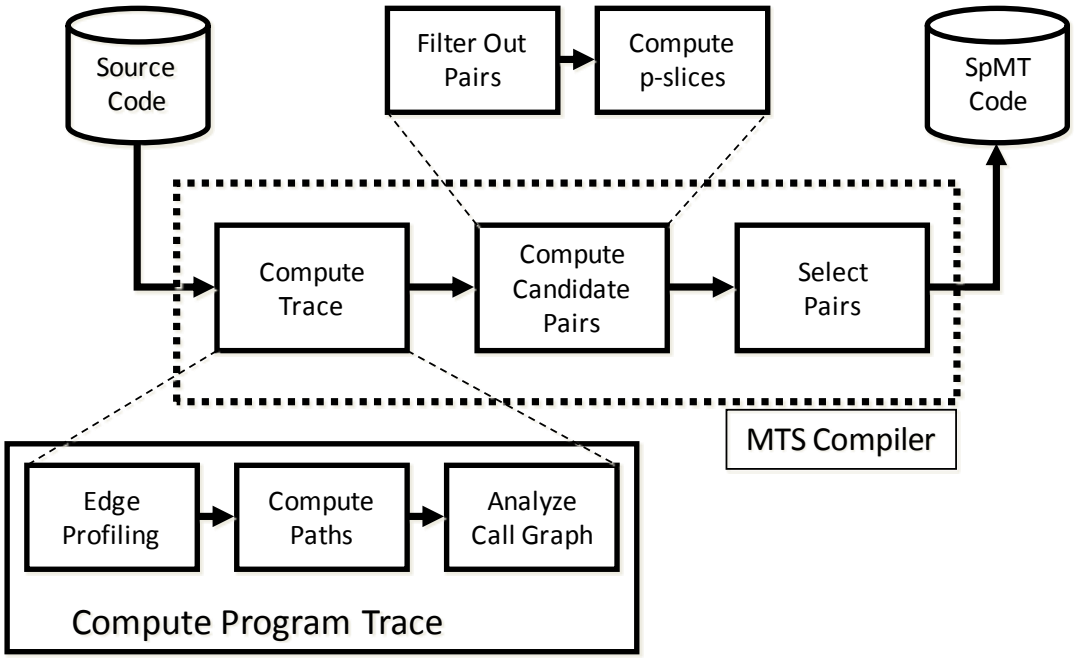


Figure 8. Mitosis compilation steps

2) Generates the *p-slice* for each pair in order to predict/compute the thread live-ins.

3) Optimizes each *p-slice* to minimize its overhead without sacrificing too much accuracy.

Dynamic or runtime information is extremely valuable in order to efficiently perform these tasks. The compiler obtains this runtime information making use of different profilers. On the one hand, the information provided by an edge profile is used to better select a set of spawning pairs. This information includes the probability of going from any basic block to each of its successors and the execution count of each basic block. On the other hand, the information of memory dependences, which is used to accurately deal with inter-thread dependences and to properly compute the *p-slice*, is obtained with the help of a memory dependence profiler.

Figure 8 shows a scheme of the different steps performed by the Mitosis compiler. Notice that, as mentioned, the aforementioned tasks are tightly coupled and are decomposed into multiple steps in this scheme. In a nutshell, starting from the program source code, the compiler first generates a program trace using runtime information obtained by the edge profiler. The objective of this first stage is to have a representative trace of the whole program execution. Then, it computes a set of possible candidate spawning pairs, filtering out those not worth to consider, and generates for each pair its corresponding *p-slice*. With all this information, the compiler selects the best set of spawning pairs using a greedy search.

The search is guided by a cost model that estimates the expected benefit of any set of potential spawning pairs. Finally, the selected spawning pairs with its corresponding *p-slice* are passed to the compiler Code Emit stage to generate the SpMT, or Mitosis, binary.

We will detail all these steps in the following sections. Although the order of these steps is the one shown in Figure 8, for the sake of clarity we will first describe in Section 2.4.1 all the steps related to the identification and selection of spawning pairs, and then we will deal with the generation and optimization of *p-slices* in Section 2.4.2.

2.4.1 Spawning Pair Identification and Selection

Mitosis partitions an application into speculative threads without being constrained to program structures (loops, routines, modules, …). As previously mentioned (see Section 2.1), this is crucial for achieving good parallelization coverage in many irregular applications.

In order to perform the thread partitioning, the compiler first identifies all the possible candidate spawning pairs from any point of the program and then selects the best set of them. This is a complex process that can be decomposed into three main steps (see Figure 8):

1) *Compute Trace*: Build a synthetic trace using runtime information obtained by the edge profiler. The objective of this first stage is to produce a small trace representative of the whole program execution.

2) *Compute Candidate Pairs*: Generate a set of candidate spawning pairs by filtering out those pairs that do not meet some desirable characteristics like high start probability, minimum and maximum region size, maximum *p-slice* overhead, etc… All these pair characteristics are measured based on the synthetic trace built on the previous step.

3) *Select Pairs*: Obtain the set of selected pairs by means of a greedy search algorithm. The search algorithm is directed by a cost model that estimates the expected performance benefit of any set of potential spawning pairs by traversing the synthetic trace and keeping track of the evolution of the threads.

Notice that candidate spawning pairs are obtained based on desirable characteristics on the execution of the pair. This requires accurate dynamic information of the whole program execution. The compiler gets this information from the synthetic trace generated on the first step.

Once the candidates are obtained, selecting the best set of spawning pairs requires assessing the performance benefit of any given candidate pair. However, determining the benefits of a particular spawning pair is not straightforward and cannot be done on a pair-per-pair basis. The effectiveness of a pair depends not only on the control flow between the SP and the start of the thread, the control flow after the start of the thread, and the accuracy and overhead of its *p-slice* but also on the number of hardware contexts (i.e. cores) available to execute speculative threads and interactions with other speculative threads running at the same time.

This analysis requires a model of program execution. To avoid capturing and repeatedly traversing a full path trace of the program, we use again the small synthetic trace of execution that captures the dynamic behavior of the program. The key idea is to traverse this trace while keeping track of the threads that are active at any time. For each thread, its state is maintained to emulate its evolution during its lifetime. This analysis emulates the timing behavior of the speculative threads, assuming a simple model where each instruction takes a fixed time (i.e. one cycle). Based on this, the compiler can estimate the expected benefits of any set of spawning pairs, and select those that are expected to minimize total execution time.

All these steps will be explained in more detail in the following subsections.

2.4.1.1 Compute Trace

We build a synthetic trace of the program to translate the edge profile data into path information, without having to capture and maintain a full path profile. The synthetic trace is built based on edge profiling information at the basic block level. The analysis performs a reverse topological traversal of the call graph. This means that callee routines are analyzed before callers.

For each routine, we compute a set of paths. A routine path is defined as a list of connected nodes in the control flow graph (CFG) from the entry node to an exit node. We assume here that a routine has only one entry. A path node can be one of these types: basic block (BB), loop, or call. Loop and call nodes are macronodes in the sense that they include more than a single basic block. A loop node consists of the header of a given loop and contains all the nodes (basic blocks, inner loops or calls) that belong to that loop. A call node is just a basic block that ends with a call to a particular function. The exit node of a routine path could be either a return (node with no successor) or a call to the exit function. Thus, there are two types of routine paths: return paths and exit paths. Each routine path is characterized by its total length (in instructions) and its probability (using the edge profiling information). This is summarized in the following expressions:

```
ROUTINE = SET OF {ROUTINE PATH}
ROUTINE PATH = LIST OF {NODE} + TYPE + LENGTH + PROB
ROUTINE PATH TYPE = {RETURN | EXIT}
NODE = {BB | LOOP | CALL}
```

A loop node requires more analysis. For each loop, we compute a set of loop paths. A loop path is defined as a sequence of nodes in the CFG of the loop from the head of the loop to a possible loop exit node. A loop exit node can be a node with an edge to the loop head (continue path), an edge outside the loop (break path) or a call to a function that may call the exit function (exit path). We are assuming here that a loop has a single header. As in the case of routines, each loop path has its length (in instructions) and probability. This is summarized with the following expressions:

```
LOOP PATH = LIST OF {NODE} + TYPE + LENGTH +
PROB LOOP PATH TYPE = {CONTINUE | BREAK | EXIT}
```

Next, a set of synthetic traces are built for each loop. A loop synthetic trace consists of selecting NITER – 1 loop paths of type continue and 1 loop path of type break or exit, where NITER is the average trip count. For large loops, we build the trace assuming a fixed maximum value for the trip count. The rest of the iterations are considered to have the same behavior as the analyzed ones. The selection of the paths inside the loop consists of a random weighted model according to each path probability.

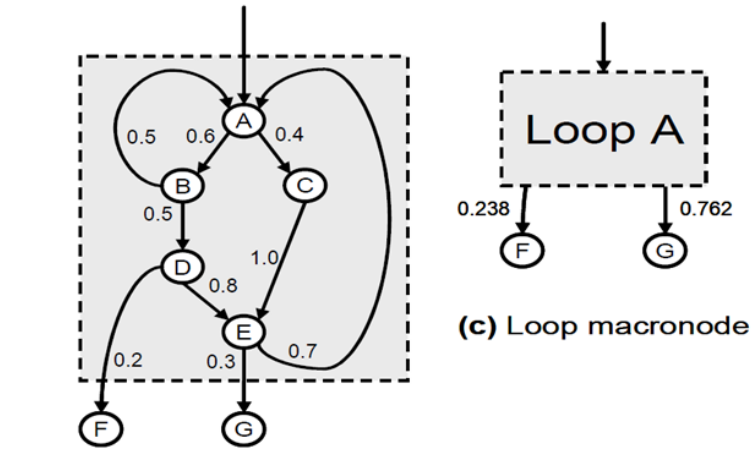
The number of loop synthetic traces that are built per loop is a parameter of the compiler and can be adjusted depending on time and memory space requirements. We call each loop's synthetic trace a loop instance. For each loop node the compiler keeps a list of instances and a pointer to one of them (initially the first one), that is updated in a circular fashion.

```
LOOP = CIRCULAR LIST OF {LOOP INSTANCE} + POINTER + TRIP COUNT + LENGTH
LOOP INSTANCE = LIST OF {LOOP PATH}
```

The length of a loop node is computed as the weighted average length of the loop instances. Figure 9 shows an example of how a given loop is split in its different paths and the shape of the resultant loop macronode.

A call node also has attached a list of instances. A call instance refers to a possible path in the callee routine. The compiler builds this list of instances, similarly to loops, by randomly selecting (based on the probabilities) routine paths from the callee routine. In the same way, a pointer is also attached.

```
CALL = CIRCULAR LIST OF {CALL INSTANCE} + POINTER + LENGTH
CALL INSTANCE = ROUTINE PATH
```



(a) Control-flow graph of the a loop

(c) Loop macronode

TYPE	PATH	PROB	NEXT
Continue	{A B}	0.300	A
Continue	{A B D E}	0.168	A
Break	{A B D E}	0.072	G
Break	{A B D}	0.060	F
Continue	{A C E}	0.280	A
Break	{A C E}	0.120	G

(b) Loop paths

Figure 9. Construction of a loop macronode

The length of a call node is computed as the weighted length of routine paths in the callee function. The synthetic trace will be traversed by starting the path at the main routine of the program. As the main routine is called only once, there is only one routine path for that function. When a loop node is found, the traversal proceeds through the loop instance pointed by the loop instance pointer and the pointer is set to the next instance. In the case of a call node, the traversal proceeds through the path in the callee routine described by the call pointer and the pointer is set to the next one in the same way as for loops. When an exit path (or the only path in the main routine) is completely traversed, the program traversal finishes.

2.4.1.2 Compute Candidate Pairs

A key feature of the proposed compilation tool is its generality in the sense that it can discover speculative TLP in any region of the program. The tool is not constrained to analyze potential spawning pairs at loop or subroutine boundary, but practically, any pair of basic blocks is considered a candidate spawning pair.

To reduce the search space, we first apply the following filters to eliminate candidate pairs that likely have little potential:

- Spawning pairs in routines whose contribution to the total execution of the program is lower than a threshold are discarded.
- Both basic blocks of the spawning pair must be located in the same routine and at the same loop level. Notice that this does not constrain the possibility of selecting candidate pairs different from loop iterations and subroutine continuations. For instance, potentially important spawning pairs like those that partition large sequential regions are also eligible.
- The length of the spawning pair (as the average length of all the paths from the SP to the CQIP) must be higher than a certain minimum size in order to overcome the initialization overhead when a speculative thread is created. It must also be lower than a certain maximum size in order to avoid very large speculative threads and stalls due to the lack of space to store speculative state.
- The probability of reaching the CQIP from the SP must be higher than a certain threshold.
- Finally, the ratio between the length of the *p-slice* and the estimated length of the speculative thread must be lower than a threshold. This ratio is a key factor in determining the potential benefits of the thread.

This step analyzes the different routines in the program one by one. For each routine, all combinations of basic blocks are considered and passed through the different filters. The result of this process is a set of candidate spawning pairs (candidate pairs for short) of the whole program. For each candidate pair the following information is kept: (i) basic block for the spawning point, (ii) basic block for the CQIP, (iii) probability that the *p-slice* reaches the CQIP and average length in this case, and (iv) average length of the *p-slice* when the CQIP is not reached (in this case, the speculative thread is cancelled before its body is started).

2.4.1.3 Select Pairs

Once the set of candidate pairs is built, the selection of pairs follows the greedy algorithm shown in Figure 10. The basic idea is to include pairs in the selected set until a negligible benefit is obtained. Among all pairs in the candidate set, the new pair chosen (if any) is the one that provides the best improvement to the current selected set. The benefit is computed using a model that estimates the execution behavior of a set of pairs for a given number of thread units (i.e. cores). Notice that with this method, the algorithm takes into account at each step all the interactions of the candidate pair being evaluated with the previous selected pairs.

```
[ 1] t_exec = SeqExecTime;
[ 2] Selected_Pairs = ∅;
[ 3] exit = FALSE;
[ 4] while (!exit) {
[ 5]   select = NULL;
[ 6]   for (cand=First_Candidate(Candidate_Pairs); cand!=NULL; cand = Next_Candidate(cand)) {
[ 7]     Analized_Pairs = Selected_Pairs + cand;
[ 8]     t_exec_tmp = Model_Set_of_Pairs(Analyzed_Pairs, Program_Trace, N_Thread_Units);
[ 9]     if (t_exec_tmp < t_exec) {
[10]       t_exec = t_exec_tmp;
[11]       select = cand;
[12]     }
[13]   }
[14]   if (select == NULL)
[15]     exit = TRUE;
[16]   else {
[17]     Candidate_Pairs = Candidate_Pairs – select;
[18]     Selected_Pairs = Selected_Pairs + select;
[19]   }
[20] }
```

Figure 10. Greedy algorithm to select spawning pairs

The inputs to this algorithm are: (i) the program trace (see Section 2.4.1.1), (ii) the set of candidate pairs (see Section 2.4.1.2), and (iii) the number of thread units. Initially, the execution time is set to the equivalent execution time when no spawning pairs are considered (*SeqExecTime*) and the set of selected pairs is empty (lines 1 and 2 in Figure 10). Then, the greedy loop begins (line 4). At each iteration, all individual pairs in the candidate set are considered one by one, in conjunction with the pairs already selected (line 7). The execution time of the program for each of these new sets of spawning pairs is estimated. The model for this estimation is explained in detail in the following subsection. If any new set of pairs is better than the currently selected one (line 9), the new pair is kept in the *SELECT* variable and execution time is updated. If no new set results in significant improvement, the greedy algorithm finishes (lines 14 and 15). However, if a given combination improved the previous execution time, the *SELECT* pair, which contains the best pair in the candidate set, is removed from the candidate set (line 17) and added to the selected set (line 18).

The result of the greedy algorithm is the set of selected spawning pairs to speculatively parallelize the application. Note that, the algorithm performs an exhaustive search and so under certain parameters for the filtering, this can be a long time consuming process. However, improving such potential compile time issue is not part of this thesis. As we achieve success and experience with finding effective pairs, we expect to be able to refine the search process significantly (see Chapter 4 for future work avenues).

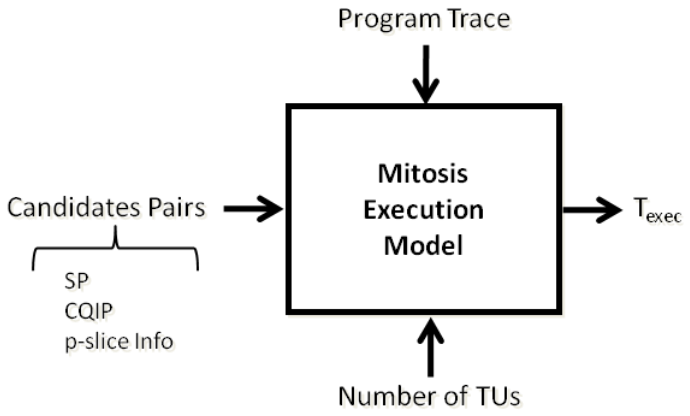


Figure 11. SpMT estimation model used by the pair selection algorithm

2.4.1.3.1 SpMT Estimation Model

As described before, the core of the selection algorithm is an objective function able to estimate the execution time of a program for a given set of spawning pairs (line 8 in Figure 10). This objective function is based on a causal model that analyzes the behavior and interactions of the set of spawning pairs when the program is executed on the given processor with SpMT support. As shown in Figure 11, the inputs of that model are: (i) the number of thread units, (ii) the synthetic program trace, and (iii) the set of candidate spawning pairs.

For each candidate pair, the model uses the following information: spawning point (SP); control quasi-independent point (CQIP); average length of the *p-slice* when the CQIP is not reached and probability that this happens; average length of the slice when the CQIP is reached and probability that this happens (*p-slice* info in Figure 11).

The output of the model is the estimated SpMT execution time. For the sake of simplicity in the model we assume that the execution of any instruction takes a unit of time. However, the model can be extended in a straightforward manner to include different execution times for each static instruction (e.g., using average memory latencies obtained through profiling).

The model analyzes the evolution of threads during execution traversing the synthetic program trace. It works as follows: the program trace is analyzed sequentially. Only a subset of basic blocks is analyzed. The analyzed blocks are: the first and last basic blocks in the trace, and SP and CQIP basic blocks. During the trace traversal, two global variables are updated:

- *Current time*: the time at which the current basic block instance is being executed.

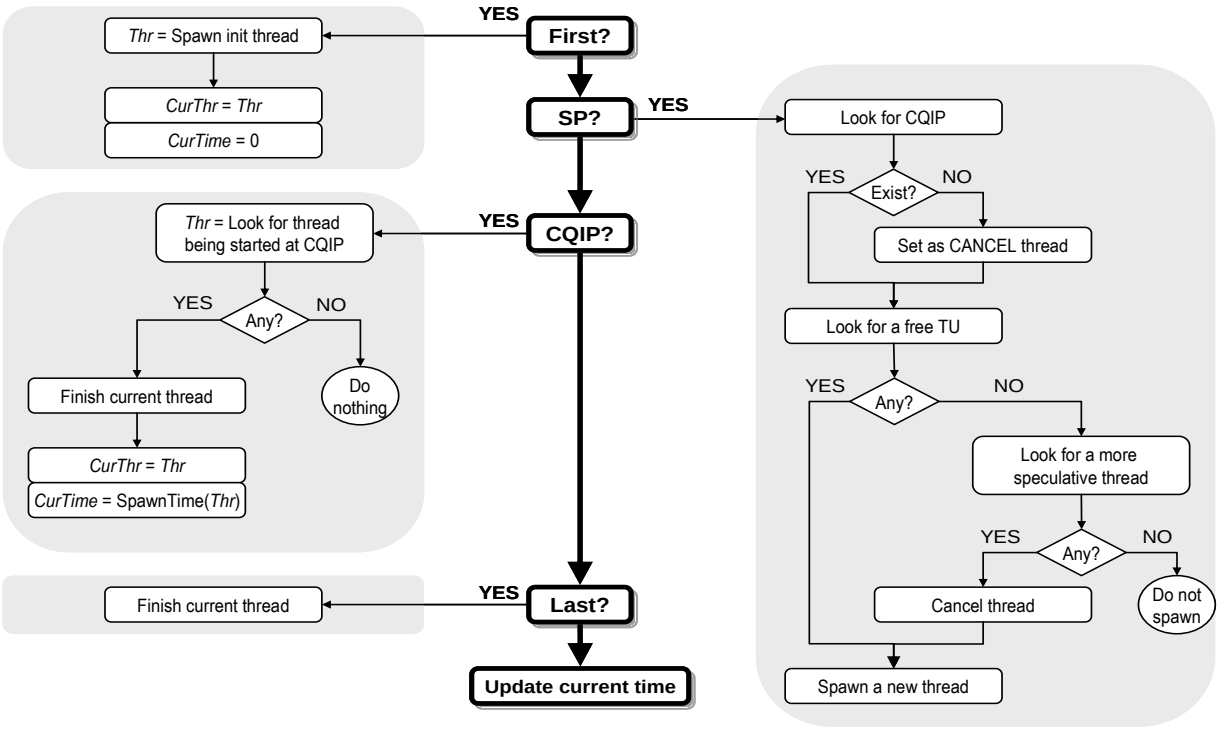


Figure 12. Flowchart describing the modeling of Mitosis SpMT execution

- *Current thread*: the thread that executes the current basic block instance under analysis.

The core of the model is shown in Figure 12. Initially, a single non-speculative thread is assumed. This thread is allocated to any thread unit and is supposed to be responsible for the execution of the whole trace, so its end time is accordingly initialized.

When a basic block that corresponds to a SP in the set of pairs is found, the actual modeling of the SpMT starts. If the basic block corresponding to the CQIP does not exist in the remainder of the trace or the given CQIP has been already executed by another thread, the thread is marked as a CANCEL thread (a thread that exits before the CQIP is reached); otherwise, we mark it as a NORMAL thread. In either case we look for a free thread unit. If a thread unit is available we assign it to the new thread. Otherwise, we check if the most speculative thread is further in program order (more speculative) than the new thread. If so, the most speculative thread is cancelled and the freed unit is allocated to the new thread. Otherwise, the spawn is discarded.

When a basic block that corresponds to a CQIP in the set of pairs is found, it is checked whether any more speculative active thread was started at this basic block instance. If this is the case, the current thread is terminated and current thread and current time variables are updated accordingly. Finally, the

last basic block of the trace just terminates the current thread. The commit time of this last thread represents the SpMT execution time of the program.

Spawning a new thread requires the following actions:

1) Identify the order of this new thread with respect to the current ones. Its previous thread is the thread that contains the CQIP of the new thread. Its next thread is the thread that was the successor (before spawning) of its previous thread.
2) Decide whether this is a CANCEL thread or a NORMAL one: this is randomly selected based on the cancel probability of each type for this particular thread.
3) Record the start and end basic blocks of the thread. The former is the current CQIP and the later is the start of the next thread.

Finally, canceling a thread requires the following actions: (1) identify previous and next threads, and (2) update links and end information of the previous thread.

2.4.2 P-slice Generation and Optimization

The Mitosis architecture handles inter-thread dependences through the execution of a pre-computation slice (*p-slice*) inserted at the beginning of every speculative thread. The Mitosis compiler is responsible of generating the *p-slice* of every speculative thread.

In order to generate the *p-slice* of a thread, the compiler first builds the Program Dependence Graph (PDG) [26] of the thread. The PDG of a program consists of two sub graphs: (i) the Data Dependence Graph (DDG), and (ii) the Control Flow Graph (CFG). The DDG is defined as the graph G_{data}: (V_{data}, E_{data}) where V_{data} is the set of all vertices representing the instructions in the program. E_{data} denotes the set of edges representing the data dependencies between the instructions. An edge $u \rightarrow v$ such that *u* and *v* are nodes in the graph, shows that *v* is data dependent on *u*. Similarly, CFG is defined as the graph $G_{control}$: ($V_{control}$, $E_{control}$) where $V_{control}$ denotes the set of vertices representing all instructions and $E_{control}$ denotes the set of edges $u \rightarrow v$ such that *u* is a branch instruction and *v* is control dependent on the branch *u*.

Figure 13(a) shows an example PDG of a thread bounded between the SP and CQIP points. The nodes representing branch instructions are illustrated using the diamond shaped boxes, and the non-branch instructions are shown using the round shaped ones. The straight line edges between the nodes show the data dependencies between the instructions. Note that for simplicity the control dependences are shown at the granularity of the basic-blocks only. For example, the edge connecting nodes 1 and 3 implies

that instruction 3 takes as input the value produced by the instruction 1. The dotted edges represent the control dependencies. For example, the edge connecting node 4 to the basic blocks BB2 and BB3 implies that the instructions in basic blocks BB2 and BB3 will execute depending on the outcome of the branch instruction 4, hence they are control dependent on 4.

As described in Section 2.2.2, a *p-slice* is built in two steps: 1) firstly identifying the instructions of interest (i.e. the slicing criterion), and 2) traversing the PDG starting from the instruction of interest in the bottom-up direction.

2.4.2.1 Identifying Live-ins

In the case of *p-slices* for the Mitosis SpMT architecture, the instructions comprising the slicing criterion are the instructions in the *spawner* thread which directly produce the values consumed by the *spawnee* thread. These instructions are called the live-in instructions and the values produced by them are called live-ins, as these values are live across the thread boundary (i.e. are the input values of the thread).

The way live-ins are identified is as follows: The PDG of the *spawnee* thread is traversed from top to bottom starting at CQIP, noting every register and memory value read by the thread which it did not produce itself. If any of these values were produced by an instruction between the SP and CQIP of the *spawner* thread, it is marked as a live-in instruction. Notice that for memory values the information of memory dependences is obtained based on profiling.

It is important to notice that once the *spawner* thread commits, its updated values are available to the *spawnee* thread. Thus, any value that is produced by the *spawner* thread but consumed by the *spawnee* thread only after the *spawner* has committed does not need to be produced by the *p-slice*. Therefore only those live-ins which appear in the *spawner* thread that are overlapped with the *spawnee* execution should be taken into account for *p-slice* generation. Since there is no way to know the size of the actual *p-slice* a-priori, the overlap region between the *spawner* and the *spawnee* thread should be approximated. An approximate way of measuring the overlap region is to assume its maximum possible size without taking into account the size of the *p-slice*. The maximum size is obtained subtracting from the *spawner* thread length the spawn overhead, where length and spawn overhead are measured in terms of instruction count.

2.4.2.2 Generating P-slices

Conceptually the *p-slice* would be a subset of instructions from the *spawner* thread which when executed would produce the live-ins. In order to construct this subset of instructions, the PDG of the *spawner* thread is traversed in the bottom-up direction starting at the live-in instructions, following all the data and

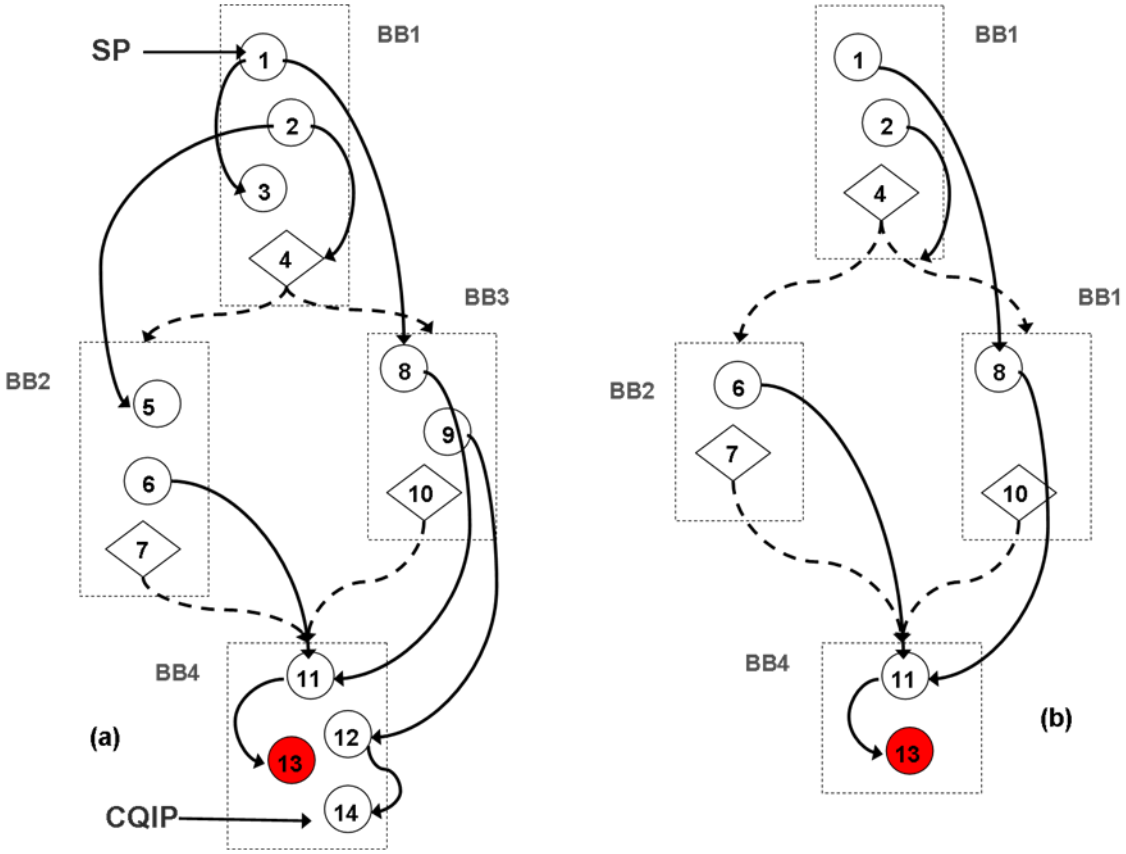


Figure 13. *P-slice* construction (a) PDG of thread (b) PDG of slice

control dependencies, including every instruction traversed as long as it is younger than the SP in program order.

Node 13 in Figure 13(a) represents a live-in instruction, i.e. the value produced by it is directly consumed by the *spawnee* thread. The *p-slice* of this thread, then, is a subset of instructions from the thread that can correctly generate the value produced by instruction 13. In this example, instruction 13 is dependent on instruction 11, which in turn is dependent on instructions 6 and 8 and so on. Depending on whether the branch 4 is taken or not-taken, instructions 6 or 8 may execute. Thus by traversing the data-dependence and control-dependence edges in the PDG, one can identify all those instructions between the boundaries of SP and CQIP, which are needed to correctly compute the live-in instruction. For the thread shown in Figure 13 (a), by traversing the dependence edges starting from the live-in instruction, we construct the *p-slice* as shown in Figure 13(b).

One key aspect of the generation of the *p-slices* is the accuracy on determining the dependences, and in particular for the memory related ones, memory disambiguation is a very important aspect to

consider. Initially, all dependences among instruction as given by the compiler in the conventional / conservative way are considered. This is desirable for register values but it is by far too conservative for memory dependences, where static memory disambiguation is used. The compiler tends to introduce too many memory dependences to guarantee correctness, which may dramatically increase the number of instructions included in the *p-slice*. In order to alleviate this issue, the Mitosis compiler obtains the information of memory dependences, for identifying the thread live-ins and generating the *p-slice*, using a memory dependence profiler (see the following section).

Upon finding a call to a subroutine, the side-effects (i.e. values generated) of that subroutine as well as the use of the returned value(s) are analyzed, and if there is any dependence, the call instruction to the subroutine is included into the slice. This means that the slice includes the whole subroutine, although the full code of that subroutine may not be needed[1].

As shown in Figure 13, the *p-slice* contains only a subset of instructions from the original thread and hence potentially executes faster than the thread. The amount of overlap between the *spawner* thread and the speculative *spawnee* thread is determined by the execution time of the *p-slice*, and hence it directly impacts the performance achieved by the system.

In this thesis, we propose speculative optimizations to reduce the length of *p-slices* together with mechanisms to constraint the aggressiveness so that we do not sacrifice too much accuracy.

2.4.2.3 Optimizing P-slices

P-slices built using the conservative assumptions of the compiler, as described above, are normally very large, which significantly constrain the benefits of speculative threads. However, a key feature of the Mitosis SpMT architecture is that it can detect and recover from misspeculations. Regardless of the code we generate for a given *p-slice*, the Mitosis architecture guarantees a functionally correct execution of the program. Thus accuracy of *p-slices* only affects performance, not correctness. This opens the door to new types of speculative / unsafe optimizations that otherwise could not be applied by the compiler, and which have the potential to significantly reduce the overhead of *p-slices*.

Speculative optimizations require a new factor in the thread partitioning analysis: the misspeculation probability. This factor represents the probability that a given *p-slice* is incorrect. This

[1] A possible optimization, not considered by the Mitosis compiler, would be to inline or specialize some subroutines [80].

happens when some live-ins are not computed or they are incorrect. This probability is attached to each candidate pair and used by the model described in Section 2.4.1.3.1 when deciding whether a spawned pair is NORMAL or CANCEL.

In the following subsections, we describe the set of safe and speculative optimizations included in the Mitosis compiler. For the actual threshold values used in the Mitosis compiler for performing these optimizations see the description of the experimental framework (Section 2.6.1). Results for several evaluation studies of these optimizations can be found in Section 2.6.2.

2.4.2.3.1 Early Cancellation

When a new thread is spawned, it is not always guaranteed that its starting point, the CQIP, is going to be reached by the previous thread. This is due to the fact that the CQIP is not fully control-independent of the SP (i.e. there are control paths from the SP that do not reach the CQIP). This is a safe optimization that allows a thread to cancel itself as soon as it is known that its starting point (CQIP) will not be reached. The purpose of this optimization is not to reduce the *p-slice* overhead but to reduce the useless activity of the SpMT system. In the case of not reaching the CQIP of a spawned thread, the thread would keep executing useless instructions (wasting power and keeping a thread unit busy) until it is squashed by a less speculative thread.

This optimization performs a reachability analysis on the CFG from the SP to the CQIP to identify the points in the program where we can guarantee that the CQIP will never be reached. Then, the optimization introduces a *cancel* instruction in each of these points of the *p-slice* in order to squash the thread when executed.

Figure 14(a) shows the CFG of a spawning pair and its conservative *p-slice* (b). Notice in (a) that from basic block '*I'* the CQIP cannot be reached. After applying the Early Cancellation optimization, a *cancel* instruction is introduced in the *p-slice* at that point to cancel the execution of the speculative thread when this path is taken.

2.4.2.3.2 Memory Dependence Speculation

As commented in Section 1.3.1, parallelizing compilers often fail to parallelize applications because of ambiguous memory dependences. Many memory dependences are only included because the compiler cannot prove that the corresponding instructions are independent, but in fact they are. In many other cases, two static instructions do have a memory dependence, but this dependence only happens for a very few dynamic instances of these instructions.

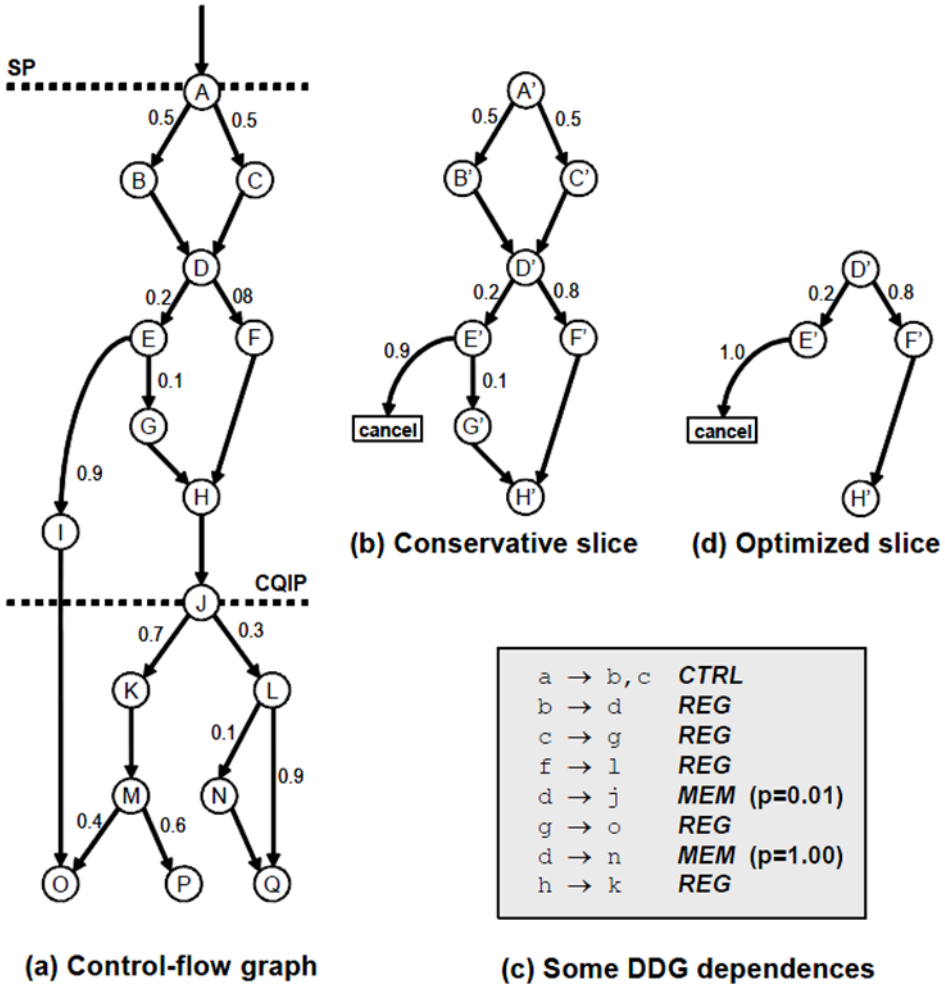


Figure 14. Example of *p-slice* optimizations

The Mitosis compiler uses profiling information for memory dependences to minimize the number of unnecessary dependences considered when generating *p-slices* by speculating that infrequent dependences never occur. The implemented profiler computes the dependence frequency between any pair of *store-load*, *store-call*, *call-load* or *call-call* instructions for each subroutine (the SP and CQIP of a spawning pair are in the same subroutine). Dependences for *calls* refer to dependences due to any memory reference inside the called routine.

With the memory profiling information, the compiler only considers that two instructions have a memory dependence whenever this dependence has happened with a frequency above a given threshold. In other words, dependences that never occur in practice, or occur very infrequently, are not considered for *p-slice* generation.

2.4.2.3.3 Branch Pruning

Branch pruning is a speculative optimization that focuses on conditional branches that are strongly biased (i.e., they normally follow the same edge) and optimizes the code by eliminating the infrequently taken paths. This optimization allows ignoring those paths that exhibit low probability of being taken when generating a *p-slice*. These paths may belong to either the body of a speculative thread or its *p-slice*, with different consequences in each case.

Pruning branches of the body of a speculative thread is done during the process of identifying the live-ins of the thread (see Section 2.4.2.1). A pruned branch is still included in the thread body code, but any live-in in the pruned path is ignored when the *p-slice* is generated, which reduces the size of the *p-slice*. Notice that in case a pruned live-in is actually consumed by the thread, the value may be incorrect. This case is handled by the architecture, which validates that all consumed live-ins values are correct and squashes the thread otherwise.

On the other hand, pruning a branch in the *p-slice* removes all the instructions of the pruned path. Additionally, predecessors of these removed instructions are also removed if their output is not used elsewhere. In the place of a pruned path, a *cancel* instruction is inserted; if this path happens to be taken, the thread input values will likely be miscomputed, and it is preferable to cancel the thread and free this thread unit for another thread.

Examples of both types of branch pruning are shown in Figure 14. On the left, (a) shows the CFG of a spawning pair. Each edge is annotated with its probability of being taken (edges without label have probability 1.0). On the right (b), the CFG of the conservative *p-slice* is shown. Basic blocks are labeled with prime letters to indicate that they contain just a subset of the instructions of the original basic blocks. Some data dependences among instructions in some basic blocks are also listed in Figure 14(c) (lower case letters represent instructions in basic blocks with the corresponding capital letter, e.g., instruction '*a*' is in basic block '*A*'). A possible edge (i.e., branch) to be pruned in the speculative thread due to its low probability is $L \rightarrow N$, which will remove a live-in (data dependence $d \rightarrow n$) and then some instructions in the slice (dependence $b \rightarrow d$ is not needed). On the other hand, an example of pruning in the *p-slice* would be for edge $E \rightarrow G$, which will remove the data dependence $g \rightarrow o$. This will remove in turn the need for dependence $c \rightarrow g$ in the *p-slice*. In this example, as no instructions are needed from basic blocks *B* and *C* (since their dependences have been removed), the control dependence $a \rightarrow b,c$ can also be removed from the slice. The resulting optimized *p-slice* is show in Figure 14(d).

2.4.2.3.4 Dependence Pruning

Data dependences that are infrequent can also be ignored. For memory dependences, the profiler described in Section 2.4.2.3.2 is used. In the case of register dependences, the probability of reaching the producer once the spawn has been executed is computed and multiplied by the probability of reaching the consumer after executing the producer. Note that a consumer can be located either in the slice or the speculative thread body. As in the case of memory dependences, if this probability is lower than a threshold, the dependence is ignored for the purpose of generating the final *p-slice*.

2.4.2.3.5 Cancel Elimination

As previously discussed, *cancel* instructions are inserted at points where the compiler can guarantee that the speculative thread is incorrect or the flow cannot reach the CQIP. This allows the processor to squash early a speculative thread in order to free the thread unit for other threads.

However, inserting a *cancel* instruction implies that the branch instruction leading to the pruned code cannot be removed (neither all its ancestors in the dependence graph). This overhead may be large in some cases, which significantly impacts the effectiveness of the speculative thread. In these cases, it may be more effective just to remove the *cancel* operation and the associated branch instruction (which will also remove some of its ancestors). This will make the slice always follow the frequent path, which can be incorrect in some infrequent cases. In these cases the architecture will detect the misspeculation and squash the thread.

In fact, in the Mitosis compiler, cancel elimination and branch pruning optimizations are performed together. A greedy algorithm is used in order to select the set of branches that are pruned based on a cost function that computes the benefit of pruning a branch and eliminating the *cancel* instruction associated with it. This mechanism accurately models the effect of each branch pruned and *cancel* eliminated in terms of length and correctness of the resulting *p-slice* code, thus, making the selection of branches to prune and *cancel* instructions to eliminate a more informed decision. We describe the algorithm, called cost-effective branch pruning, in the following section.

2.4.2.3.6 Cost-effective Branch Pruning

Figure 15 depicts the flowchart of the algorithm used in the Mitosis compiler for performing branch pruning and cancel elimination at the same time. In a nutshell, the algorithm starts with a set of candidate branches to be pruned (those that are more biased than a given threshold), and assigns a benefit score to each of them based on a cost function that takes into account both the reduction in length of the *p-slice* and the misspeculation probability when the branch is pruned and when the associated *cancel* instruction is removed.

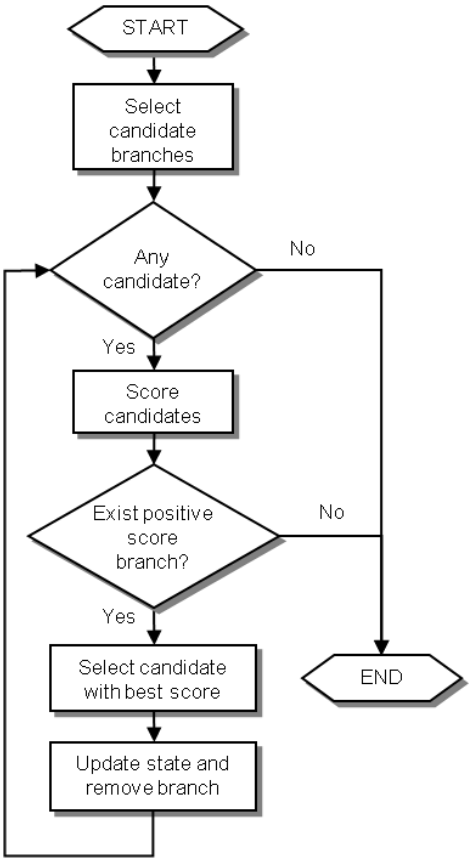


Figure 15. Flowchart describing the cost-effective branch pruning greedy algorithm

Once all the candidates have been assigned a benefit score, the subset of branches to prune is selected. There may be several heuristics to select that set of branches to prune. Finding the optimal solution would require to try all combinations, which is a NP-complete problem. We use a greedy algorithm to select the branch that gives the best benefit at each step. After removing this branch, the benefits of the remaining candidates is recomputed and the process is repeated until the benefit of removing every candidate branch left is smaller than the benefit of the current configuration. The main steps of the algorithm, the branch scoring and the updating of the state, are described in more detail in the following bullets.

- *Branch Scoring:*

Each candidate branch is given a value that represents the goodness of pruning it (with or without inserting the associated *cancel* instruction). This value describes the additional number of instructions that are correctly parallelized when the branch has been prunned. The branch score is based on a BENEFIT function that estimates the number of correctly parallelized instructions. The value

assigned to the branch (i.e. the score) is the difference between the current benefit and the benefit after the branch is pruned. If this branch is selected for removal, current benefit will be updated accordingly.

$$SCORE_{branch} = BENEFIT_{after} - BENEFIT_{current}$$

The BENEFIT expression estimates the amount of correctly parallelized instructions taking into account the size of the optimized *p-slice*, its misspeculation and cancel probability, and the number of parallelized instructions that are not wasted in a misspeculation thanks to including a *cancel* instruction on a pruned path. The BENEFIT function is computed as follows:

$$BENEFIT = PROB_{correct} \times (LEN_{total} - LEN_{CQIP}) + PROB_{cancel} \times FACTOR_{cancel}(LEN_{total} - LEN_{cancel})$$

Each member of the BENEFIT function is defined as follows:

- $PROB_{cancel}$ → Probability that the *p-slice* executes a *cancel* instruction
- $PROB_{CQIP}$ → Probability that the *p-slice* does not execute a *cancel* instruction and reaches the CQIP
- $PROB_{correct}$ → Probability that the *p-slice* correctly computes all live-ins
- LEN_{CQIP} → Average length of the *p-slice* when it reaches the CQIP
- LEN_{cancel} → Average length of the *p-slice* when it cancels due to a *cancel* instruction
- LEN_{total} → Average length of the *p-slice* plus the length of the overlapped region in the speculative thread (see Section 2.4.2.1)

The previous members are computed based on all the possible paths of the *p-slice* and the overlapped region of the speculative thread. Each member is computed as follows:

$$PROB_{cancel} = \sum_{\forall PATH} (IS_{cancel} \times PROB_{occur})$$

$$PROB_{CQIP} = (1 - PROB_{cancel})$$

$$PROB_{correct} = \sum_{\forall PATH} ((1 - IS_{cancel}) \times (1 - PROB_{misspec}) \times PROB_{occur})$$

$$LEN_{CQIP} = \sum_{\forall PATH} ((1 - IS_{cancel}) \times LEN_{p\text{-}slice}) \Big/ PROB_{CQIP}$$

$$LEN_{cancel} = \sum_{\forall PATH} (IS_{cancel} \times LEN_{p\text{-}slice}) \Big/ PROB_{cancel}$$

$$LEN_{total} = \sum_{\forall PATH} ((1 - IS_{cancel}) \times (LEN_{p\text{-}slice} + LEN_{thread})) \Big/ PROB_{CQIP}$$

For each path in the *p-slice* and the overlapped region of the speculative thread, the following information is used in the previous formulas:

- $LEN_{p\text{-}slice}$ → Length of the path in the *p-slice* in number of instructions
- LEN_{thread} → Length of the path in the overlapped region of the speculative thread in number of instructions
- $PROB_{occur}$ → Probability that the path occurs
- $PROB_{misspec}$ → Probability that the path incurs in a misspeculation (at the beginning of the algorithm all the paths have probability to misspeculate of 0.0)
- IS_{cancel} → A boolean that indicates if the path finishes with a *cancel* instruction

The function $FACTOR_{cancel}(n)$ in the BENEFIT expression is introduced to represent the benefit of using *cancel* instructions. As previously commented, these allow the detection of a misspeculation in a timely manner. If this function was not present, the early detection would not have any benefit with respect to letting it be incorrect and detect it with a validation mechanism (that would not detect the misspeculation as early as the *cancel* does). No *cancel* instruction would be inserted, and, of course, early detection may be beneficial. We define $FACTOR_{cancel}$ as a linear function:

$$FACTOR_{cancel}(n) = 0.35 \times n$$

Notice that the actual value of this function depends on the effects that early detection has in the particular context where branch pruning is applied. The value chosen (0.35) for $FACTOR_{cancel}$ has been obtained based on experimental results.

- *Updating the State*

When a branch is selected to be pruned, the state has to be updated in order to reflect this change in the set of paths of the *p-slice* and the overlapped region of the speculative thread. The actions taken are different depending on whether the pruned branch is in the *p-slice* or in the overlapped region of the speculative thread.

In the second case, pruning a branch in the speculative thread body, as described in Section 2.4.2.1, actually means ignoring live-ins (no *cancel* is inserted nor instructions are removed). In this case, all the *p-slice* paths that contain any producer of the live-in being pruned are marked with $PROB_{misspec} = 1.0$.

In the first case, pruning a branch in the *p-slice*, we distinguish two different situations:

1) The branch is pruned, but a *cancel* instruction is inserted at the target of the pruned edge. In that case, all *p-slice* paths containing that edge are marked as cancel paths by setting IS_{cancel} to true.

2) The branch is pruned and no *cancel* instruction is inserted. In that case, all the paths containing the pruned edge and a dependence chain from a producer instruction being pruned to a consumer in the speculative thread are marked with $PROB_{misspec} = 1.0$.

After computing the new misspeculation probabilities ($PROB_{misspec}$), all the paths containing the pruned edge will be removed from the list of paths and the probabilities of the removed paths will be distributed among the alternative paths with the non-pruned edge. In addition, after pruning the branch, the length of each path of the *p-slice* is recomputed in order to account for the removed instructions.

After updating all paths, the set of variables defined in the previous bullet will be recomputed with the new status in order to score the remaining candidate branches to be pruned.

2.4.2.4 Related work

Program Slices, of which pre-computation slices (*p-slices*) is a specific type, was first proposed as a technique for debugging large programs [102]. Since then, program slices have been used to solve a host of different software engineering problems like program understanding, testing, differencing, and merging.

P-slices have also been used for a variety of purposes in computer architecture. Mainly among them are Helper Threads [15][19][105]. Helper threads are built in similar fashion to *p-slices* for speculative threads but they differ in their purpose. Helper threads are used to prefetch long latency load data or predict hard to predict branches. Due to the potentially higher impact of a misspeculation in case of speculative threads, the *p-slices* are built to deliver higher accuracy than helper threads. The work presented in this thesis, the Mitosis architecture [30][54], has been a pioneering work in the use of *p-slices* for Speculative Multithreading. There have been other SpMT proposals later that used some sort of pre-computation to deal with inter-thread dependences.

MSSP [106] used what they call as a Distilled Program which runs ahead of the Slave threads and computes their inputs beforehand. Conceptually the distilled program is a kind of *p-slice* but there is just one program (a.k.a thread) which computes the *p-slices* for all the threads, unlike Mitosis where every thread executes its own *p-slice*. In Mitosis, the fact that the computation of the thread live-in values is done by the speculative threads allows the processor to spawn threads out of the program order, and to often compute the live-ins for speculative threads in parallel.

Program Demultiplexing (PD) [5] uses handlers which are similar in functionality to *p-slices*, their role is to compute the function parameters for the speculative threads, since in PD threads are limited to functions. Compared to Mitosis, this is a much more constrained model since only functions can be speculatively parallelized. The experimental results in Mitosis confirm that there is a lot of potential parallelism inside functions that this model can not exploit.

Finally, Speculative Fission [108] is yet another recent work that uses pre-computation. This scheme focuses on loops as threads candidates. The compiler is responsible for identifying all the do-all loops (i.e. loops that the compiler can guarantee the independence among iterations) as well as those loops which can be easily converted into do-all loops by pre-computing all the dependencies between the loop iterations, before entering into the loop body. At the first sight it might look to be more advantageous than a *p-slice* based scheme like Mitosis, as *p-slices* compute the dependencies but do not update the architectural state and hence are pure overhead. However, it is important to note that in many sequential applications which are very control intensive and have features like pointer chasing etc., the instructions that compute the dependencies between the iterations tend to make up a significantly large portion of the loop body itself and hence chunking it out of the loop leaves very little code to be parallelized. On the other hand, as described in previous section, *p-slices* are speculative pieces of code and hence aggressive optimizations can be applied to reduce their overhead.

Several SpMT architectures, like MSSP [106], have proposed the use of branch pruning to reduce the length of speculative code. However, the pruning performed in these proposals is, compared to the proposal presented in this thesis, very simplistic. They just ignore infrequent code (and the branches leading to them), but they do not perform any analysis on what is the advantage in terms of length of removing a branch and they do not allow the insertion of *cancel* instructions in the pruned branches. Moreover, the analysis implemented in the Mitosis compiler considers whether it is better to insert a *cancel* or not if the branch is pruned. Furthermore, the misspeculation rate is not predictable in these simpler schemes (they can just have an inaccurate upper bound of it). This implies that with a more accurate scheme, like the proposed for Mitosis, the model may aggressively prune branches if it predicts this is a good decision.

Finally, an alternative technique for performing branch pruning in *p-slices*, a.k.a. Slice Specialization, has recently been proposed by Ranjan [80]. The idea is to remove those branches that are easily predictable by hardware branch predictors and as a result create specialized slices for the different possible control flows inside the slice.

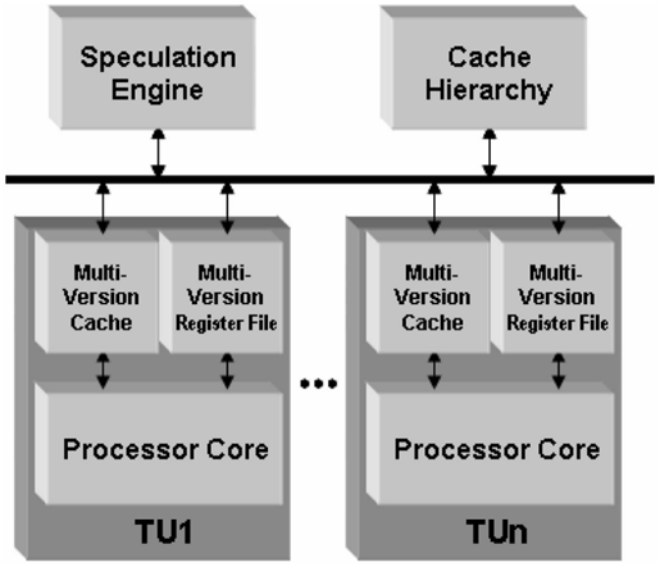


Figure 16. Mitosis multi-core processor scheme

2.5 The Mitosis Multi-core Processor

The Mitosis architecture presented in this chapter is composed of two big components: (i) the Mitosis compiler described in Section 2.4, which is responsible of partitioning an application into speculative threads, and (ii) the Mitosis processor, which includes all the microarchitecture support for running Mitosis speculative threads. This section describes in detail the Mitosis processor, with special emphasis on the specific features required for supporting the Mitosis *p-slice* based SpMT execution model.

Figure 16 shows a high level block diagram of the Mitosis processor. It has a multi-core design based on an on-chip multiprocessor (CMP), where each thread unit (TU) is similar to a conventional superscalar core and is able to execute a single thread at a time. Every core has three operation modes: (i) non-speculative mode for conventional execution, (ii) speculative mode for executing speculative threads, and (iii) *p-slice* mode for executing the *p-slice* of every speculative thread.

To support the execution model presented in Section 2.3, some subsystems of a conventional CMP processor have to be modified due to the speculative nature of the threads. Thus, the Mitosis microarchitecture provides specialized hardware support for spawning, validating, squashing and committing speculative threads. All these operations are orchestrated by a unit called Speculation Engine.

In addition, the Mitosis processor includes mechanisms to manage the speculative architectural states (or versions) that every running thread generates and the inter-thread data dependences among

them. Notice that in a SpMT architecture, each speculative thread may have a different version for each logical register and memory location. This is because speculative threads share the register and memory space and can access the same registers and memory locations. However, each thread has a different view of a given register or memory location that corresponds to the particular point in the dynamic instruction stream where this thread is running.

The ability to have different active versions for each register and memory location and handle inter-thread communication of register and memory values is supported by means of the Multi-version Register File (MVRF) and the Multi-version Cache (MVC). These two distributed structures maintain a single global view of register and memory contents while allowing multiple local versions of those values. More precisely, they deal with the following tasks:

- *Speculative state buffering*: When a core is in speculative mode, the register and memory state produced by the speculative thread is locally kept in each TU and not propagated so as not to compromise the correct execution of the application. In particular, since the execution of speculative threads may be incorrect, their state cannot be merged with that of the safe state of the program. If a violation is detected the state generated by the misspeculated thread is discarded. Otherwise, at thread commit the state is allowed to propagate to the safe architectural state of the program.

- *Inter-thread data dependence violation detection*: Communication of memory and register values among threads is handled taking into account the sequential program order. In addition, data dependences are dynamically tracked and in case of a read-after-write violation (i.e. a younger thread consumes a value before being produced by an older thread) a misspeculation is signaled.

- *Violation recovery*: The system is able to restore any speculative change, so that the architectural state remains valid when a data dependence violation has occurred. In case of misspeculation, the faulting thread and all its successors in sequential program order are squashed (the speculative memory and register state is invalidated and the thread is killed).

In the field of SpMT systems, some register file and memory architectures able to keep multiple versions of each architectural register and memory location have already been studied (see Section 1.3.2.4). However, they do not fit the Mitosis execution model well due to its novel scheme to deal with inter-thread data dependences based on *p-slices* to pre-compute the live-ins of each speculative thread. This has several implications and challenges for the microarchitecture. Although the *p-slice* and the

thread execute consecutively on the TU, they differ in significant ways. First, the *p-slice* executes in the speculative domain of the parent (at the *spawning point*), whereas the thread executes in a more speculative domain, following the CQIP. Second, register and memory values produced by the *p-slice* must be validated but never committed: they predict the processor state but are not processor states. Conversely, values produced in the thread must be committed but do not need to be validated.

Therefore, the microarchitecture must include the following features for supporting the execution of *p-slices*:

- Due to the aforementioned differences on the execution of the *p-slice* and the body of the speculative thread, every core has to distinguish between both. In other words, Mitosis introduces two operation sub-modes to the speculative mode for distinguishing between the execution of the *p-slice* and the body of the speculative thread.
- *P-slice* register and memory writes must be handled differently from regular writes. In particular, data produced while in *p-slice* mode is stored in special buffers and will be used as input for the body of the speculative thread and later discarded when the thread commits. Data produced by the speculative thread is kept in the regular structures of the thread unit (register file and private data caches) and will be committed once the thread becomes non-speculative.
- In order to support the execution of the *p-slice* as it was executing in the speculative domain of the parent (at the *spawning point*), the state of the parent thread must remain visible to the *p-slice.* That implies that all the values overwritten by the parent must be kept during the execution of the *p-slice*.

The specific implementation of these features, as well as other novel features of the architecture to support the Mitosis SpMT model, will be described in the following sections. The Multi-version Register File and the Multi-version Memory subsystem will be discussed in Section 2.5.1 and Section 2.5.2, respectively. Due to the strong coupling with these units, the operations orchestrated by the Speculation Engine will be detailed in these sections too.

2.5.1 Multi-version Register File

To achieve correct execution at high performance, the Mitosis register file architecture must simultaneously support the following seemingly conflicting goals: (i) a unified view of all committed register state, (ii) the coexistence of multiple versions of each architectural register, (iii) register

dependences that cross TUs, and (iv) a latency similar to a single local register file. This support is provided by the Multi-version Register File (MVRF).

Like in traditional speculative multithreaded schemes, the proposed MVRF architecture allows multiple versions of each architectural register and incorporates mechanisms to communicate these versions among the running threads. However, in order to support the features of the Mitosis execution model some additional requirements have to be met. These requirements are:

- During the execution of a speculative thread, any register read should return the previous written value in sequential order, also known as the last version. Thus, on a register read the required register value can be obtained from the thread itself, if it has previously written the register, or from the closest thread in sequential order that has produced a version of the register.
- When a new thread is spawned, in order to correctly execute its *p-slice*, any read within the slice to a not previously written register should return the value available in the parent thread before the *spawning point*.
- Before committing a speculative thread, it is necessary to validate that its register live-ins have been correctly pre-computed and that all register inter-thread dependences are met.

Figure 17 shows the overall block diagram of the MVRF architecture on a Mitosis processor with *N* thread units, or cores, and *R* architectural registers. As can be seen, the register file has a hierarchical organization. In order to keep multiple versions of each architectural register, all TUs have its own Local Register File (LRF), and there is a Global Register File (GRF) that maintains the committed non-speculative register state. There is also a table, the Register Versioning Table (RVT), that has as many rows as logical registers and as many columns as TUs. The table acts as a directory and tracks for each architectural register which TUs have a copy of that register.

The elements that compose the MVRF architecture, the Local Register File (LRF), Register Versioning Table (RVT), the Register Validation Store (RVS), and the Global Register File (GRF), will be described in the following points.

2.5.1.1 Local Register File

The Local Register File (LRF) of each thread unit stores the architectural register values produced and consumed by the running thread. The LRF is a regular architectural register file enhanced with a few additional bits per entry. As shown in Figure 17, these bits are:

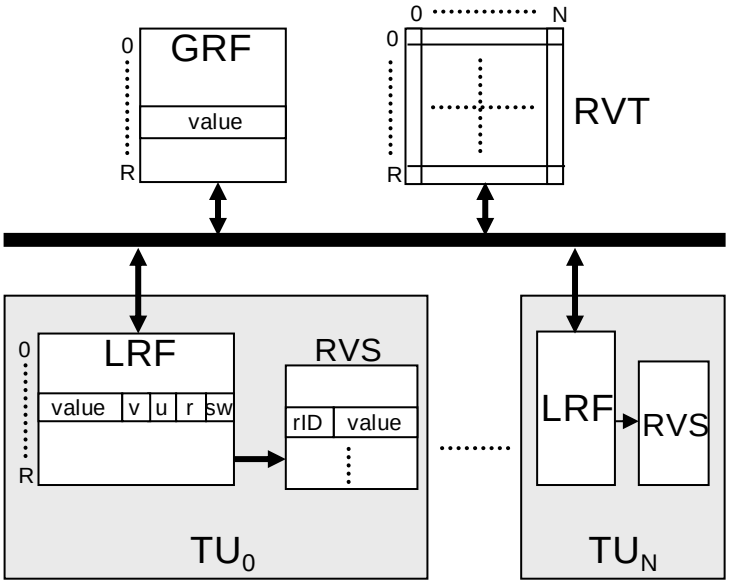


Figure 17. Block diagram of the Multi-version Register File

- *Valid bit (v)*: This bit indicates whether the register is available on the LRF or the local register is empty. On a read or write to a register the corresponding valid bit is set.
- *SliceWrite bit (sw)*: During the execution of the *p-slice*, this bit is set on the first write to the register to indicate that it has been produced by the *p-slice*.
- *Updated bit (u)*: During the execution of the thread body, this bit is set on the first write to the register to indicate that the register has already been written, and thus that a new version has been created.
- *Read bit (r)*: In case of a read to a register with the updated bit unset, this bit is set indicating that the register has been read before being written. A register with the read bit set means that it is an input value of the speculative thread, also known as a thread register live-in.

When a new speculative thread starts on a thread unit all these bits are unset indicating that no registers are available in the LRF. After spawning a new speculative thread, the register values consumed by its *p-slice* (i.e. the *p-slice* register live-ins) need to be the ones available at the *spawning point* of the parent thread. Note that *p-slices* have the characteristic that they are neither more nor less speculative than the parent thread. In fact, they execute a subset of instructions of the parent thread, so the speculation degree is the same. Therefore, the *p-slice* needs to access the same register versions, as the parent thread

would see at the *spawning point* (while executing in a different TU). In order to guarantee this, the *spawn* instruction initiates a copy operation of the *p-slice* live-in registers to the LRF of the spawned thread. These registers are identified by a mask in the *spawn* instruction that is set by the compiler when the *p-slice* is built. During the execution of the *p-slice*, a read from a register not yet copied would be stalled until the value becomes available.

Any register copied and not written within the slice (with the *sw* bit unset) is invalidated when the execution of the *p-slice* finishes because its value is potentially old. On the other hand, when a speculative thread becomes the non-speculative one all the registers produced by the *p-slice* and not consumed or overwritten by the thread (with the *sw* bit set and the *r* or *u* bits unset) are invalidated because they are no longer needed and their values are potentially wrong.

During the execution of the thread body, a read from a register not produced by the thread (or its *p-slice*) should ideally obtain its value from the closest less speculative thread that has written the register. Thus, a register read from an invalid (empty) register would cause the core to stall the reading instruction and issue the read from other core. This core is found through the Register Versioning Table (RVT). Then a register transfer request is sent to the target thread unit in order to obtain the register value. In case no other core has produced the register, the register value is transferred from the Global Register File (GRF). Following reads to the same register would obtain the value from the LRF, just as a read from a valid register would do.

An evaluation of the performance impact of the hierarchical MVRF design has been done. On average, more than 99 percent of the register accesses are satisfied from the LRF for the benchmarks evaluated (see Section 2.6.1). Thus, the average perceived latency for register access is essentially equal to the latency of the LRF, meeting the goals for the proposed register file hierarchy. When there is a miss in the LRF, most of the accesses are to the GRF, and only a few go to a remote register file. The rationale behind this is that most of the register values needed by the speculative thread are usually computed by its *p-slice*. As a result, few register transfers among cores are required. This is another side benefit of the accuracy of the *p-slice* mechanism.

Finally, in case of a thread squash all the registers in the LRF are invalidated.

2.5.1.2 Register Versioning Table

The Register Versioning Table (RVT) is a centralized table that tracks the available register versions among the different thread units. In an *N* thread unit processor with *R* logical registers in each thread unit,

the RVT is a table of *N*x*R* bits, with each column representing a thread unit and each row representing a logical register.

The RVT can be seen as a directory that contains per each architectural register all the thread units that have written to it and that thus have a version of it. This information combined with the sequential order among the running threads is used to identify for a given thread and a given register the thread unit that has produced the latest available version of the register. With this mechanism any thread unit can find from which thread unit it must request the register value in case of a read of an invalid register.

The RVT is updated using the following scheme. When a thread unit produces a new version of a register the RVT is signaled to make it visible to other threads. Thus, a register write to a previously not written register (with the *u* bit unset in the LRF), will signal the RVT to set the corresponding bit in the table.

Despite the fact that the RVT is a centralized structure, this scheme can be argued to be relatively inexpensive because of the following points:

- The number of write operations to the RVT from each thread unit and speculative thread is limited to *R* (i.e. the number of logical registers).
- The number of read operations to the RVT from each thread unit and speculative thread is also limited to *R*, but is usually much fewer because most of the input registers are produced by the *p-slice*, and thus read from the LRF.
- If a read and a write operation from two thread units happen in the same cycle, the read does not have to return the value from the writing thread unit since the reading thread unit is speculative anyway.
- Each thread unit always writes in the same column of the RVT. This may allow for banking for multiple write ports.

When a thread finishes execution or is squashed the corresponding column in the RVT is cleared. This ensures that when a new thread starts executing on a thread unit the corresponding column in the RVT will be cleared since a thread starts with an empty LRF.

2.5.1.3 Global Register File

The Global Register File (GRF) is a centralized logical register file that holds the committed non-speculative versions of the architectural registers. The purpose of the GRF is to minimize the amount of register state that has to be copied on a thread commit.

When the non-speculative thread finishes execution (it reaches the starting point of the next speculative thread), all the registers written by this thread, those with the *u* bit set in the LRF, are copied to the GRF. This is required because the next speculative thread, after being properly validated, will become the non-speculative and the current thread unit will be freed.

As previously mentioned, the GRF is accessed when a thread unit performs a read to a register not available in its LRF and no less speculative thread running has written the register, i.e. if the RVT indicates that no previous thread has written the register.

2.5.1.4 Register Validation Store

Each thread unit needs an additional buffer to store the register input values consumed by a speculative thread. The goal of this buffer is to support the register validation process needed to guarantee that the register values consumed by a speculative thread are correct, and thus that inter-thread register dependences are met. As can be seen in Figure 17, this buffer is called the Register Validation Store (RVS).

When a thread unit reads a register with the *r* bit and the *u* bit unset in the LRF (i.e. it is a thread live-in), a copy of the value along with the register identifier is stored in the RVS. In this way the RVS saves thread register live-ins, either read from another thread (not produced by the *p-slice*) or from its LRF (generated by the *p-slice*), for later validation. When this thread is validated, the values in the RVS are compared with the actual values of the corresponding registers in the predecessor thread. By doing so, it is ensured that values consumed by the speculative thread are identical to those that would have been seen in a sequential execution. Because this mechanism explicitly tracks the consumed register values, incorrect live-ins produced by the *p-slice* that are not consumed do not cause misspeculation.

Finally, when a thread finishes execution or is squashed the RVS is flushed.

2.5.1.5 Related Work

Some SpMT architectures proposed in previous research works include multi-version schemes for the architectural register file. These architectures, like Mitosis, use a distributed register file to store the different speculative states. However, since most of the designs proposed in the past used synchronization

mechanisms to deal with inter-thread register data dependences, their communication schemes among the local register files are quite different. Some examples of these architectures are the Multiscalar [9], which uses register synchronization based on compiler information, and the Synchronizing Scoreboard [44] that incorporates hardware to support the register producer-consumer synchronization. Other schemes, like the Dynamic Multithreading Processor (DMT) [3], use a full copy mechanism when a new thread is spawned in order to initialize its local register file.

None of the previous schemes propose any mechanism like the Register Versioning Table to dynamically find the last available version of an architectural register. Moreover, since the Mitosis execution model is a completely novel approach to exploit speculative thread level parallelism, none of the previous techniques can support it. In particular, this execution model demands mechanisms to support the execution of *p-slices* and the validation of register live-ins, which are not present in any previous proposal. In this way, the Multi-Version Register File (MVRF) architecture described in this thesis incorporates the mechanisms needed to support the computation of the thread register live-ins through *p-slices* and its later validation.

2.5.2 Multi-version Memory System

The memory subsystem is one of the most crucial and complex elements of any SpMT architecture. Its design may significantly impact the scalability and performance of the whole system. Similar to the register storage, to efficiently implement a SpMT execution model on current multi-core processors the memory subsystem has to be extended with support to manage speculative state, detect inter-thread data dependence violations, and commit or squash the speculative state. In addition, it has to provide support for multi-versioning; that is, allow having different versions of each memory location per thread.

In this thesis, a memory subsystem extended with all this speculation and multi-versioning support is called a Multi-version Cache (MVC). Some desirable features of the MVC are not to increase the latency of accessing the data caches, and to have a distributed organization to favor system scalability. In the literature, MVCs with a distributed design have already been proposed (see Section 1.3.2.4). They leverage well-known MSI/MESI-like cache coherence protocols, which are extended with additional state bits and control logic to implement the speculation support.

Previous MVCs schemes assume a SpMT model in which the thread ordering between two threads remains the same during the lifetime of these threads. However, this assumption does not hold in the Mitosis SpMT model due to the presence of *p-slices*. Note that in Mitosis when a new speculative thread

is spawned, the *p-slice* execution is neither more nor less speculative than its parent, whereas the speculative part of the thread is more speculative. This fact implies that in the Mitosis model a speculative thread changes its logical order with respect to the other threads during its execution. This poses some additional challenges over conventional MVCs, in correctly maintaining the memory state as needed by the *p-slice* and by the thread. Thus, like in the MVRF, specific hardware support in the MVC is required to handle *p-slice* execution.

Another important issue that has to be taken into account when designing a MVC is hardware complexity. Designing a cost-effective microarchitecture support is important for achieving good performance benefits keeping hardware complexity low.

In the Mitosis architecture, we propose two different implementations of a MVC:

1) *The Mitosis MVC*: Full-blown hardware implementation of a conventional MVC, based on the Speculative Versioning Cache (SVC) [32], extended with support for running Mitosis speculative threads.

2) *The MU[E]SLI Protocol*: A MVC based on a novel light-way extension of the MESI cache coherence protocol to implement speculation support. The proposed scheme significantly differs from the previous MVC proposal by simplifying the hardware and moving some of the required functionality to support the Mitosis SpMT execution model to the software, in an attempt to find the best trade-off between hardware and software complexity.

Both MVC designs will be described in the following sections.

2.5.2.1 The Mitosis Multi-version Cache

The Mitosis Multi-version Cache (MVC) is a novel distributed memory system with speculation and multi-versioning support. The architecture of the MVC is based on the Speculative Versioning Cache (SVC) [32], with notable extensions to handle *p-slices* and the particularities of the Mitosis execution model. In a nut-shell, the MVC supports multiple versions of memory locations (one per thread), inter-thread communication of memory data, and eager detection of inter-thread memory dependence violations.

Figure 18 shows the block diagram of the Mitosis MVC. As can be seen, each TU has its own L0 and L1 data caches, which are responsible for maintaining all speculative values, since speculative threads are not allowed to modify the main memory until they are validated and commit. All the speculation support is implemented on the L1 cache, which is extended with additional state bits, whereas the L0

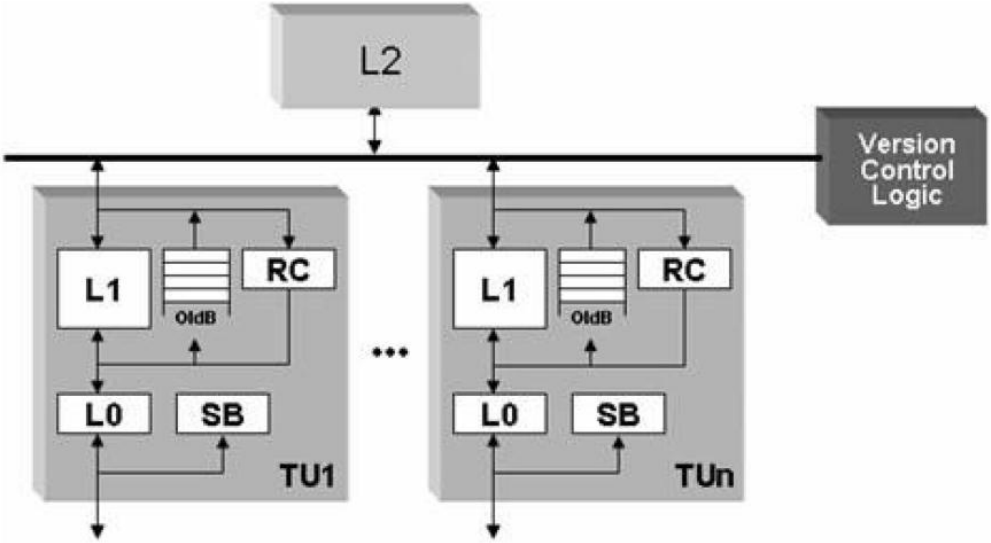


Figure 18. Block diagram of the Multi-version Memory System

cache is a conventional write-through data cache. This configuration allows having enough space in L1 for storing speculative values, without impacting much the access time to the first-level cache. Moreover, three additional structures are needed at each TU: the *Slice Buffer (SB)*, the *Old Buffer (OldB)*, and the *Replication Cache (RC)*. Finally, there is a global L2 cache shared among all the TUs that can only be updated by the non-speculative thread. Like in the SVC, there is a centralized logic called the Version Control Logic (VCL) to handle the order list of the different memory versions.

The MVC, like the original SVC [32], implements the ordering between the different versions of each memory location by storing a pointer in every cache line which points to the cache containing the next more speculative version of the data. This list of cache lines is called the Version Ordering List (VOL). The VCL implements the logic to manage the VOL. When a thread hits in the local cache, it does not need to use VOL or VCL. But on a cache miss it requests the VCL, which based on its speculative order (i.e. the order of the requesting thread with respect the other running threads) and the VOL, determines the cache that has the appropriate version of the data.

In addition, like in the SVC, the MVC enhances each L1 cache line with additional state bits to manage the speculative data. These additional bits are: *Load (L)*, *Valid (V)*, *Store (S)*, *Commit (C)*, *sTale (T)* and *Architectural (A)*. The *L* bit is set when a thread reads a cache line without first writing to it. This is done to track a potential invalidation that might happen if later on a previous thread writes to the same cache line. The *S* bit is set when a speculative thread does a store to a cache line, i.e. it becomes speculative dirty. The *C* bit is used to keep the cache lines in the local caches past the thread commits in

order to preserve data locality as well as avoiding bulk commit. When a thread commits, it sets the *C* bit in all the valid cache lines. This avoids the need to write the speculative dirty lines to the next level cache. When a thread is squashed, the cache might contain data which is the same as the non-speculative version of the data (e.g. the squashed thread might have read the value from the L2 cache). On a squash, the *A* bit is set in all the cache lines containing non-speculative data. This is done to indicate that the cache line has valid architectural version of the data. When a new thread is spawned the data in these lines are valid. The *T* bit is used to mark whether it is safe to use a committed line or not, since it might have become stale because a new version of the data has been produced.

It is important to remark that the SVC protocol is a complex yet very well designed protocol to support the SpMT execution model. In addition, since it has a distributed design, it favors system scalability, it includes techniques to reduce traffic on the inter-connection network, and it does not impact much the memory access time. These have been the main reasons for choosing the SVC as the base for the Mitosis MVC.

As previously pointed out, the Mitosis execution model poses some additional challenges to the memory system, in correctly maintaining the memory state as needed by the *p-slice* and by the thread. From the point of view of the MVC, the main requirements for supporting the execution of *p-slices* are:

- Any running thread should know whether it is executing a slice, and if any of its children are executing a slice.
- When a thread is spawned, and meanwhile the *p-slice* is being executed, the memory values at the spawning time should be visible for the slice. In order to guarantee this, when the spawner thread writes on a memory location, the previous value should be held until the spawned thread completes the execution of the *p-slice*.
- As it has been outlined in the previous point, the memory values read by a *p-slice* correspond to the values of the memory locations at the time of spawning the thread. Therefore, such values read during the slice execution must not be visible to more speculative threads and the body of the thread itself since they may be old values; that is, there may exist a more recent version.
- Before a thread commits, it is necessary to validate that the input values of the speculative thread have been correctly pre-computed. Therefore, all the memory values produced by the *p-slice* have to be stored for further validation.

In order to meet these requirements, the SVC has been enhanced with the following new components and bits:

- *Old-Buffers (OldB):* A buffer for storing the values that are overwritten by a thread while it has a child executing the *p-slice*. Its purpose is to preserve the previous memory values for the children *p-slices*. In fact, it is necessary to have as many buffers per TU as number of speculative threads that can be running their *p-slice* simultaneously.
- *Old bit (O)*: Each L1 cache line is extended with this new control bit. The values read during the execution of a *p-slice* are marked with this bit to indicate that the data is potentially old and should not be read by any other thread and discarded when the *p-slice* finishes.
- *Slice Buffer (SB)*: An additional buffer for storing the memory values computed by the *p-slice* for later validation.

These new components and state bits, and the *Replication Cache* (*RC*) as well, will be described in the following sections.

2.5.2.1.1 Old Buffers

Old buffers (*OldBs*) are used to maintain the memory values at the point a thread is spawned in order to provide the correct memory input values for the *p-slice* of the spawned thread. Figure 19(a) shows a diagram of the *OldBs* with their corresponding fields. Each thread unit has as many *OldBs* as threads may be spawned and may be run their *p-slice* simultaneously. As can be seen in the figure, each *OldB* contains *k* lines for storing the *tag* and the *data* of the overwritten memory locations.

When a new thread is spawned, a new *OldB* is allocated at the thread unit of the spawner thread. If there is not a free *OldB* the spawn is aborted. While a thread is executing, any store on a memory location writes the previous value in all the active *OldBs* of the TU. If that memory location was previously stored in an *OldB*, the new value is discarded.

When a speculative thread that is executing the *p-slice* performs a load to a memory location, it first checks for a local version at its local memory. In case of a miss, it checks in its corresponding *OldB* from the parent thread. If the value is there, then it is forwarded to the speculative thread. Otherwise, it looks for it at any less speculative thread cache. When a spawned thread finishes its *p-slice*, it sends a request to its parent thread to deallocate its corresponding *OldB*.

Note that if the non-speculative thread finishes its execution (i.e., it reaches the starting point of the least speculative thread), it may be possible that some of the threads that have been spawned by it are still

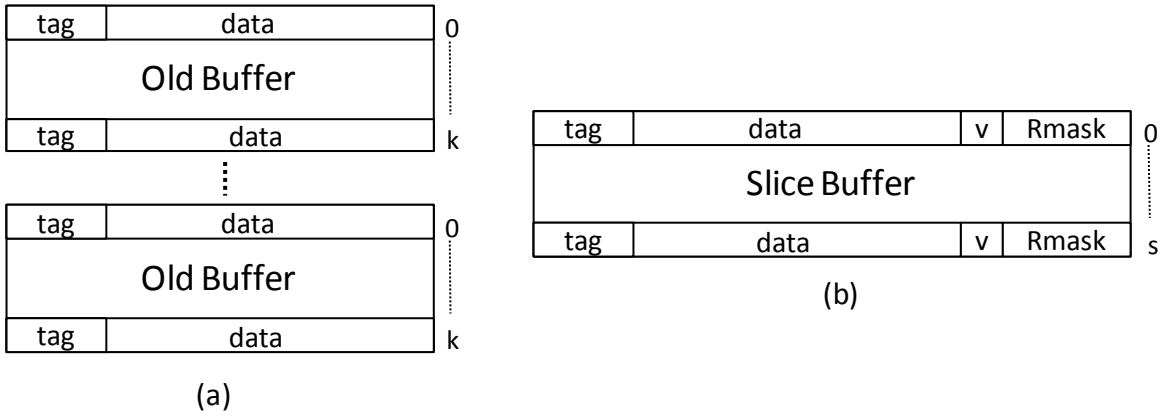


Figure 19. (a) Old Buffers and (b) Slice Buffer diagram

executing the *p-slice*. Then, those *OldBs* cannot be freed until these threads finish their corresponding slices.

2.5.2.1.2 Old Bit

The values read from the corresponding *OldB* during the execution of a *p-slice* are stored in the local caches but have to be marked in some way to avoid being read by any other thread and mistaken as state expected to be valid beyond the CQIP. To prevent a more speculative thread from incorrectly reading these values, a new bit is added to the SVC protocol, which is referred to as the *Old Bit (O)*.

When a thread that is executing the slice performs a load from the parent *OldB*, the value is inserted into the local cache with the *O* bit set, to indicate that the value is potentially old. Moreover, when a thread that is not executing the slice performs a store, for each more speculative thread that is executing the slice and has the line, the *O* bit is also set in order to get rid of all potential old lines at slice exit.

The *O* bit ensures that when a thread requests a value from a less speculative thread it does not get an old version, but the last available version. In order to do that, on a request to the VCL to obtain the last available version of a given line, those threads whose requested line has the *O* bit set are not considered.

Finally, when the *p-slice* finishes, all the cache entries with the *O* bit set are invalidated to prevent this thread and more speculative threads from reading an old value.

2.5.2.1.3 Slice Buffer

The purpose of the Slice Buffer (*SB*) is to store the values pre-computed by the *p-slice* in order to validate that the speculative thread has been executed with the correct input values. Figure 19(b) shows a diagram of the *SB* with its data fields. Each entry of the *SB* contains the *tag*, the *data* value, a read bit mask (*Rmask*), and a valid bit (*v*).

When a new speculative thread is spawned, and it is allocated on a TU, its *SB* is reset. All the stores performed during the execution of the *p-slice* go directly to the *SB*, bypassing the cache.

When the *p-slice* finishes and the speculative thread starts the execution of its body, every time a value is read from memory, the slice buffer is checked first. If the value is there, the corresponding read bit is set in the *Rmask* to indicate which thread unit has read the value, and the value is copied to the local cache of the requesting TU.

When the thread is validated, all the values that have been consumed from the *SB* have to be checked for their correctness, since these are the values that the *p-slice* may have computed wrong and have been consumed by a speculative thread. Then, those entries of the *SB* that have any read bit set are sent to the previous thread to validate their values. In case of misspeculations, the threads that have consumed the values and its successors are squashed.

Since the values stored in the *SB* may be wrong, the non-speculative thread does not access the *SB* in any case. Therefore, the *SB* is reset when the thread becomes the non-speculative one.

2.5.2.1.4 Replication Cache

The SpMT execution model can cause problems for the cache subsystem, especially when implemented on a multi-core architecture without shared L1 caches. A sequential execution would pass values from stores to loads within the single cache, whereas with speculative multithreading, it is common that the load executes in a different thread context than the store. Additionally, data that exists in the cache when the thread starts is not necessarily valid for this thread. Since this architecture is capable of exploiting parallelism from relatively small threads, if threads only exploited same-thread memory locality, cache miss rates would be extremely high. However, without modifying the memory subsystem, this is exactly what happens, and preliminary experiments confirmed this result: every newly spawned thread begins with a completely empty cold cache (see Section 2.6.2.3 for the evaluation).

The miss ratio can be reduced by introducing a stale bit in the memory protocol, as proposed in the original SVC design [32]. The stale bit marks whether committed data on a local L1 cache, pending to be propagated to main memory, is stale and thus cannot be used on a local cache access. Without the stale bit, every access to the committed data on a local L1 cache has to be treated as a miss, because the data is potentially stale. As it is shown in Section 2.6.2.3, the stale bit achieves a drastic reduction in miss ratio by allowing sharing between threads that follow each other temporally on the same core, but the miss rate is still high comparing to the sequential execution, because there is no locality between threads on different cores.

To further reduce the miss ratio, the Mitosis processor includes a Replication Cache (*RC*). This cache works as follows: On the MVC protocol, when a thread performs a store, it sends a bus request to know whether any more speculative thread has performed a load on that address in order to detect misspeculations due to memory dependence violations. Together with this request, the value is sent and it is stored in the *RC* of all the threads that are more speculative and all free TUs. Thus, when a thread performs a load, the local L1 and *RC* are checked simultaneously. If the value requested is not in L1 but in the *RC*, the value is moved to L1 and supplied to the TU.

This simple mechanism prevents the TUs from starting with cold caches and taking advantage of locality. The *RC* enables write-update coherence. Although write-update (versus write-invalidate) is less common among recent multiprocessor implementations, it is justified for two reasons: 1) the write messages piggyback messages already required in this system and 2) more importantly, in this architecture, implementing a SpMT execution model, store-load communication between cores is much higher than on a system executing conventionally parallelized code.

2.5.2.2 The MU[E]SLI Protocol

The Mitosis MVC described in the previous section is a full-blown hardware implementation of a distributed MVC, enhanced with specific support for the Mitosis SpMT execution model (i.e. mainly support for *p-slice* execution). The resulting design, although good in terms of performance (see Section 2.6.2 for the performance studies), is a pretty complex system that requires non negligible modifications to the memory subsystem and the cache coherence protocol present in current CMPs. This fact may potentially lower the chances of the Mitosis execution paradigm for being adopted in future multi-core systems.

In order to address this complexity issue, we propose building a MVC based on a light-way extension of the MESI cache coherence protocol to implement speculation support. The proposed scheme significantly differs from the previous Mitosis MVC proposal by simplifying the hardware and moving some of the required functionality to support the Mitosis SpMT execution model to the software, in an attempt to find the best trade-off between hardware and software complexity.

Thus, *MU[E]SLI* is a novel multi-version cache coherence protocol that is based on the regular MSI / MESI protocol in current multi-core processors. Its main features are:

- Speculative memory state is kept on each core's local data cache and is not propagated to upper levels until it is committed. Eviction of a speculatively modified cache line causes the squash of the speculative thread running on that core.

- Coherence activity due to non-speculative loads and stores works at line granularity (as a regular MSI / MESI protocol). For speculative loads and stores, the protocol works at word granularity to support speculative versions and reduce unneeded squashes due to false sharing.
- The protocol supports one speculative version of a given word per thread and per data cache (one speculative thread is supported per data cache).
- Inter-thread memory dependences are dynamically checked. When a read-after-write or write-after-read inter-thread dependence is detected a violation handler is invoked. The violation handler executes at application level and is responsible for squashing the speculation activity appropriately.

Note that to reduce hardware complexity, the protocol has no concept of thread order. Thus, for a speculative multithreading execution model where there is a sequential order among threads, a software layer that manages the thread ordering is required[2]. For instance, in case of an inter-thread data dependence violation, the invoked software handler is responsible for checking whether it is a read-after-write dependence before squashing the speculative thread that has performed a read and all its successors.

To implement the *MU[E]SLI* protocol, the status bits of the local data cache lines for implementing the MESI protocol [76] are extended with two additional bits per word:

- *Bit U (Unsafe)*: It is set when a store of a speculative thread writes in the corresponding word.
- *Bit L (speculative Loaded)*: Indicates that the data word has been read by the speculative thread and it was not produced by it. A speculative load sets this bit on any read word that has the *U* bit cleared.

Figure 20 depicts the design of the protocol with two state transition diagrams (responses to processor events, and responses to bus messages). Transient states have not been included for the sake of clarity. In addition some of the specific processes of the protocol (like *commit* and *squash*) are not included on the diagram also for clarity and are later discussed. State transitions are labeled following the nomenclature "events or received messages / actions and/or generated messages".

[2] Although it is out of the scope of this thesis, the *MU[E]SLI* protocol can be easily adopted to implement a Hardware Transactional Memory (HwTM) system, where there is no order among threads.

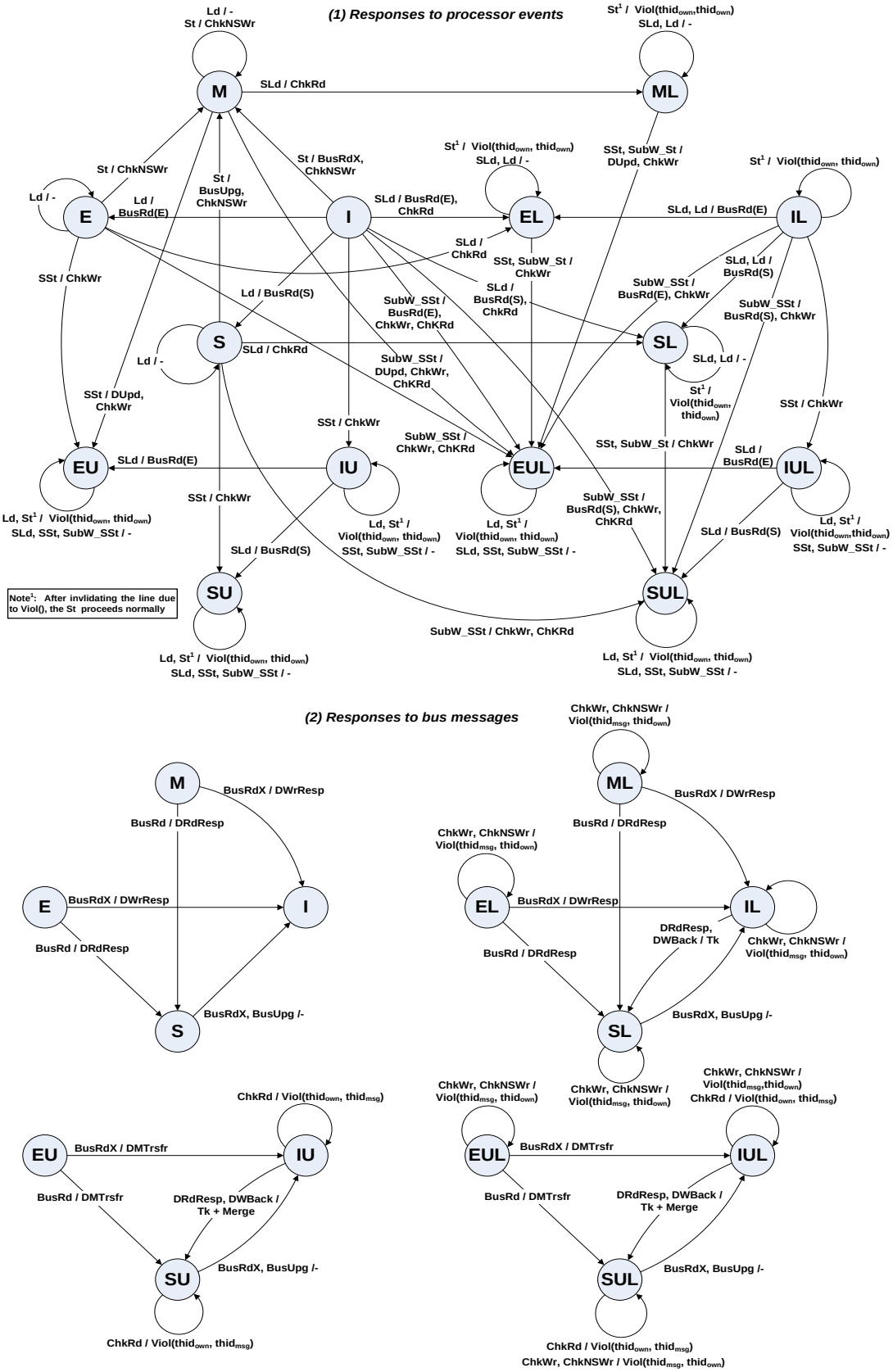


Figure 20. *MU[E]SLI* diagram states for responses to processor events (1) and bus messages (2)

State	Description
M, E, S, I	Original MESI states (per line): Modified, Exclusive, Shared, Invalid
ML, EL, SL, IL	Speculative loaded states (per word): Word speculatively loaded before produced
EU, SU, IU	Speculative stored "Unsafe" states (per word): Word speculatively written
EUL, SUL, IUL	Loaded&Unsafe states (per word): Word loaded speculatively and later written
Event	**Description**
Ld	Non-speculative load
St	Non-speculative store
SLd	Speculative load
SSt	Speculative store
SubW_SSt	Speculative store to a sub-word
Action	**Description**
Tk	Take the value from the bus and copy it to the line
Merge	Merge the new content of the line with the speculative (unsafe) values
Viol (*tid_{prod}*,* tid_{cons}*)	Call the violation handler due to a dependence violation between tid_{prod} and tid_{cons}
Message	**Description**
BusRd (@$_{line}$*)*	Bus Read – Requests the value of the line
BusRdX (@$_{line}$*)*	Bus Read eXclusive – Requests the value of the line exclusively (invalidate other copies)
BusUpg (@$_{line}$*)*	Bus Upgrade – Invalidates other copies of the line
DRdResp (@$_{line}$*)*	Data Read Response – Value of the read line
DWrResp (@$_{line}$*)*	Data Write Response – Value of the written line
DWBack (@$_{line}$*)*	Data Write Back – Value of the evicted line
RelOwner (@$_{line}$*)*	Release Ownership – Returns ownership of the line to memory
ChkRd (@$_{line}$***, word_mask)***	Check Read – Checks that no store has written to a word in the *word_mask*
ChkWr (@$_{line}$***, word_mask)***	Check Write – Checks that no load has read from a word in the *word_mask*
ChkNSWr (@$_{line}$***, word_mask)***	Check Non-Speculative Write – *ChkWr* produced by a non-speculative store
DUpd (@$_{line}$***)***	Data Update – Update memory with the value of the line without releasing ownership
DMTrsfr (@$_{line}$***)***	Data Memory Transfer – Memory acquires the onwnership of the line and send its clean copy to the bus
BusKill (tid)	Squash – Squash the speculative (unsafe) values of the thread *tid* cache
BusCommit (tid)	Commit – Commit the speculative (unsafe) values of the thread *tid* cache

Table 1. States, events, actions, and messages of the *MU[E]SLI* protocol

The states, events, actions, and messages involved on the protocol are described on Table 1 (extensions to the base MESI coherence protocol are in bold).

Two points that deserve special mention are the handling of evictions and the interaction between non-speculative load / stores and speculative ones. Evictions of non-speculative cache lines are handled as in the regular MESI protocol. However, since the speculative memory state and information status is kept locally on the data cache and cannot be propagated to upper levels until it is committed, any eviction of a speculatively accessed (read or written) cache line causes a violation, like inter-thread data dependences do (see the following section).

On the other hand, the protocol allows for the mix of speculative and non-speculative load / stores on the same local data cache. Generally, any load / store performed by a speculative thread is treated as speculative. However, the protocol contemplates the presence of special load / stores called "safe" that can be performed by any thread (speculative or not) to access to shared memory. These "safe" operations are always treated as non-speculative. Thus, some interactions could happen when speculative and non-speculative load / stores performed by the same thread access to the same locations. On that case, the protocol solves this issue raising a violation whenever a non-speculative load / store access to a speculatively accessed word location (with the *U* or *L* bit set).

On the following sections some of the key aspects, and novelties, of the protocol will be discussed. For other details of the protocol see the state transition diagrams on Figure 20.

2.5.2.2.1 Inter-thread Data Dependence Checking

An essential feature of the *MU[E]SLI* coherence protocol is the capability to dynamically keep track of the memory data words speculatively accessed and check for inter-thread data dependences in order to detect misspeculations on-the-fly.

The mechanism to dynamically check for inter-thread dependences is based on three new bus messages: *ChkRd*, *ChkWr*, and *ChkNSWr*, which are tagged with the thread id of the thread that generates the message, the address of the accessed cache line and a *word_mask* to indicate the accessed words within the line. As can be seen in Figure 20, a *ChkRd* is generated by any speculative load that accesses a word location that has not been previously accessed (read or written) by the same thread. On the other hand, the *ChkWr* message is generated by any speculative store to a word location that has not been previously written speculatively (is not unsafe). Note that check messages are only generated on the first speculative load or store to a cache word location. Finally, the *ChkNSWr* message is generated by any non-speculative store (i.e. a store produced by the non-speculative thread or a "safe" store produced by any thread).

The actual dependence checking is performed whenever a cache controller receives one of these messages (see Figure 20) by comparing (bitwise AND) the *word_mask* of the message with the *L* or *U* bits of the accessed cache line accordingly. Basically, *ChkRd* compares the message *word_mask* against *U* bits, whereas *ChkWr* and *ChkNSWr* compare it against *L* bits. In case that there is any conflict, i.e., the bitwise AND is different to zero, an inter-thread data dependence is detected and a violation is raised. This violation invokes an application handler with the thread ids of the producer and the consumer threads as parameters. When the violation is generated due to a *ChkNSWr* message, a special thread id is passed as the producer parameter to notify that the producer is non-speculative.

We assume that the application, through the violation handler, is responsible for managing the squashing of the speculation activity appropriately. On the Mitosis speculative multithreading execution model, this involves checking the order between the producer and the consumer threads to verify that the violation is caused by a read-after-write dependence and then squash the consumer thread and all its successors. An important point is that the protocol supports multiple versions of the same word location (one version per thread), so in consequence, write-after-write inter-thread data dependences do not cause any violation. However, sub-word accesses are specially handled. Any speculative store to a sub-word location implies a violation when another thread writes to the same word because cache coherence would not be guaranteed otherwise. In order to do that, any sub-word speculative store sets the *U* bit and sends a *ChkWr*, as a regular store, but also sets the *L* bit and sends a *ChkRd*. This ensures that whenever another thread performs a store to same word location an inter-thread data dependence violation will be detected.

2.5.2.2.2 The Commit Process

When the speculation activity is validated and no failure is detected, the speculative memory state has to be committed to the architectural state. This is handled by the *MU[E]SLI* commit process through a bus message called *BusCommit* that includes the thread id of the speculative thread to be committed. When the *BusCommit* is received by the cache controller corresponding to the message thread id, the speculative memory commit process is fired. This process, with the actions performed for each cache line, is summarized on Table 2 (*BusCommit* column).

The commit process resets all the *L* and *U* bits on the local data cache lines and puts in *M* state those lines that have any speculatively written (unsafe) word. In order to do that a scan of the local data cache is performed. Those lines in *SU* or *SUL* state, generate a *BusUpg* bus message in order to invalidate other copies of the cache line. In a similar way, those lines in *IU* or *IUL* state, generate a *BusRdX* message to request the architectural value of the line and locally perform the merge with the words speculatively

State	*BusCommit*		*BusKill*	
	Message	**New State**	**Message**	**New State**
IL	–	I	–	I
SL	–	S	–	S
EL	–	E	–	E
ML	–	M	–	M
IU	*BusRdX*	M	–	I
SU	*BusUpg*	M	–	I
EU	–	M	*RelOwner*	I
IUL	*BusRdX*	M	–	I
SUL	*BusUpg*	M	–	I
EUL	–	M	*RelOwner*	I

Table 2. The commit and the squash processes at the reception of a *BusCommit* or *BusKill*

written. On the other hand, lines in *EU* or *EUL* states (exclusive), shift to *M* state without generating any bus message since the line is only present on that cache.

2.5.2.2.3 The Squash Process

In case of a speculation failure, the speculative activity has to be squashed. The *MU[E]SLI* protocol is in charge of discarding all the speculative memory state produced by the squashed thread. This is implemented through a bus message called *BusKill* that includes the thread id of the speculative thread to be squashed. When the *BusKill* is received by the cache controller with thread id equal to the message thread id, the speculative memory squash process is fired. This process, with the actions performed for each cache line, is summarized on Table 2 (*BusKill* column).

Basically, this process consists on resetting all the *L* and *U* bits on the local data cache and invalidating those cache lines that have any speculative (unsafe) word. Non speculatively accessed lines are not changed. Thus, those lines in *EU*, *EUL*, *SU*, *SUL*, *IU*, or *IUL* states, shift to *I* (invalid) state, and those lines in *ML*, *EL*, *SL*, or *IL* shift to *M*, *E*, *S*, and *I* respectively. Note that lines in *EU* or *EUL* state have to release ownership of the line to memory, through a *RelOwner* bus message, when they shift to invalid.

2.5.2.3 Related Work

The generic concept of Multi-Version Cache is not new, but the particular architecture presented in this thesis is. There are several prior proposals to support multiple speculative versions of each memory location and dynamically detect inter-thread memory dependence violations. Some of the most relevant works are the Speculative Versioning Cache [32] proposed by Gopal et al., the Memory Disambiguation

Table [44] proposed by Krishnan et al., the Thread Level Data Speculation (TLDS) [92] proposed by Steffan et al. and the Hydra [33] project proposed by Hammond et al. Some of these works are based on distributed data caches and modifications of an MSI-like snoop-based cache coherence protocol, such as the designs presented in this thesis.

However, any of these previous multi-versioning schemes cannot support the execution of pre-computation slices without the type of extensions presented in the Mitosis MVC design (see Section 2.5.2.1). Even though these extensions have been implemented on top of the Speculative Versioning Cache [32], they can be included in any of the other schemes.

In addition, all these previous Multi-Version Cache designs based on MSI-like protocols have two main sources of extra complexity that are not present in our second proposal: the *MU[E]SLI* protocol. On the one hand, any speculative multithreading execution model has to deal with the concept of thread ordering in order to communicate values among threads and detect true inter-thread memory violations. In previous Multi-Version Cache proposals (including the Mitosis MVC), the thread order is directly handled by the coherence protocol, adding extra bits in the hardware and extra logic to code and manage the version order. In the *MU[E]SLI* protocol, the thread order is handled by the software (i.e. compiler generated code), who is responsible for filtering out write-after-read inter-thread memory dependences. On the other hand, to reduce hardware complexity, the *MU[E]SLI* protocol doesn't implement communication of speculative memory values among speculative threads. Not implementing this feature significantly simplifies the hardware, since it avoids a centralized version control logic, but may have a negative impact on performance for those threads with read-after-write data dependences. However, for the Mitosis speculative multithreading execution model the impact should be minimal because inter-thread data dependences are dealt with by pre-computation[3].

2.6 Experimental Evaluation

2.6.1 Framework

This section describes the infrastructure framework that has been used for conducting the performance evaluation of the Mitosis architecture. The whole architecture has been modeled in detail, including the

[3] Notice that this is a qualitative assumption based on results of the Mitosis MVC (see Section 2.6.2). Non experimental evaluation of the *MU[E]SLI* protocol has been performed yet.

Fetch, in-order issue and commit bandwidth	2 bundles (6 instructions)	Crossfeed latency	3 cycles
Pipeline Length	14 stages	Replication Cache	4-way 16 KB
Reorder Buffer Size	512 instructions	Local Register File	1 cycle
I-Cache	64KB	Global Register File	6 cycles
L0-Cache	4-way associative 16KB – hit latency: 1 cycle	Spawn/Validation overhead	5 cycles/15 cycles
L1-Cache	4-way associative 1MB – hit latency: 4 cycles	Slice Buffer	1K-entry – hit latency: 1 cycle
L2-Cache (share)	4 way associative 8 MB – hit latency: 8; miss latency: 250	Old Buffers	3 of 128-entry each

Table 3. Mitosis 4-core processor configuration

Mitosis compiler and the Mitosis multi-core processor. For the purpose of the different experiments, the Mitosis compiler has been implemented on top of the ORC compiler [38], which generates IPF (Itanium™ Processor Family) code. The performance of the Mitosis processor has been evaluated through a cycle-accurate execution-driven microarchitecture simulator based upon SMTSIM [97], configured to model a research Itanium™ multi-core processor with four TUs. Each TU is configured as a 6-way in-order issue core. The main parameters of the processor configuration are shown in Table 3. The numbers in the table are per thread unit, except for the L2 cache that is shared.

To evaluate the potential performance of the Mitosis architecture, a set of non-automatic parallelizable codes have been used. These benchmarks correspond to a subset of the Olden benchmark suite [13]. They have been chosen because they exhibit the complex interdependence patterns between prospective threads, targeted by SpMT techniques, but are small enough to be handled easily by our compiler. The benchmarks used are the following:

- *MST*: Computes the minimum spanning tree of a graph.
- *EM3D*: Simulates the propagation electro-magnetic waves in a 3D object.
- *Barnes-Hut (bh)*: Solves the N-body problem using hierarchical methods.
- *Perimeter*: Computes the perimeter of a set of quad-tree encoded raster images.
- *Health*: Simulates the Columbian health care system.

The rest of the Olden suite has not been considered due to the recursive nature of the programs. The Mitosis compiler is not currently able to extract speculative thread-level parallelism in recursive routines.

Another characteristic that makes Olden benchmarks an appealing target for Mitosis is that they are pointer intensive programs for which automatic parallel compilers are unable to extract thread-level parallelism. To corroborate this, we have compiled the Olden suite using the state-of-the-art Intel® C++ production compiler (*icc*) with the auto-parallelization flag turned on. Almost no part of the code was parallelized for any benchmark.

In all the performance studies, different input sets have been used for profiling and execution. We have used a train input set for profiling of around 10M instructions per benchmarks, and a reference input set that on average executes around 300M instructions for gathering performance results. Notice that all the statistics have been obtained for the whole execution of these programs.

The ORC compiler has been used with full optimizations enabled (-O3). For the Mitosis *p-slice* optimizations, we have considered a 5% threshold for dependence pruning and 15% for branch pruning.

Finally, it is worth to mention that we have not paid special attention to the compilation time. Our first attempt at using filters to trim the search space is very promising (see Section 2.4.1.2), but timing aspects require further work.

2.6.2 Results

In the following sections we present the results of several studies that we conducted with the objective of evaluating and characterizing the Mitosis architecture, both hardware and software components (i.e. compiler and microarchitecture).

2.6.2.1 Speculative Threads Characterization

The first thing we studied is the characteristics of the speculative threads the Mitosis compiler generated for the Olden benchmarks. Table 4 presents some statistics corresponding to this characterization. The last row shows the arithmetic mean for the evaluated benchmarks.

The second column shows the number of spawned threads by benchmark, and the third column shows the average number of instructions executed by the speculative threads. It can be observed that *bh* spawns the fewest number of threads, but the average size is about 30 times larger than for the rest of the benchmarks. On the other hand, *mst* spawns the most, but the average size is the lowest. This fact shows that the Mitosis architecture is able to effectively extract and exploit TLP at different granularities.

The fourth column shows the average dynamic size of the *p-slices*, and the fifth column shows the relationship between the sizes of the speculative threads and their corresponding *p-slice*. This percentage

OLDEN	Spawned Threads	Thread Size	Slice Size	Slice / Thread	Thread Live-ins	Squash %
bh	422	15543	196	1.3%	4.4	0.7%
em3d	396638	422	9	2.1%	1.0	0.3%
health	198497	1112	41	3.7%	2.7	26.9%
mst	1367114	271	5	2.1%	2.3	0.8%
perimeter	493725	576	24	4.2%	3.6	1.0%
MEAN	**491279**	**3585**	**55**	**2.7%**	**2.8**	**6.0%**

Table 4. Characterization of the speculative threads extracted from the Olden benchmarks

is consistently quite low for all the studied benchmarks and, is less than 3% on average. The low *p-slice* overhead comes from three sources: (i) the careful choice of SP-CQIP pairs, (ii) the elimination of all unnecessary computation, (iii) and the aggressive (and sometimes unsafe) optimizations performed by the Mitosis compiler. In the following Section 2.6.2.2 we present a characterization of these *p-slice* optimizations.

The sixth column shows the average number of thread input values that are computed by the *p-slice*; that is, on average it is only necessary to compute three values to execute the speculative threads. This supports the fundamental hypothesis that irregular programs contain parallel regions of code not easily detectable with conventional compiler techniques.

Finally, the rightmost column represents the average percentage of threads that are squashed. For most of the benchmarks, this percentage is rather low, except for *health*, where almost one of every four threads is squashed. We have observed that for this benchmark, memory dependences for the train and reference input sets are significantly different, which result in inaccurate profiling information. This fact introduces many thread squashes due to inter-thread memory dependence violations.

2.6.2.2 P-slice Optimizations Characterization

Using the Olden benchmarks, we have characterized the benefit of the proposed optimizations for the *p-slices*. For doing that, we have used a metric that we call *average benefit per pair*. It is an approximation of the number of parallelized instructions by each instance of the pair. The expected benefit of a single pair is computed as follows:

```
Overlap = PairLength – (SliceLength + Init)
ProbCorrect = (1-Cancel) * (1-Misspec)
Benefit = Overlap * Count * ProbCorrect
```

`PairLength` and `SliceLength` show the average length of the pair and the slice, respectively. `Init` represents the latency of the *spawn* instruction. `Cancel` shows the probability that the *p-slice* is cancelled, and `Misspec` is the probability that the slice is incorrect due to speculative optimizations. Finally, `Count` is the number of times the spawning instruction is executed. From those expressions, the *average benefit per pair* is computed as:

$$AvgBenefitPerPair = SUM(Benefit_i) \,/\, SUM(Count_i), \text{ for all } PAIR_i$$

To quantify the effect of the different optimizations applied to *p-slices*, Table 5 shows the *average benefit per pair* for all pairs after filtering, that is, the set of candidate pairs that will be later considered by the pair selection scheme (see Section 2.4.1.2). We show in the different columns the proposed metric without any speculative optimization (*DFL*), after dependence pruning (*DPR*), after branch pruning (*BPR*) and finally after cancel elimination (*CCL*). Each optimization is added on top of previous ones.

We can observe that all optimizations significantly improve the quality of *p-slices*, with the exception of branch pruning. It is important to notice that this result does not imply that branch pruning is not an effective optimization. On the first hand, in that case the pruned infrequent paths do not bring any extra benefit on *p-slice* length reduction with respect to dependence pruning because these instructions had already been eliminated with this optimization. On the second hand, branch pruning enables cancel elimination, which is shown to be quite effective. The rationale behind this is that a significant part of a *p-slice* is introduced to resolve the control path. Eliminating *cancel* instructions and thus all the computation that leads to the associated branch resolves in important *p-slice* length reductions.

Figure 21 shows the estimated speed-up using the model implemented in the Mitosis compiler (see Section 2.4.1.3.1) for the different optimizations. As in the previous table, each optimization is applied on top of the previous one. We can observe that the improvement in the *p-slice* overheads shown in Table 5

Default	Dependence pruning	Branch pruning	Cancel elimination
1.9	106.5	106.5	287.6

Table 5. Benefit of *p-slice* optimizations on all candidate pairs

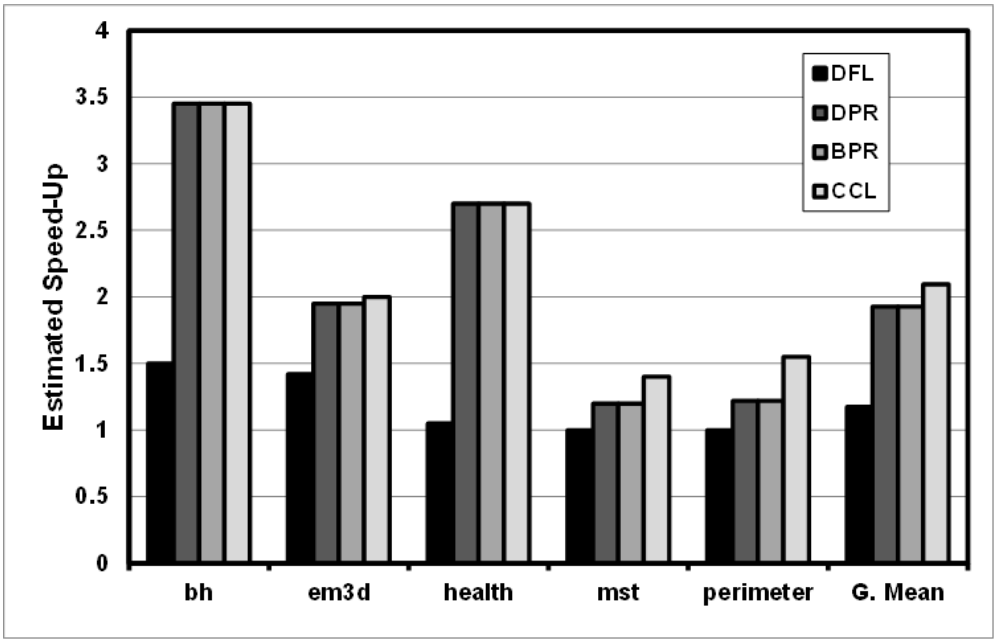


Figure 21. Total speed-up estimated by the Mitosis compiler for the proposed optimizations

actually translate into expected speed-up. On average, the expected speed-up grows from 1.17 without optimizations up to almost 2.1 when all optimizations are set. An important remark is that, in terms of estimated speed-up, for some benchmarks cancel elimination does not contribute as much as it was expected by the results in Table 5. The reason is that for many pairs the high reduction of *p-slice* length provided by cancel elimination comes at expenses of significantly increasing the number of misspeculated threads. This effect is taken into account by the selection algorithm, which finally select those pairs that have the best tradeoff between *p-slice* length and misspeculation probability, and thus provide the biggest benefit.

The *p-slice* optimization results presented in this section shows how important is to properly optimize *p-slices* in order to reduce their overhead and the big effect these optimizations have on the overall performance of the Mitosis architecture.

2.6.2.3 MVC Characterization

As described in Section 2.5.2.1, the memory subsystem is a key component of any SpMT architecture and its design has to address different challenges. One of these challenges is the ability to exploit cache locality among different threads, running either on the same TU or different TUs, in order to achieve a cache behavior (i.e. miss rate) comparable to a sequential execution.

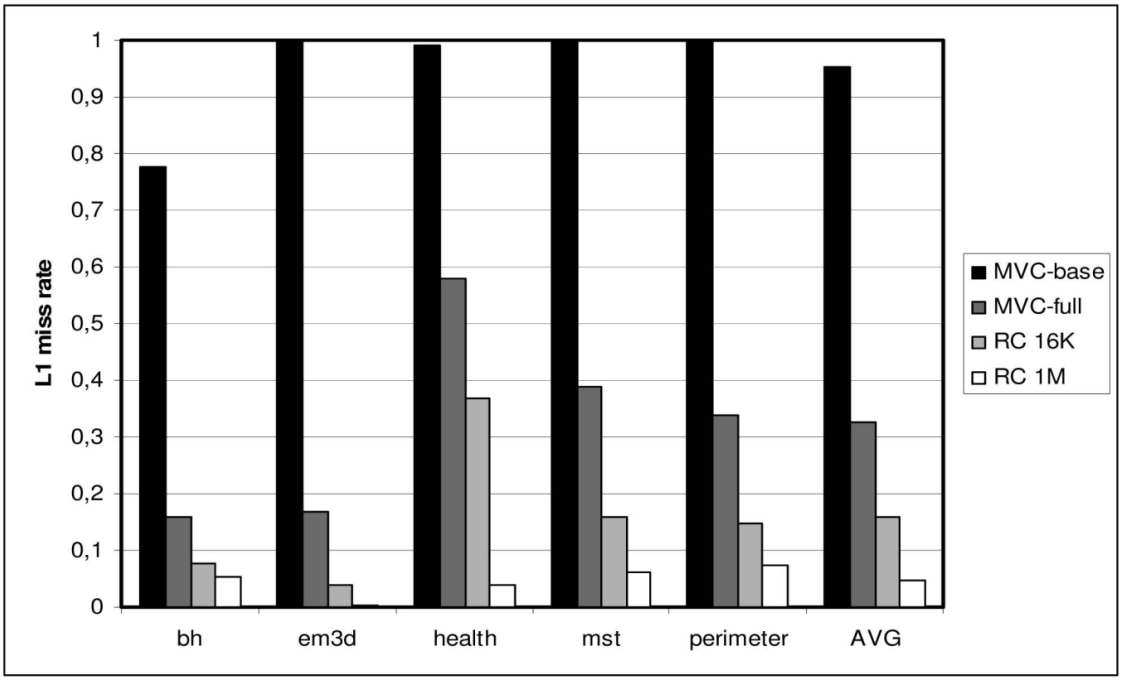


Figure 22. Multi-version Cache miss rate for the base protocol, the full protocol, and the RC

We have evaluated the cache miss rate of the Multi-version Cache (MVC) implemented in the Mitosis architecture (L1 cache). Several configurations presented in Section 2.5.2.1.4 have been evaluated. The results of this study for a subset of the Olden benchmarks are shown in Figure 22. The processor configuration and memory subsystem used in this study is shown in Table 3.

The leftmost bar corresponds to the baseline implementation of the MVC, which has not been included in the Mitosis microarchitecture and it is shown here to motivate the adoption of the full design (second bar). As can be seen, the cache miss rate is extremely high (above 95% on average) for the baseline configuration. This is due to the fact that the MVC baseline protocol does not include any technique to exploit data locality among threads running on different cores, neither among consecutive threads that run on the same core. Since this architecture is capable of exploiting parallelism from relatively small threads (see threads characterization on Section 2.6.2.1), if threads only exploit same-thread memory locality, cache miss rates are extremely high. However, without modifying the memory subsystem, this is exactly what happens, and the experiment confirms this result: every newly spawned thread begins with a completely empty cold cache.

The second bar shows the results for the full version of the MVC, which is the design implemented in the Mitosis architecture. The full design introduces a *stale bit* in the memory protocol, as proposed in

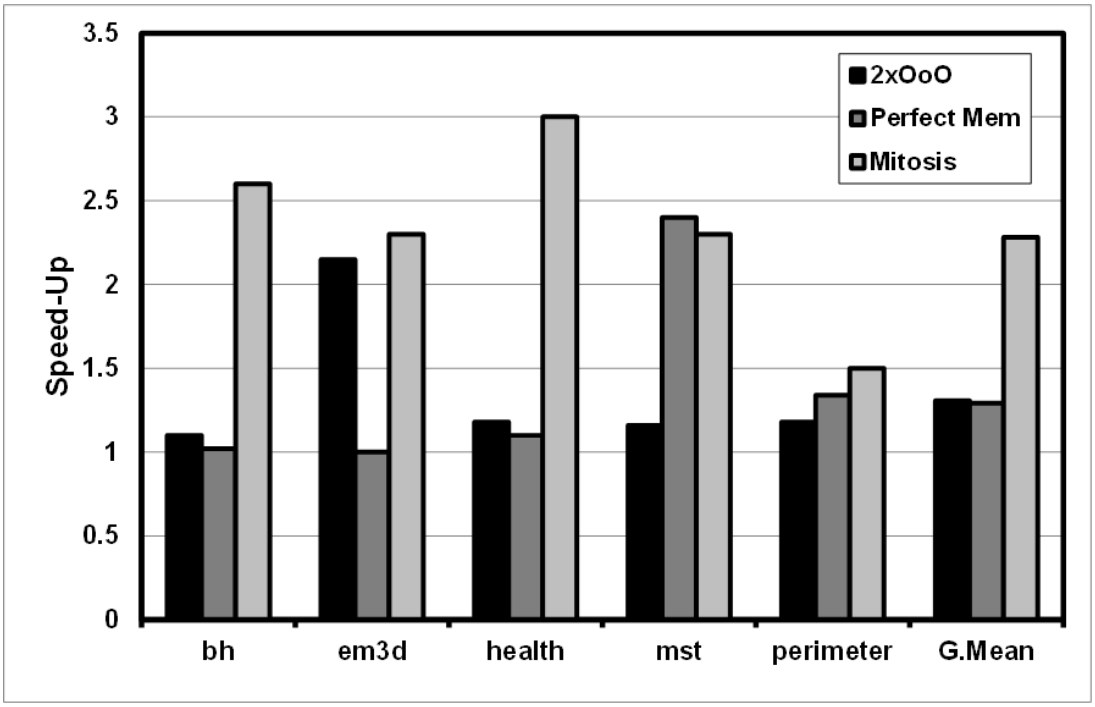


Figure 23. Speed-up of the Mitosis architecture over single-thread execution

the original SVC design [32]. The *stale bit* marks whether committed data on a local L1 cache, pending to be propagated to main memory, is stale and thus cannot be used on a local cache access. Without the *stale bit*, every access to the committed data on a local L1 cache has to be treated as a miss, because the data is potentially stale. As it is shown in Figure 22, the stale bit achieves a drastic reduction in the miss ratio (to about 30% on average) by allowing sharing between threads that follow each other temporally on the same core, but the miss rate is still high comparing to the sequential execution, because there is no way to exploit locality between threads on different cores.

To further reduce the miss ratio, the Mitosis MVC includes a Replication Cache (*RC*). The miss ratio for two configurations of the *RC* is shown in the rightmost bars. It can be observed that the use of a four-way 16-Kbyte *RC* significantly reduces the miss ratio to 15% on the average. As it is shown, further reduction in the miss rate can be achieved by a larger 1-Mbyte *RC*.

2.6.2.4 Performance Characterization

We have characterized the performance of the Mitosis architecture using the Olden benchmarks. Figure 23 shows the speed-ups of the Mitosis 4-core processor over a superscalar in-order processor with no speculative threading support and about the same resources as one Mitosis TU. For comparison, we also show the speed-up of a more aggressive processor, with twice the amount of resources (functional units),

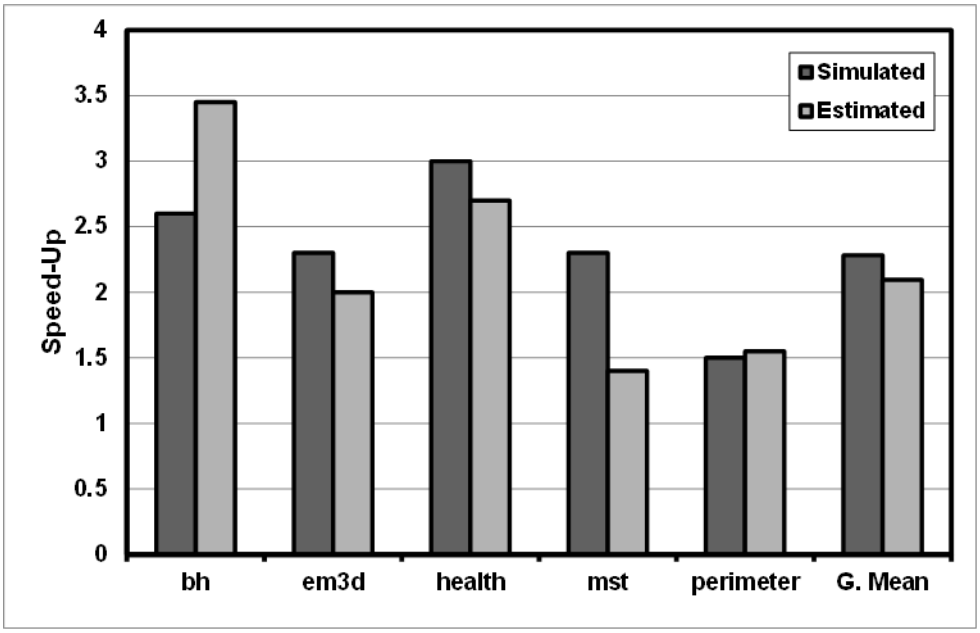


Figure 24. Simulated versus estimated speed-up

twice the superscalar width and out-of-order issue (with no speculative threading support), and a processor with a perfect first-level cache (an aggressive upper limit to the performance of Helper Threads [19] that target cache misses).

It can be observed that the Mitosis processor achieves an average speed-up close to 2.3 over single-threaded execution, whereas the rest of the configurations provide a much lower performance. Perfect memory achieves a speed-up of just 1.29, and the more aggressive out-of-order processor only provides a 1.31 speed-up. These results show that when the lack of execution resources are the bottleneck (*em3d*, with high instruction-level parallelism), Mitosis alleviates this problem as effectively as a much more complex single-core by spreading the computation among multiple cores. When memory latency is the bottleneck (*mst*, with L1 miss ratio over 70%), the ability to hide memory latencies in Mitosis (speculative threads do not stall waiting for load misses on other cores) gives the performance of a perfect cache. In summary, we find Mitosis mirrors an aggressive superscalar when the ILP is high, an unattainable memory subsystem when memory parallelism is high, and outperforms both when neither ILP nor memory parallelism is high.

Figure 24 shows the speed-up estimated by the compiler model, with full optimizations for the selected pairs, compared to those obtained with the simulator. In three of the five benchmarks (*em3d*, *health* and *perimeter*) the speed-up predicted by the model is relatively close to that of the simulation. In

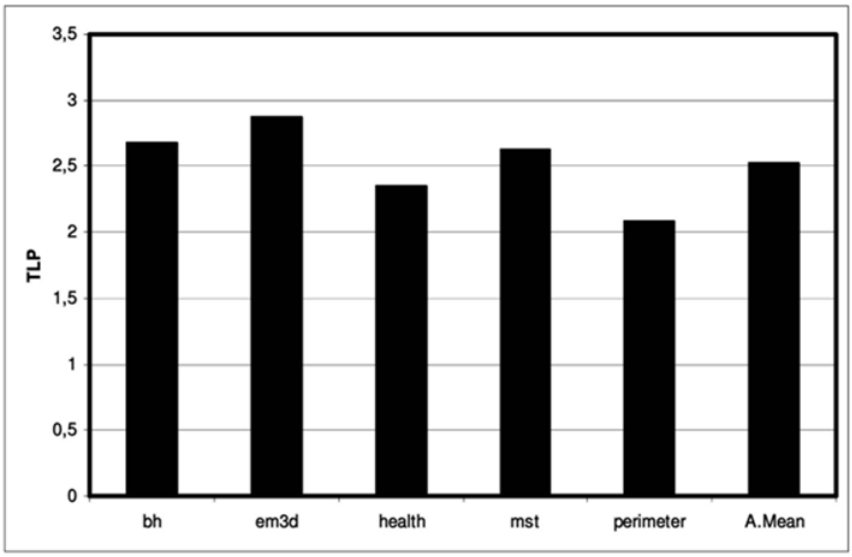


Figure 25. Average number of active threads per cycle

the case of *mst* we have observed that the difference is due to many high-latency instructions. Including a more accurate latency for each instruction in the model (instead of the fixed 1-cycle currently assumed) would significantly improve performance in these cases. In the case of *bh*, the main source of discrepancy between simulated and estimated speed-ups are due to the use of average lengths to estimate the timing of the *p-slices* and speculative threads. We have observed that for this program, these lengths experience a significant variability, and thus, the selected threading scheme is not optimal for the cases that significantly depart from the average.

Figure 25 shows the degree of speculative TLP that is exploited in the Mitosis processor. It can be observed that even though parallel compilers are unable to find TLP in these benchmarks, there is, in fact, a high degree of TLP. On average, the number of active threads per cycle that perform useful work is around 2.5. We have also measured that, on average, 1.2 TUs are idle due to the lack of threads.

Figure 26 shows the active time breakdown for the execution of the different benchmarks in the Mitosis processor. As expected, the most of the time the TUs are executing useful work (the sum of the non-speculative and the speculative execution). On the average, this is nearly 80% of the time that the TUs are working and higher than 90% for *bh* and *em3d*. Notice that the overhead added by this execution model represents less than 20% for these benchmarks. The most significant part of it comes from the wait time. This time stands for the time that a TU has finished the execution of a speculative thread but it has to wait until becoming non-speculative to commit. The other components of the overhead are the *p-slice* execution, initialization, validation, and commit overhead. It is worth noting that the overhead of the

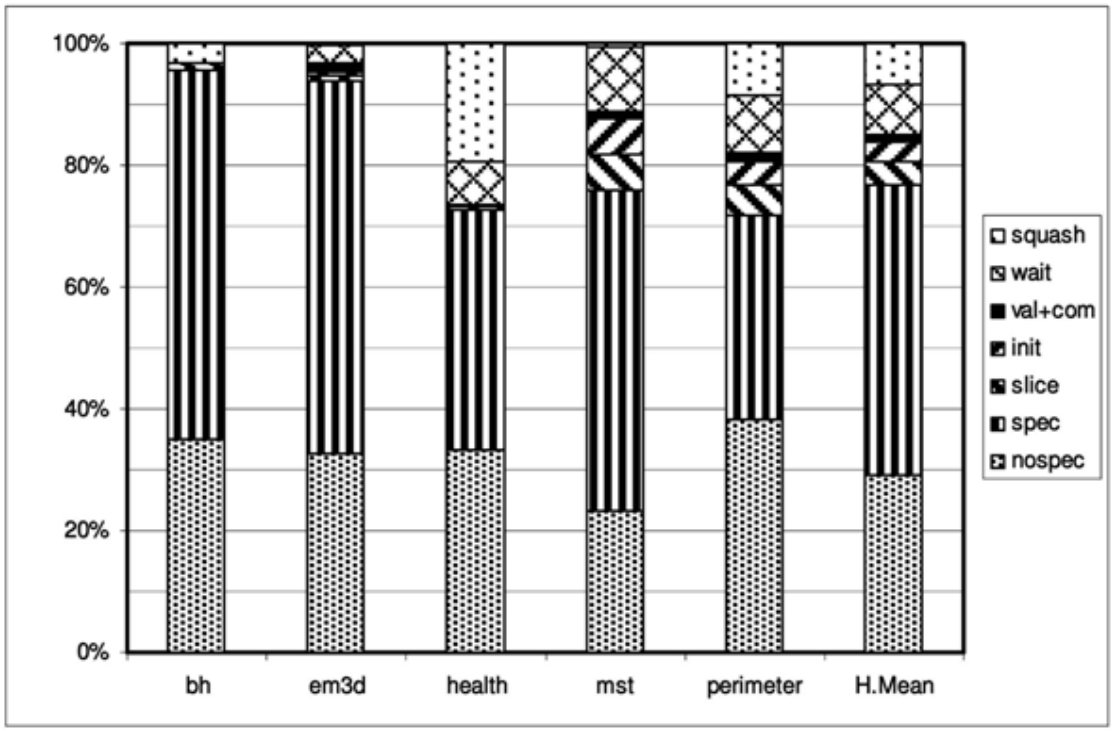


Figure 26. Time breakdown for the Mitosis processor

slices only amounts to 4% for these benchmarks. Finally, the top of the bars shows the average time that TUs are executing incorrect work, i.e. executing instructions in threads that are later squashed. This percentage is only 8% overall, mostly due to *health*, where the overhead is almost 20%. In this case, most of the squashes are due to memory violations, as previously pointed out (see Section 2.6.2.1), and the cascading effect of the squashing mechanism. Recall, however, that *health* still maintains a three-time speed-up, despite these squashes.

These results show the effectiveness of the *p-slice* based Speculative Multithreading execution model introduced in this thesis in meeting the Mitosis design goals: high performance, resulting from high parallelism (low wait time), high spawn accuracy (very low squash rates), and low spawn and prediction overhead (very low *p-slice* overheads).

2.7 Conclusions

In this chapter we have presented and evaluated the Mitosis architecture, a hybrid software/hardware platform, which exploits speculative TLP through a novel software value prediction scheme based on pre-computation to manage inter-thread data dependences. It does so by inserting a piece of code at the

beginning of each speculative thread that speculatively computes the thread input values. This code, referred to as *p-slice*, is built from a subset of the code after the *spawn* instruction and before the beginning of the speculative thread. A key feature of this scheme is that *p-slices* do not need to be correct, which allows the Mitosis compiler to use aggressive optimizations when generating them.

We have described the whole Mitosis architecture. On the software side, we have presented the Mitosis compiler with special emphasis on its two major novelties: (1) the use of pre-computation slices to handle inter-thread data dependences and the set of aggressive optimizations to reduce the overhead, and (2) a powerful thread selection scheme, not constrained to program structures, that uses a model of the whole system to identify which parts of the program will provide the highest benefit when parallelized through speculative threads. This model takes into account possible misspeculations, overheads, and load balancing.

On the hardware side, the key microarchitecture components of the Mitosis processor have been presented: (1) a novel multi-version register file organization that supports a unified global register view, multiple versions of register values, transparent communication of register dependences across processor cores, all with no significant latency increase over traditional register files, and (2) different proposals of a multi-version memory subsystem (MVC and *MU[E]SLI*) to manage the speculative memory state and inter-threads dependences. In addition, we have introduced to the MVC a replication scheme to effectively mimic the temporal locality available to a single-threaded processor with a single cache.

Thanks to the combination of all these hardware and software techniques, the results obtained by the Mitosis architecture with four TUs for a subset of the Olden benchmarks show significant performance improvement over single-thread execution. It outperforms the single-threaded execution by 2.3 times and provides more than a 1.75 speed-up over a double-sized out-of-order processor. A similar speed-up was achieved over a processor with perfect memory. These results confirm that there are large amounts of available TLP for code that is resistant to conventional parallelism techniques. However, this parallelism requires highly accurate dependence prediction and efficient data communication between threads, as provided by the Mitosis architecture.

One of the crucial design points in Speculative Multithreading architectures is the kind of threads extracted from the sequential applications and the parallelization coverage achieved. In the Mitosis architecture we have proposed a thread partitioning model that is able to place *spawning pairs* in any point of the program and identify the most effective points to spawn speculative threads. Though the proposed selection mechanism might give us the best points to spawn speculative threads, the exploited

parallelism may be constrained by the characteristics of the threads. Specifically, speculative threads are chunks of consecutive instructions and decomposing an application into these coarse-grain threads may not be the best approach to exploit parallelism on irregular applications. In fact, on the evaluation of the Mitosis architecture we have seen that one of the main sources of overhead is the *wait* time produced by workload imbalance (see Figure 26 on Section 2.6.2.4). Moreover, better parallelization coverage would provide fewer idle TUs and better performance (see Figure 25 on Section 2.6.2.4). Many hard to parallelize codes require fine-grain thread decomposition techniques in order to extract speculative threads that adapt to the available parallelism, and have good workload balance with few inter-thread dependences.

In the next chapter we intend to look deeper into these issues and explore mechanisms to create threads using a fine-grain thread decomposition scheme. We propose a new speculative threading architecture, called Anaphase, which addresses the main constraints of the Mitosis architecture.

Chapter 3

The Anaphase Architecture

The Anaphase architecture is a novel hybrid software / hardware SpMT system that combines the best of pure Speculative Multithreading schemes, like Mitosis, and the concept of fusing cores introduced in adaptive CMP architectures, like Core Fusion. The main distinguishing feature of Anaphase is its unique thread decomposition algorithm that decomposes an application into threads at instruction granularity. The flexibility that this fine-grain thread decomposition provides contributes to effectively exploit more ILP, TLP and Memory Level Parallelism (MLP) on irregular applications than conventional thread partitioning schemes.

This chapter presents in detail the Anaphase architecture. On the software side, the Anaphase thread decomposition algorithm, with all the partitioning heuristics and mechanisms to deal with inter-thread dependences, is described. On the hardware side, the cost-effective hardware support for running Anaphase threads on top of a conventional multi-core processor is presented. Finally, several studies for evaluating the Anaphase architecture and the conclusions extracted from them are presented.

3.1 Introduction

In the previous chapter we investigated the Mitosis [30][54] execution paradigm to boost sequential applications on multi-core platforms. In this scheme inter-thread dependences are managed through a software value prediction technique based on pre-computation slices (*p-slices*). We have seen that accurately predicting inter-thread dependences without incurring in large overheads is crucial for achieving high performance on SpMT architectures. Although Mitosis provides this thanks to the use of *p-slices*, there are other fundamental aspects that may constraint the exploitable parallelism and the performance of the SpMT system.

One of these constraints is the kind of threads that are extracted. In traditional SpMT architectures, like Mitosis, threads are chunks of consecutive instructions which might not be optimal for performance because they might have large number of dependences among them. Mitosis alleviates this problem thanks to the thread selection scheme that is able to identify the most effective points of the program to spawn speculative threads, taking into account the overhead introduced by inter-thread dependences among other factors. Though the proposed selection mechanism might give us the best points to spawn speculative threads, the coarse-grain nature of the threads may ultimately harm the parallelization coverage and the achievable performance. This is especially serious on irregular applications, where finding large chunks of consecutive instructions with good workload balance, good coverage, and few dependences among them is not always possible.

Moreover, the performance of conventional SpMT systems is highly dependent on the Thread Level Parallelism (TLP) available in the program. However, for many sequential applications the amount of exploitable TLP is limited or may vary significantly over different regions of code. For example, in some regions of the program there might not be enough TLP available to keep the cores busy, but there might be more Instruction Level Parallelism (ILP) than what a single core can exploit. Conventional SpMT schemes do not take advantage of this; in such cases the additional cores would stay idle, thereby wasting the resources as well as opportunities for performance improvement. Researchers have investigated adaptive multi-core designs, like Core Fusion [36], which are inspired from Clustered architectures [11]. In these proposals the resources of a multi-core chip dynamically adapt to exploit the available ILP in the sequential programs. Since work is assigned to the cores at instruction granularities, they can use the individual cores together to give the impression of a wider core which can exploit the additional ILP. Though just like TLP, the ILP characteristics of the programs also vary over code regions. When the program does not have enough ILP, under these schemes the cores stay idle, even though there

might be TLP available. The reason these proposals do not exploit TLP is that they do not use speculation at large distances like SpMT schemes do.

These limitations of conventional SpMT schemes and adaptive multi-core architectures motivated the work presented in this chapter. We propose Anaphase [55][56], which is a hybrid software/hardware architecture that combines the concepts of Speculative Multithreading and of adaptive multi-core architectures. Anaphase takes the best of both worlds, doing thread partitioning at a fine granularity like adaptive multi-core architectures, and at the same time exploiting speculative TLP like conventional Speculative Multithreading schemes.

The main distinguishing feature of this novel SpMT scheme resides in its unique and powerful thread decomposition algorithm that decomposes an application into speculative threads at instruction granularity. The flexibility to assign individual instructions to threads allows that the resulting speculative threads adapt to the available parallelism of the application. For example, in case of abundant ILP, Anaphase may create a partition where near instructions are interleaved among different threads. On the other hand, if TLP is abundant, Anaphase may distribute instructions that are far away in the dynamic stream into different threads. This fine-grain thread decomposition algorithm is based on a multi-level graph partitioning technique implemented in the Anaphase compiler that includes specific and smart heuristics to guide the partition. Heuristics have been designed to target thread partitions that provide the following benefits: (i) minimize the dependences among the resulting threads, (ii) improve the workload balance, and (ii) increase the amount of MLP that can be exploited by taking into account how delinquent loads and their consumers are distributed among threads.

As we have seen in the previous chapter, handling inter-thread dependences with good accuracy and low overhead is crucial to achieve high performance on SpMT systems. In that sense, the Anaphase thread decomposition scheme incorporates different techniques to deal with inter-thread dependences and a mechanism to statically combine and select the most appropriate technique for each case. Following what we learned on the Mitosis architecture, Anaphase implements a flavor of pre-computation slices (*p-slices*) to deal with data and control dependences. Thanks to the fine-grain thread decomposition, in conjunction with the use of *p-slices* inherited from the Mitosis paradigm, Anaphase is able to generate speculative threads that effectively exploit ILP, TLP and Memory Level Parallelism (MLP).

On the hardware side, the Anaphase architecture implements a cost-effective hardware support on top of a conventional multi-core processor that allows the execution of Anaphase threads with low hardware complexity. It presents several novel and unique components that are in charge of: (i) reconstructing the

original sequential order of instructions that have been arbitrarily assigned to speculative threads, (ii) managing the speculative and register memory state, and (iii) performing checkpointing with very little overhead. In particular, the Anaphase multi-core processor implements a novel module placed in the uncore, named Inter-Core Memory Coherency Module (ICMC), to orchestrate all the aforementioned tasks with very little interference with the cores.

Overall, the Anaphase software/hardware architecture has been designed with the necessary elements and techniques to overcome the constraints of previous SpMT systems and adaptive multi-core architectures. As we will show in this chapter, performance results for the Anaphase architecture confirm that it effectively boosts sequential and irregular applications on current and future multi-core processors.

The rest of the chapter is organized as follows: In Section 3.2 and Section 3.3 we describe the execution and the threading model used in Anaphase. In Section 3.4 we describe the compiler and the thread decomposition algorithm, a multi-level graph partitioning technique, with special emphasis on the heuristics used to guide the partition. Section 3.5 presents the multi-core processor and the extensions proposed to support the execution of Anaphase threads. In particular, we describe the mechanisms to reconstruct the sequential order, manage the memory and the register state, and perform checkpointing. Section 3.6 gives details of the experimental framework used to analyze the proposals made in this work and it presents the results of the conducted studies. Finally, we conclude this chapter in Section 3.7.

3.2 The Anaphase Execution Model

From ten thousand feet, Anaphase is like other SpMT systems. It takes a serial program, it parallelizes it at compile time with an automatic parallelizer and then run the resulting program on a CMP extended with specific hardware support for running speculative threads. As a result, the performance of a single-thread application is boosted thanks to its decomposition into multiple threads that run in parallel using multiple cores.

One thing to notice is that in this scheme Anaphase threads run inside a tile and the rest of the system is not aware of any speculative activity that happens inside it. In fact, from the point of view of the cores outside the tile, the tile behaves like a single-core system running a single-thread application. As a result, we could say that on that case we are virtually fusing the cores inside the tile. Figure 27 shows a high level scheme of an Anaphase system assuming two cores per tile.

The Anaphase SpMT system works as follows. The Anaphase compiler detects that a particular region *B* is suitable for applying speculative multithreading. Hence it decomposes *B* into speculative

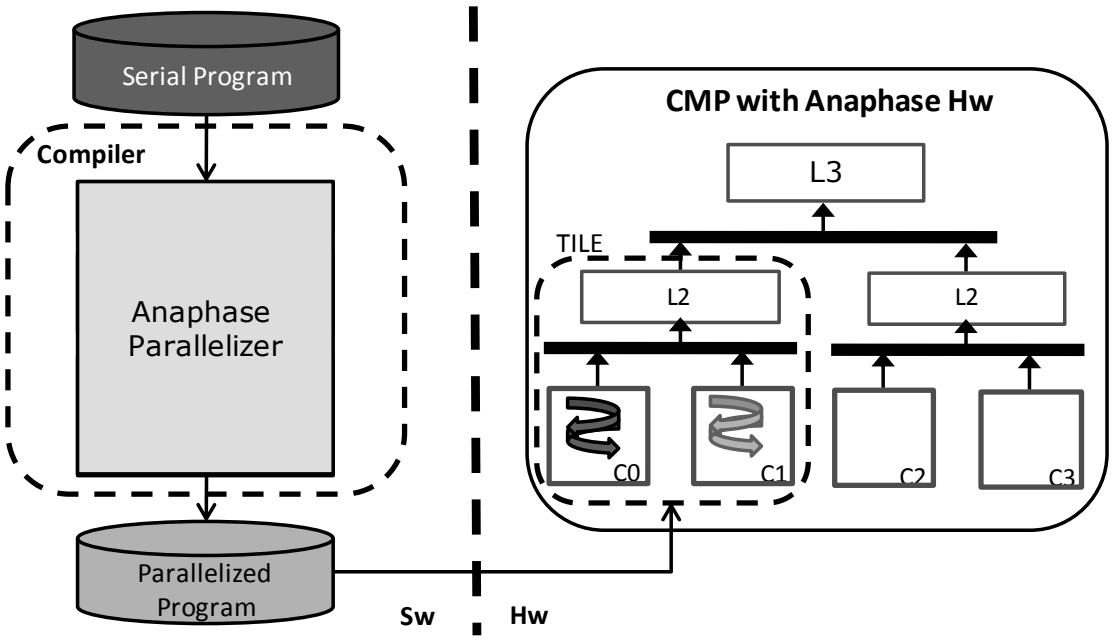


Figure 27. High-level overview of an Anaphase system comprised by 2-core tiles

threads that are mapped somewhere else in the application address space. We refer to this version of *B* as the optimized version (see Figure 28 for an example of a region of code decomposed into two threads).

A *spawn* instruction is inserted in the original code before entering region *B*. Such a spawn operation creates the new threads, and all speculative threads start executing the optimized version of the code. All the speculative threads execute in *cooperative* mode within a tile. For simplicity, we assume that when a *spawn* instruction is executed, all the other cores in the same tile are idle. Otherwise, the *spawn* instruction is handled as a *nop* and the spawn operation is not performed.

Violations, exceptions and/or interrupts may occur while in *cooperative* mode, and the speculative threads may need to be cancelled. In order to properly handle these scenarios in *cooperative* mode, each speculative thread performs partial checkpoints regularly as discussed in Section 3.5.3 and complete checkpoints are performed by the thread executing the oldest instructions. When the hardware detects a violation, an exception or an interrupt, the execution is redirected to a code sequence referred to as the rollback code (see Figure 28). This code is responsible to roll back the state to the last completed checkpoint and to resume execution from that point on in *independent* mode by redirecting the *spawner* thread to the appropriate location in the original version of the code. Then, *cooperative* mode will restart when a new *spawn* instruction is encountered.

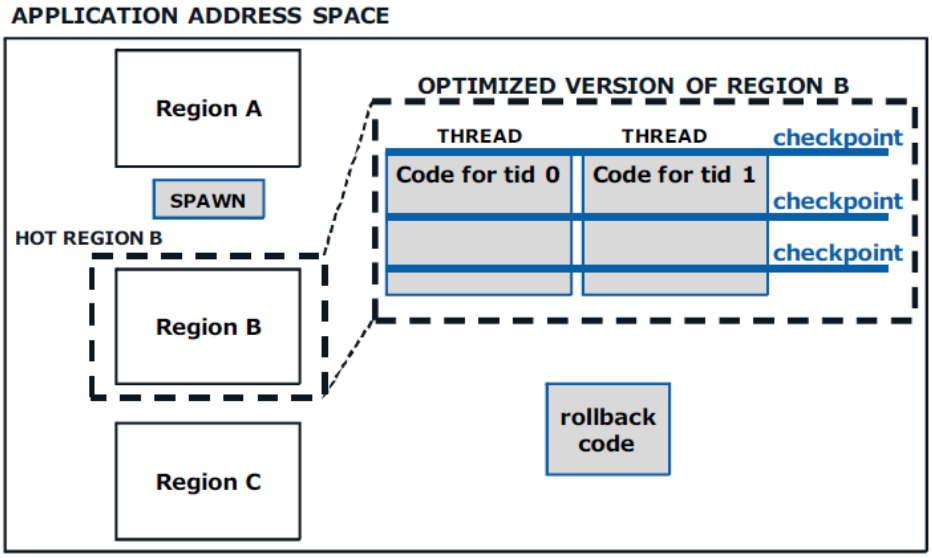


Figure 28. Example of a region decomposed into two Anaphase threads

Although checkpoints are drawn as synchronization points in Figure 28, this is just for clarity purposes. A speculative thread that goes through a checkpoint does not wait to the other speculative thread to arrive to that point, as discussed in Section 3.5.3. When all speculative threads complete, they synchronize to exit the optimized region, the speculative state becomes non-speculative, the execution continues with one single thread, and the tile resumes to *independent* mode.

It is important to remark again that the main distinguishing features of Anaphase with respect to other SpMT schemes are the threading model, the decomposition algorithm and the hardware support. These features are described in the following Sections 3.3, 3.4, and 3.5, respectively.

3.3 The Anaphase Threading Model

The main novelty of Anaphase, compared to previous SpMT systems, resides on its threading model. The Anaphase threading model has the following unique features that make it more powerful than conventional models:

- The proposed threading model is able to decompose a sequential code into multiple speculative threads at a very fine granularity (individual instruction level), in contrast to traditional threading techniques in which big chunks of consecutive instructions are assigned to threads. This flexibility is crucial to exploit TLP on irregular sequential applications where traditional partitioning schemes end up with many inter-thread data dependences that may limit

performance. Other benefits of this fine-grain thread decomposition are that it improves the workload balance of the threads, and it increases the amount of memory level parallelism (MLP) that can be exploited.

- The Anaphase SpMT model allows three different approaches to manage inter-thread dependences: (i) use explicit inter-thread communications, (ii) use pre-computation slices (replicated instructions) to locally satisfy these dependences or (iii) ignore them, speculating no dependence and allow the hardware to detect the potential violation. The decomposition algorithm (see Section 3.4.2) includes heuristics to select at compile time the best way to deal with each inter-thread dependence.
- In Anaphase, each speculative thread is self-contained from the point of view of the control flow and includes all the branches it needs to resolve its own execution. Therefore, the Anaphase multi-core processor does not need any special hardware in the core front-end to handle the control flow of the threads or to manage branch mispredictions. In the Anaphase architecture, each core fetches, executes and commits instructions independently (except for the synchronization points incurred by explicit inter-thread communications; see Section 3.5).

These features will be described in more detail in the following subsections.

3.3.1 Fine-grain Thread Decomposition

As previously pointed out, the main feature of the Anaphase speculative multithreading paradigm is what we refer as fine-grain thread decomposition. The idea is to generate threads from a sequential code flexibly distributing individual instructions among them. In the Anaphase architecture proposed in this thesis this process is implemented statically at compile time. However, the same concept could be implemented dynamically by hardware or by a dynamic optimizer.

Figure 29 shows the concept of fine-grain thread decomposition with a simple example of a small loop formed of four basic blocks (*A*, *B*, *C*, *D*). Each block consists of several instructions, labeled as *Ai*, *Bi*, *Ci*, and *Di*. Figure 29 (a) shows the original control-flow graph (CFG) of the loop and a piece of the dynamic instruction stream when it is executed. Figure 29(b) shows the result of one possible fine-grain thread decomposition of the loop into two threads. The CFG of each resulting thread and its dynamic instruction stream when they are executed in parallel is shown in the figure. Notice that this thread decomposition is more flexible than traditional schemes where big chunks of instructions are assigned to threads. Typically, a traditional threading scheme on this example would assign loop iterations to each

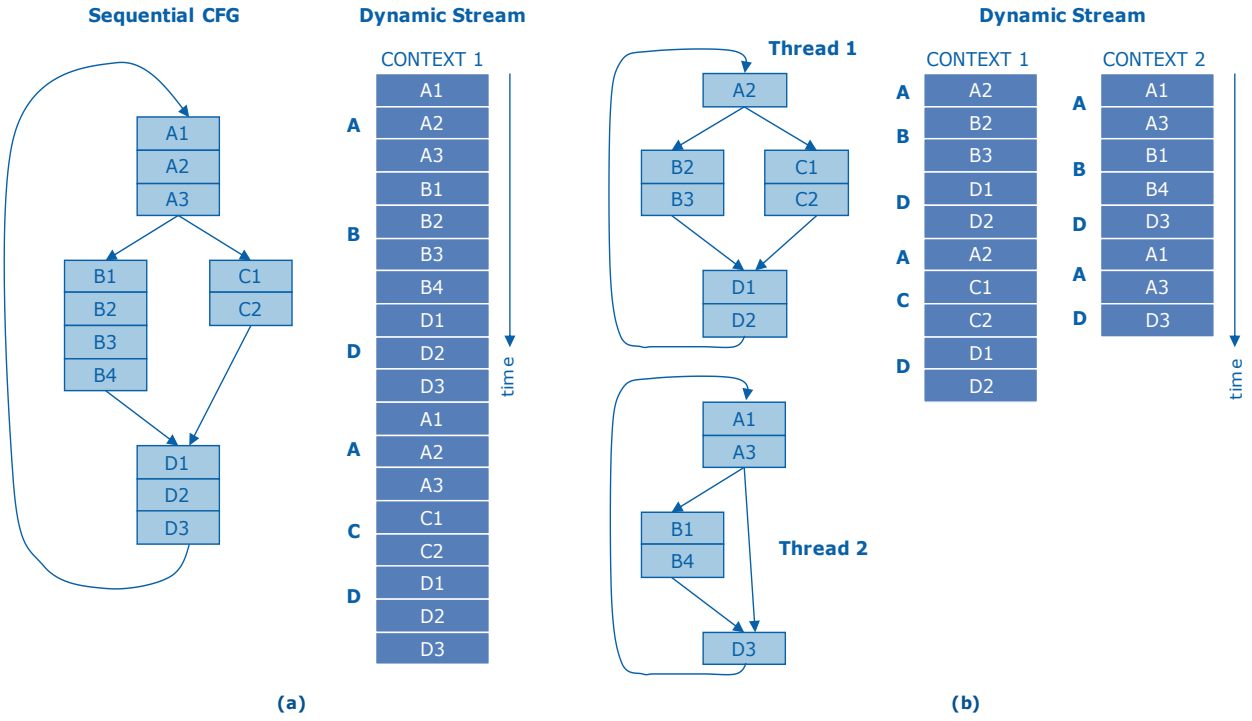


Figure 29. Example of Anaphase fine-grain thread decomposition of a loop

thread. In that sense, the Anaphase threading model is a superset of traditional threading schemes. In addition, even though the example shown in the figure is a loop, the concept of fine-grain thread decomposition is orthogonal to any high-level code structure and can be applied to any piece of sequential code.

The flexibility to distribute individual instructions among threads can be leveraged to implement different policies for generating them. Some of the policies that contribute to a better thread decomposition of a sequential code because they allow exploiting more ILP, TLP, and MLP are the following ones:

- Instructions may be assigned to threads in order to minimize the amount of inter-thread data dependences. This is one of the main advantages of Anaphase in respect of previous threading models. This policy is crucial to speculatively parallelize irregular sequential codes where traditional threading models fail to exploit TLP.
- Instructions may be assigned to threads in order to balance their workload. Notice that the fine-grain thread decomposition allows a fine tune of the workload balance because decisions to balance the threads can be done at instruction level.

- Instructions may be assigned to threads in order to better exploit memory level parallelism (MLP). MLP is an important source of parallelism for memory bounded applications. For these applications, an increase on MLP may imply a significant increase in performance. The fine-grain thread decomposition technique allows distributing load instructions among threads in order to increase MLP.

All these desirable policies are exploited by the Anaphase thread decomposition algorithm described in Section 3.4.2.

3.3.2 Inter-thread Data Dependences Management

As stated several times on this thesis, one of the key issues of any speculative multithreading paradigm is how inter-thread data dependences are handled. This management has more or less impact on performance depending on the type of threads that are generated. Whereas some simplified threading schemes assume no dependences among threads and do not provide any mechanism to solve them, in Anaphase speculative threads may have many dependences among them that need to be gracefully handled in order to not harm performance.

A part of the possibility of speculating (ignoring) some dependences, two additional mechanisms to solve the inter-thread data dependences are considered in Anaphase:

- As in Mitosis (see Section 2.3 in previous chapter), we propose the use of pre-computation slices (*p-slices*) to break inter-thread data dependences and to satisfy them locally. Given an instruction *I* assigned to a thread *T1* that needs a datum generated by a thread *T2*, all required instructions belonging to its *p-slice* (the subset of instructions needed to generate the datum needed by *I*) that have not been assigned to *T1*, are replicated into *T1*. In Anaphase, these instructions are referred to as replicated instructions. The main distinguishing feature with previous threading schemes that use pre-computation, like Mitosis, is that replicated instructions are treated as regular instructions and may be scheduled with the rest of instructions assigned to a thread. As a result, in an Anaphase thread replicated instructions are mixed with the rest of instructions and may be reordered to minimize the execution time of the thread. Moreover, note that pre-computing a value does not imply replicating all instructions belonging to its *p-slice* because some of the intermediate data required to calculate the value could be computed in a different thread and communicated as explained in the following point.

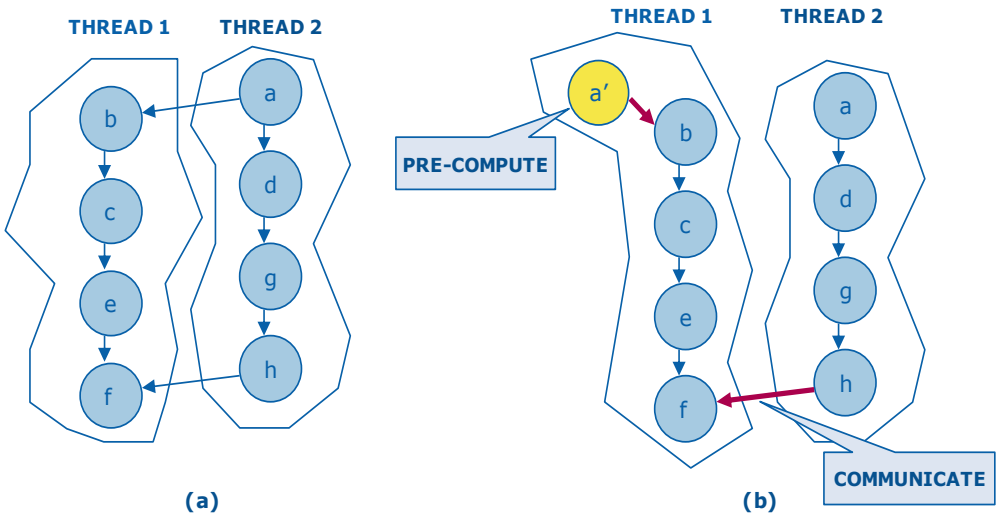


Figure 30. Example of inter-thread data dependences management in Anaphase threads

- In the Anaphase threading paradigm, those dependences that either (i) may require too many replicated instructions to satisfy them locally or (ii) can be delayed a certain amount of cycles without harming execution time, are solved through an explicit inter-thread communication. This feature reduces the amount of instructions that have to be replicated but introduces a synchronization point for each explicit communication (at least in the receiver instruction). The software / hardware mechanism to support this feature is described in Section 3.5.

Figure 30 shows a simplified example of two threads with two data dependences among them. In the figure, bullets are instructions and arrows represent data dependences between two instructions. As can be seen, one dependence is solved through a *p-slice*, which requires one replicated instruction (labeled as "*a'*"), and the other through an explicit communication.

In the Anaphase architecture, inter-thread dependences are detected at compile time analyzing the code and using profile information. Heuristics are used in order to select for each inter-thread dependence the best mechanism to deal with it (see Section 3.4.2.2.2). Notice that not all actual dependences can be identified by the profiler (i.e. memory dependences) and unresolved dependences may show up at run-time. In that sense, resulting threads are speculative and may sometimes execute wrong. As in the case of Mitosis, the hardware must provide mechanisms for detecting such dependence violations, squash the offending thread/s, and restart its/their execution.

3.3.3 Control Flow Management

In the proposed fine-grain SpMT model, distributing instructions to threads at instruction granularity to execute them in parallel poses a new challenge on how to reproduce the control flow of the original sequential execution. In the Anaphase architecture, this is managed by software when the speculative threads are generated. Therefore, the processor front-end does not require any additional hardware in order to handle the control flow of the fine-grain SpMT threads or to manage branch mispredictions. With the proposed scheme, control speculation for a given thread is managed locally in the context it executes (i.e. core) by using the conventional prediction and recovery mechanism on place.

The idea is to force that every thread includes all the branches it needs to compute the control path for its instructions. Those branches that are required to execute any instruction of a given thread and are not included in that thread are then replicated. Note that not all the branches are needed in all the threads but only those that affect the execution of its instructions. Moreover, having a branch instruction in a thread does not mean that all the instructions needed to compute its outcome need to be included in the thread because the proposed threading paradigm allows inter-thread communications. For instance, a possible scenario is that only one thread computes the branch condition and communicates it (as any other live-in) to the rest of the threads. Another scenario is that the computation of the control flow of a given branch is completely spread out among all the threads.

Replication of the required branches and its computation is managed by the thread decomposition algorithm (see Section 3.4.2.2.2). Figure 31 shows three different examples of possible outcomes of the thread decomposition algorithm when considering the control flow management. The instructions involved in the control flow are highlighted and the arrows show explicit inter-thread communications. As it can be seen, the branch has been replicated in all threads that need it (*T1* and *T2*) in all three cases. In the case of (a), the instructions that compute the branch are executed by *T2* and the outcome sent to *T1* whereas in (b) the computation is replicated in both threads (*T1* and *T2*), so there is no need for an explicit communication. Note that the computation of the branch is partitioned as any other computation in the program so it may be split among different threads that communicate explicitly (including threads that do not really care about the branch). An example of this is shown in (c). As we will see in the following section, for each branch that needs to be replicated, the thread decomposition algorithm is in charge of selecting the best way to replicate, communicate, or split the computation among the threads.

Original Code	T0	T1	T2
SI = SI + DI CX = CX * 2 **AX = AX + 1** CX = CX + AX **BX[AllFlags] = AX + 1** **Bz LABEL** CX = AX + BX AX = 0 LABEL: DX = SI*2 AX = AX*2	SI = SI + DI DX = SI * 2	CX = CX * 2 CX = CX + AX **Bz LABEL_T1** CX = AX + BX LABEL_T1:	**AX = AX + 1** **BX[AllFlags] = AX + 1** **Bz LABEL_T2** AX = 0 LABEL_T2: AX = AX * 2

(a) single control flow computation

Original Code	T0	T1	T2
SI = SI + DI CX = CX * 2 **AX = AX + 1** CX = CX + AX **BX[AllFlags] = AX + 1** **Bz LABEL** CX = AX + BX AX = 0 LABEL: DX = SI*2 AX = AX*2	SI = SI + DI DX = SI * 2	AX = AX + 1 CX = CX * 2 CX = CX + AX **BX[AllFlags] = AX + 1** **Bz LABEL_T1** CX = AX + BX LABEL_T1:	**AX = AX + 1** **BX[AllFlags] = AX + 1** **Bz LABEL_T2** AX = 0 LABEL_T2: AX = AX * 2

(b) full replication of the control flow

Original Code	T0	T1	T2
SI = SI + DI CX = CX * 2 **AX = AX + 1** CX = CX + AX **BX[AllFlags] = AX + 1** **Bz LABEL** CX = AX + BX AX = 0 LABEL: DX = SI*2 AX = AX*2	SI = SI + DI DX = SI * 2 **AX = AX + 1**	CX = CX * 2 CX = CX + AX **BX[AllFlags] = AX + 1** **Bz LABEL_T1** CX = AX + BX LABEL_T1:	**Bz LABEL_T2** AX = 0 LABEL_T2: AX = AX * 2

(c) split computation of the control flow

Figure 31. Example of three different thread decompositions and control flow management

3.3.4 Related Work

Traditional speculative multithreading schemes, including the Mitosis architecture [30][54] described in Chapter 2, decompose sequential codes into large chunks of consecutive instructions. Three main different scopes for decomposition have been so far considered: loops [16][50][98], function calls [5] and quasi-independent points [37][58]. When partitioning loops, most of the previous schemes spread the iterations into different threads, i.e. one or more loop iterations are included in the same thread. A limited number of works consider the decomposition of the code inside the iterations by assigning several basic blocks into the same thread, while others assign strongly connected components to different threads. On

the other hand, all previous works considering the decomposition at function calls shred the code in such a way that consecutive functions are dynamically executed in parallel. Finally, all works considering general control flows with the objective of decomposing the program in quasi-independent points considers the shredding of the whole application in such a way that consecutive chunks of basic blocks are assigned to different threads for its parallel execution.

The coarse grain decomposition featured by all previous speculative multithreading approaches may constraint the benefits of this paradigm. This is particularly true when these techniques must face hard to parallelize codes. When these codes are decomposed in a coarse-grain fashion, it may be the case that too many dependences appear among threads. This may end up limiting the exploitable TLP for this codes and harming performance. In order to deal with this limitation, Anaphase parallelizes applications at instruction granularity. Therefore, the new model proposed experiences a larger flexibility because it will choose the granularity that best fits a particular loop. Thus it has more potential for exploiting further TLP than previous schemes and it can be seen as a superset of all previous threading paradigms.

On the other hand, four main mechanisms have been considered so far to manage the data dependences among speculative threads: (a) speculation; (b) communication [73][91][96][103]; (c) pre-computation slices [30][106], and (d) value prediction[3][62]. Each of these mechanisms has its benefits and drawbacks, and in each particular situation (i.e. dependence in a piece of code) one mechanism may be more appropriate than another. In order to overcome this limitation, Anaphase considers the possibility of using three of these mechanisms and selecting the most appropriate solution for each dependence.

3.4 The Anaphase Compiler

As previously mentioned in this chapter, Anaphase is a hybrid software / hardware architecture. In our proposal, the software part is implemented on top of a conventional compiler (see Section 3.6.1 for more details). The main job of the Anaphase compiler is to parallelize sequential applications by decomposing them into speculative threads.

Figure 32 shows a high-level view of the main steps performed by the Anaphase compiler to parallelize a sequential application. Specifically, the compiler is responsible for: (1) profiling the application, (2) analyzing the code and detecting the most convenient regions of code for parallelization, (3) decomposing the selected regions into speculative threads, and finally, (4) generating optimized code and rollback code for every region.

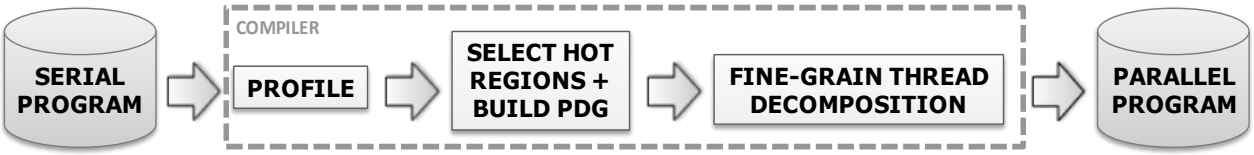


Figure 32. High-level scheme of the Anaphase compiler

Having dynamic information is crucial for selecting good regions to parallelize and performing an effective thread partitioning. In the Anaphase compiler, a first step of profiling the application is performed in order to obtain information about instruction execution counts and data dependences. Then, regions of code that are frequently executed (a.k.a. hot regions) are selected to be parallelized and the Program Dependence Graph (PDG) [26] is built for each of them. Once hot regions are identified and selected, the PDG of each region is passed to the Anaphase decomposition algorithm that partitions it into fine-grain speculative threads. This algorithm takes benefit of the flexibility of the threading model presented in Section 3.2 and generates speculative threads that fully exploit the capabilities of the aforementioned model. The decomposition algorithm uses smart heuristics and a performance model to guide the generation of speculative threads.

Finally, once a partition is computed, the compiler generates the appropriate code to represent it. This implies mapping the optimized version of the region somewhere in the address space of the application, placing the corresponding *spawn* instructions and adding rollback code.

In the following sections we will describe all these steps in detail.

3.4.1 Profiling and Region Selection

The first step of the Anaphase compiler is to profile the sequential application in order to obtain information about instruction execution counts and data dependences. This information is initially used to detect regions of code that are frequently executed and to select those regions that are going to be speculatively parallelized. The objective of the region selection step is to obtain regions that are amenable to be decomposed into speculative threads and that cover most of the execution time of the application.

The Anaphase thread decomposition algorithm is able to parallelize any code region: loops, routines, straight-line code, or any other code structure. In fact, the algorithm receives as input a Program Dependence Graph (PDG) and decomposes it into threads without requiring any knowledge about the high-level code structures present in the graph. Although the proposed fine-grained speculative multithreading paradigm can be applied to any code structure, in this thesis we have concentrated on

applying the partitioning algorithm to loops. In particular, we limit our focus to outer loops which are frequently executed according to the profiling information. The motivation behind this is that it provides a significant high coverage while keeping the region selection step simple. However, we envision more complex schemes for region selection, like the one proposed for Mitosis (see Section 2.4.1), as part of the future work.

The compiler performs some basic transformations to the selected loops to improve the threading by exposing more thread level parallelism (TLP). Specifically, selected loops are unrolled and frequently executed routines are inlined in order to enlarge the scope of the thread decomposition technique on such loops.

Finally, for each selected loop, the Data Dependence Graph (DDG) and the Control Flow Graph (CFG) are built. Such graphs are complemented by adding the profiling information. Nodes (instructions) and edges in the DDG are weighted by their dynamic occurrences. On the other hand, control edges in the CFG are weighted by the frequency of the taken path. Both graphs are then collapsed into the PDG.

3.4.2 Thread Decomposition Algorithm

PDGs of selected loops are passed to the Anaphase thread decomposition algorithm. The algorithm uses a multi-level graph partitioning approach to shred the sequential code into fine-grain speculative threads. Multilevel strategies have been shown to be very effective mechanisms to deal with graph partitioning [40]. As can be seen in Figure 33, the multi-level graph partitioning strategy consists of two main steps: coarsening and refinement.

Like any multi-level graph partitioning algorithm, the first part of the algorithm is the coarsening step. In such a step, nodes (instructions, in our case) are iteratively collapsed into bigger nodes, also known as *supernodes*, until there are as many nodes as desired number of partitions (or threads, in our case). During this process, different levels of *supernodes* are created. A node from a given level contains one or more nodes from the level below it. Also, each level has fewer nodes in such a way that the bottom level contains the original graph (the one passed to this step of the algorithm) and the topmost level only contains as many *supernodes* as partitions (or threads) we want to generate. The goal of this step is to find relatively good partitions using simple but effective parallelization heuristics.

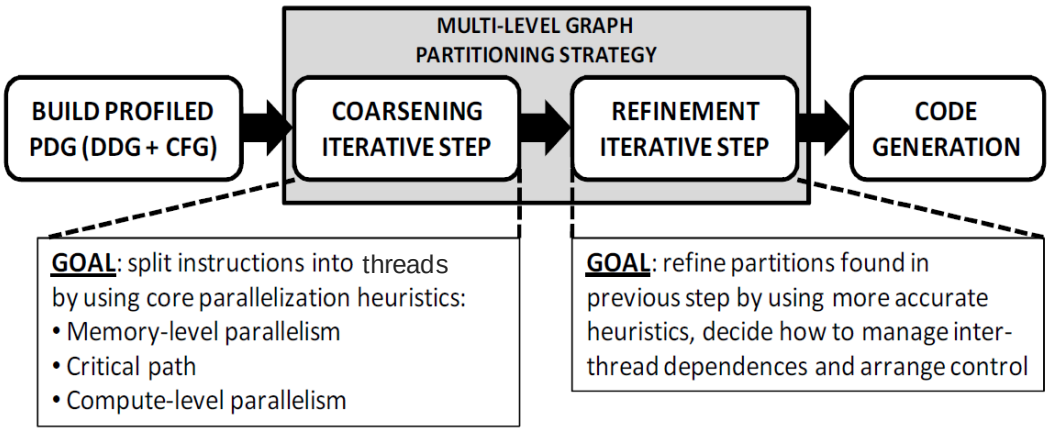


Figure 33. Anaphase partitioning flow for a given loop

After the coarsening step, a refinement process begins. In this step, the partitions found during the coarsening phase are iteratively revaluated by moving *supernodes* to other threads and estimating their benefits using more fine-grained heuristics. Furthermore, in our case it is during this phase that inter-thread dependences and control replication are managed. This step finishes when no more benefits are obtained by moving nodes (instructions) from one partition to the other based on the heuristics.

Figure 34 shows an overview of how a multi-level graph partitioning algorithm works. In this simplified example the initial nodes (instructions) are partitioned into two sets (threads) after iterating four levels (L0…L3). Although the forthcoming sections give more insights on how this kind of algorithms works, we refer the reader to [40] for further details.

The following sections describe each step in deeper detail, focusing on the exploited heuristics, which are the key components of Anaphase.

3.4.2.1 Coarsening Step

As previously mentioned, the coarsening step is an iterative process in which nodes (instructions) are collapsed into bigger nodes until there are as many nodes as the number of desired partitions. At each iteration a new level is created and nodes are collapsed in pairs (see Figure 34).

The goal of this pass is to generate relatively good intermediate partitions. Since this pass is backed up by a refinement step, the partitions must contain nodes that it makes sense to assign to the same thread but without paying much attention to the partition details, such as how to manage inter-thread dependences and how to replicate the control. It is for this reason that the coarsening step uses simple but

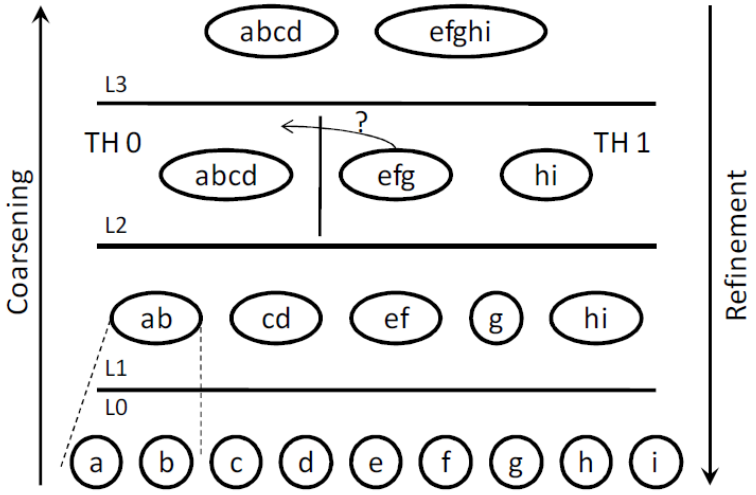


Figure 34. Multi-level graph partitioning example with four levels and two final sets

effective core heuristics to assign "dependent" code sequences to the same partition and "independent" code sequences to different threads.

These core heuristics are based on three concepts:

- *Memory-level parallelism*. Parallelizing miss accesses to the most expensive levels of the memory hierarchy (last levels of cache or main memory) is an effective technique to hide their big latencies. Hence, it is often good to separate the code into memory and computation components for memory-bound sequences of code, and assign each component to a different thread. The memory component includes the miss accesses and their pre-computation slice, while the compute component includes the rest of the code: code that does not depend on the misses and code that depends on the misses but does not generate more misses. The rationale behind this is that the memory component often has a higher density of misses per instruction and hence it normally does a better usage of the processor capabilities to parallelize misses (e.g. miss status holding registers) when assigned to the same core. For example, a core will not parallelize misses if they are at a distance (in dynamic number of instructions) greater than the ROB size. However, the distance may be reduced if some of the instructions in between are assigned to another core. This may allow servicing some misses in parallel.

- *Critical-path*. It is often good to assign instructions on the critical path to the same thread and off-loading non-critical instructions to other threads, in order not to incur in any delay in the execution of the former.

- *Compute-level parallelism*. It is often good to assign independent pieces of code to different threads and dependent pieces of code to the same thread in order to exploit the additional computation resources and to avoid unnecessary synchronization points between the two threads.

The coarsening step follows the algorithm shown in pseudo-code in Figure 35. In order to exploit the aforementioned criteria, a matrix *M* is built to describe the relationship between node pairs (see routine *create_and_fill_matrix_at_current_level* in Figure 35). In particular, the matrix position *M[i,j]* describes how good is to assign nodes *i* and *j* to the same partition (and *M[i,j] = M[j,i]*). Each matrix element is a value that ranges between 0 (worst ratio) and 2 (best ratio): the higher the ratio, the more related two nodes are (see Section 3.4.2.1.3 for the rationale behind these values). The matrix is initialized to all zeros, and cells are filled based on three heuristics that follow the aforementioned criteria.

Once matrix *M* has been filled with the three heuristics, the coarsening step of Anaphase uses it to collapse pairs of nodes into bigger nodes (see routine *collapse_nodes* in Figure 35). In particular, the algorithm collapses node pairs in any order (we have observed that the ordering in this case is not important) as long as the ratio *M[i,j]* of the pair to be collapsed is at most 5% worst than the best collapsing option for node *i* and than the best collapsing option for node *j*[4]. This is so because a multi-level graph partitioning algorithm requires a node to be collapsed just once per level [40]. Hence, as we proceed collapsing, there are fewer collapsing options at that level and we want to avoid collapsing two nodes if their ratio *M* is not good enough.

Finally, it is important to remark that matrix *M* is filled in the same manner when internal levels of the graph partitioning algorithm are considered. The main difference is that the size of the matrix is smaller at each level. In these cases, since a node may contain more than one node from level 0 (where the original nodes reside), all dependences at level 0 are projected to the rest of the levels. For example, node *ab* at level 1 in Figure 34 will be connected to node *cd* by all dependences at level 0 between nodes *a* and *b* and nodes *b* and *c*. Hence, matrix *M* is filled naturally at all levels based on the three heuristics.

When the multi-level graph is done collapsing, the algorithm proceeds to the next step, the refinement phase (see Section 3.4.2.2).

[4] We experimentally found that 5% is a reasonable threshold but other thresholds may be more adequate for different scenarios.

```
Routine coarsening step()
current_level = 0
while num_partitions > N do
  call create_and_fill_matrix_at_current_level()
  current_level++
  call collapse_nodes()
done


Routine collapse_nodes()
for each node pair (i,j) in any order
  collapse them if all the three conditions are met:
    (i) neither node i nor node j have been
        collapsed from previous level to current_level
    (ii) M[i][j] ≥ 0.95*M[i][k] for all nodes k
    (iii) M[i][j] ≥ 0.95*M[k][j] for all nodes k
endfor


Routine create and fill matrix at current level()
initialize matrix M to all zeroes
for each node i identified as a delinquent load
  for each consumer node j of i based on data deps
    M[i][j] = M[j][i] = 0.1
  endfor
endfor

compute slack of each edge
for all edges with a slack of 0 connecting nodes i & j
  if M[i][j] = 0
    M[i][j] = M[j][i] = 2.0
  endif
endfor

compute common pred. ratio F for all node pairs (i,j)
for each node pair (i,j)
  if M[i][j] = 0
    M[i][j] = M[j][i] = F(i,j)
  endif
endfor
```

Figure 35. Pseudo-code of the Anapahase coarsening step

The key elements of the Anaphase coarsening step, the three heuristics, are described in the following subsections in the order they are applied to fill matrix *M*.

3.4.2.1.1 Delinquent loads

As mentioned before, exploiting memory-level parallelism is very important to achieve good performance in memory-bound code sequences. In order to do so, the algorithm detects delinquent loads [19], which

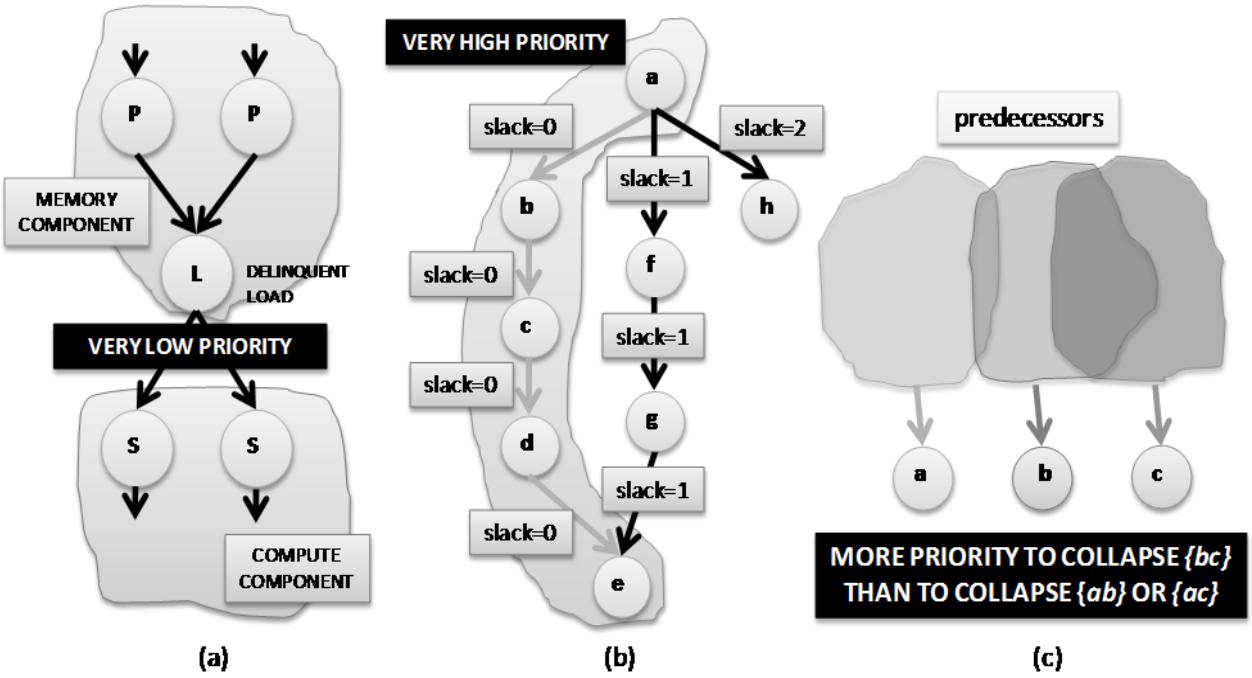


Figure 36. Coarsening heuristics: (a) Delinquent loads, (b) Slack, and (c) Common Predecessors

are those load instructions that will likely miss in cache often and therefore impact performance. After studding different thresholds, we have observed that a good trade-off is achieved by marking as delinquent all loads that have a miss rate higher than 10% in *L2* cache based on profiling information.

By using this heuristic we want to favor the formation of nodes that contain delinquent loads and their pre-computation slices, in order to allow the refinement stage to consider these loads separated from their consumers. In order to achieve this effect, the data edge that connects a delinquent load with a consumer instruction is given very low priority (see Figure 36(a) for a graphical representation). Therefore, the ratio for these two nodes is fixed to 0.1 in matrix *M* (a very low priority). The rest of the cells of *M* are filled with the following two heuristics.

3.4.2.1.2 Slack

As discussed above, grouping together critical instructions to the same thread is an obvious way to avoid delaying their execution unnecessarily. Unfortunately, computing at compile time how critical an instruction will be is impossible. However, since Anaphase has a posterior pass that refines the decisions performed at this step, a simple estimation works fine at this point.

For the purpose of estimating how critical an instruction is we compute its local slack [27], defined as its freedom to delay its execution without impacting total execution time. Slacks are assigned to edges.

In order to compute such slack, first, the algorithm computes the earliest dispatch time for each instruction considering only data dependences in the PDG and ignoring cross-iteration dependences. After this, the latest dispatch time of each instruction is computed in the same manner. The slack of each edge is defined as the difference between the earliest and the latest dispatch times of the consumer and the producer nodes respectively.

Two nodes *i* and *j* that are connected by an edge with very low slack are considered part of the critical path and will be collapsed with higher priority. We have considered critical edges those with a slack of 0 (Figure 36(b) shows an example). Therefore, the ratios *M[i,j]* and *M[j,i]* for these nodes are set to 2.0 (the best ratio). The rest of the matrix cells are filled with the following last heuristic.

3.4.2.1.3 Common Predecessors

Finally, in order to assign dependent instructions to the same thread and independent instructions to different threads, we compute how many predecessor instructions each node pair *(i,j)* share by traversing edges backwards. In particular, we compute the predecessor relationship of every pair of nodes as the ratio between the intersection of their predecessors and the union of their predecessors. The following equation computes the ratio *R* between nodes *i* and *j*:

$$R(i, j) = \frac{P(i) \cap P(j)}{P(i) \cup P(j)}$$

Where *P(i)* denotes the set of predecessors of *i*, including itself. It is also worthwhile to mention that each predecessor instruction in *P(i)* is weighted by its profiled execution frequency in order to give more importance to the instructions that have a deeper impact on the dynamic instruction stream.

The ratio *R(i,j)* describes to some extent how related two nodes are. If two nodes share an important amount of nodes when traversing the graph backwards, it means that they share a lot of the computation and hence it makes sense to map them together. They should have a big relationship ratio in matrix *M*. On the other hand, if two nodes do not have common predecessors, they are independent and are good candidates to be mapped into different threads. Note that *R(i,j)* = *R(j,i)*. Figure 36(c) graphically represents this concept.

In the presence of recurrences, we found many nodes having a ratio *R(i,j)* of 1.0 (they share all predecessors). To solve this we compute the ratio twice, one as usual, and a second time ignoring cross-iteration dependences. The final ratio *F(i,j)* is the sum of these two. We have observed that computing this ratio twice as explained here improves the quality of the obtained threading and increases performance

consequently. This final ratio is the value that we use to fill the rest of cells in *M[i,j]*. Note that this is the reason why the matrix values range between 0 and 2 and not between 0 and 1.

3.4.2.1.4 Other Heuristics

It is important to remark that although our final proposal for the coarsening step of the algorithm uses the aforementioned three heuristics, we previously tried several other techniques. For instance, we observed that applying each of them individually offers less speed-up for most of the benchmarks when the threading is complete. We also tried to collapse nodes based on the weight of the edges connecting them, or to collapse them based on workload balance. However, the overall performance was far from that obtained with the above heuristics. We even tried a random coarsening step and, although some performance was achieved by the refinement step, the overall result was poor. Hence, it is the conjunction of the three heuristics presented here that achieved the best performance, which is later reported in Section 3.5.2.

3.4.2.2 Refinement Step

The refinement step is also an iterative process that walks all the levels created during the coarsening step from the topmost levels to the bottom-most levels and, at each level, it tries to find a better partition by moving one node to another partition. An example of a movement can be seen in Figure 34: at level 2, the algorithm decides if node *efg* should be at thread 0 or thread 1.

The purpose of the refinement step is to find better partitions by refining the already "good" partitions found during the coarsening process. Due to the fact that the search at this point of the algorithm is not blind, finer heuristics can be employed. Furthermore, it is at this moment that Anaphase decides how to manage inter-thread dependences and replicate the control as required.

We have used the classical Kernighan-Lin (*K-L*) algorithm [41] to build our refinement step. Thus, at a given level, we compute the benefits of moving each node *n* to another thread by using an objective function and choose the movement that maximizes such objective function. Note that we try to move all nodes even if the new solution is worse than the previous one based on the objective function. This allows the *K-L* algorithm to overcome local optimal solutions. However, after trying all possible movements at the current level, if the solution is not better than the current one, we discard such a solution and jump down to the next level.

The following subsections describe in more detail how we apply some filtering in order to reduce the cost of the algorithm, how inter-thread dependences are managed and how the objective function works.

3.4.2.2.1 Movement Filtering

Trying to move all nodes at a given level is very costly, especially when there are many nodes in the PDG. This is alleviated in Anaphase by focusing the movements to the subset of the nodes that seem more promising. In particular we focus on those nodes that if moved may have higher impact in terms of (i) improving workload balance among threads and (ii) reducing inter-thread dependences.

For improving workload balance, we focus on the top *K* nodes that may help to get close to a perfect workload balance between the threads. Workload balance is computed by dividing the biggest estimated number of dynamic instructions assigned to a given thread by the total number of estimated dynamic instructions. A perfect balance when *N* threads are considered is *1/N*. On the other hand, Anaphase picks the top *L* nodes that may reduce the number of inter-thread dependences. The reduction on the number of inter-thread dependences is simply estimated by using the occurrence profiling of the edges that are in the cuts of the partition [40].

After some experiments trying different *K* and *L* values, we have seen that a good trade-off between improving the quality of the generated threads (that is, improving performance) and the compiling cost to generate them is achieved by $K = L = 10$. Hence, we reduce the amount of movements to 20.

3.4.2.2.2 Inter-Thread Dependences and Control Replication

As mentioned before, the refinement step tries to move a node from one thread to another thread and computes the benefits of such a movement based on an objective function. Before evaluating the partition derived by one movement, the algorithm decides how to manage inter-thread dependences and arranges the control flow in order to guarantee that both threads are self-contained. These concepts are explained in more detail in Section 3.3.2 and Section 3.3.3.

As stated in the Anaphase threading model section (see Section 3.3.2), given an inter-thread dependence, Anaphase may decide to:

- Fulfill it by using explicit inter-thread communications, which in our current approach are implemented through a software / hardware mechanism (see Section 3.5).
- Fulfill it by using pre-computation slices to locally satisfy these dependences. The producer instructions and all their dependence chains (or just a subset) are thus replicated into the other thread in order to avoid the communication.
- Ignore it, speculating no dependence if it barely occurs.

The current inter-thread dependence management heuristic implemented in Anaphase does not ignore any dependence (apart, of course, from those not observed during profiling). Hence, the main decision is whether to communicate or pre-compute an inter-thread dependence.

Communicating a dependence is relatively expensive, since the communicated value goes through the shared *L2* cache when the producer reaches the head of the ROB of its corresponding core. On the other hand, an excess of replicated instructions may end up delaying the execution of the speculative threads and hence impacting performance as well. Therefore, the selection of the most suitable alternative for each inter-thread dependence may have a significant impact on the performance achieved by Anaphase.

The core idea used in Anaphase is that the larger the replication needed to satisfy a dependence locally, the lower the performance. By leveraging this idea, when the profiled-weighted amount of instructions in a pre-computation slice exceeds a particular threshold the dependence is satisfied by an explicit communication. Otherwise, the dependence is satisfied by means of the pre-computation slice.

We have experimentally observed that by appropriately selecting the threshold that defines the amount of supported replication (see Section 3.5.2 for the actual values used in the experiments), this naïve heuristic achieves better performance than other schemes that we have tried that give more importance to the criticality of the instructions (recall that criticality at compile time is in fact an estimated approximation).

3.4.2.2.3 Objective Function

At the refinement stage, each partition has to be evaluated and compared with other partitions. The objective function estimates execution time for this partition when running on a tile of the multi-core processor (see Section 3.5 for a description of the Anaphase multi-core processor).

In order to estimate the execution time of a partition, a 20K dynamic instruction stream of the region obtained by profiling is used. Using this sequence of instructions, the execution time is estimated as the longest thread based on a simple performance model that takes into account data dependences, communications among threads, issue width resources and the size of the ROB of the target core.

3.4.2.2.4 Final Remarks

We have observed that the refinement step of the algorithm is also crucial to achieve good performance. In fact, if we used the final partition obtained by the coarsening phase, the overall performance was very poor. This does not mean that the coarsening step is useless, since other coarsening heuristics explained in Section 3.4.2.1.4 did not work out even with the same refinement pass. Recall that the main goal of the

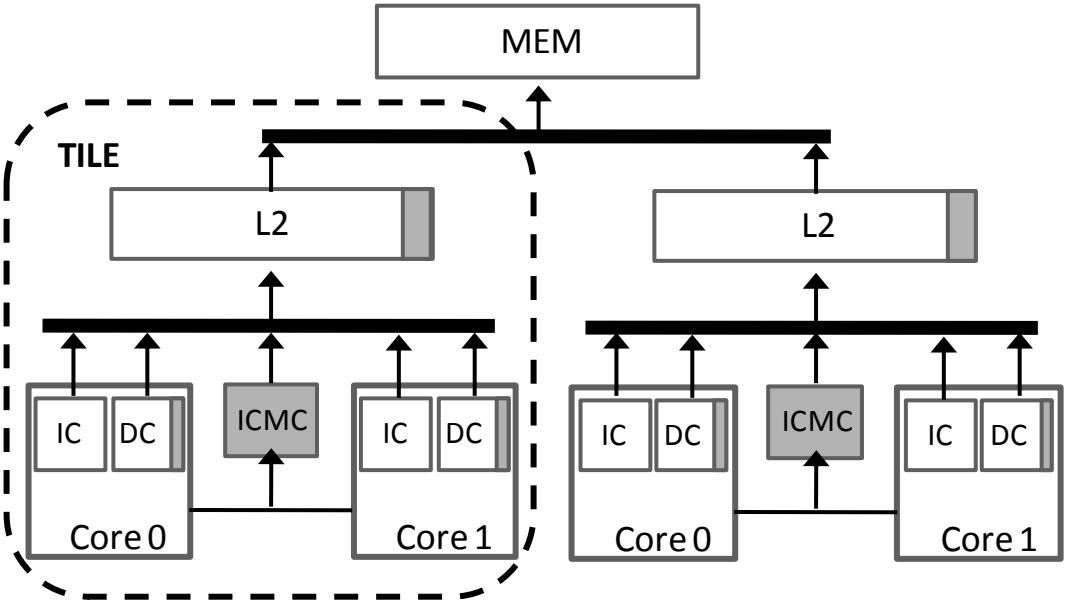


Figure 37. Anaphase multi-core processor with 2-core tiles

coarsening step is not to provide a final partition, but to provide meaningful nodes in all levels for the refinement to play with.

We also tried other heuristics in order to decide when a partition is better than another during the refinement phase. These heuristics included maximizing workload balance, maximizing miss density in order to exploit memory-level parallelism, minimizing inter-thread dependences, among others. Overall, such heuristics tended to favor only a subset of the programs and never reached the performance of the heuristic described here.

3.5 The Anaphase Multi-Core Processor

As has been previously mentioned, the Anaphase architecture presented in this chapter is composed of two big components: (i) the Anaphase compiler described in Section 3.4, which is responsible of shredding an application into speculative threads at fine-grain granularity, and (ii) the Anaphase processor, which includes all the microarchitecture support for running Anaphase speculative threads. This section describes in detail the Anaphase processor.

Figure 37 shows a scheme of the Anaphase processor. It is a conventional multi-core architecture with OoO cores [67] organized in tiles. In this specific implementation each tile includes two cores. Each core has a first level write-through data caches (*DC*) and an instruction cache (*IC*). The first level data cache includes a state-of-the-art stream hardware prefetcher. These caches are connected to a shared

copy-back *L2* cache through a split transactional bus. Finally, the *L2* cache is connected through another interconnection network to the upper levels of the memory hierarchy (*MEM*) and to the rest of the tiles.

As seen in Section 3.2, tiles have two different operation modes: *independent* mode and *cooperative* mode. The cores in a tile execute conventional threads when the tile is in *independent* mode and they execute speculative threads (one in each core) from the same decomposed application when the tile is in *cooperative* mode.

The additional hardware required for running Anaphase threads is shown in dark grey in Figure 37. In a nutshell, the Anaphase hardware support is composed of a few additional bits in the private *DCs* and *L2* cache for managing the memory speculative state and a new component per tile called *Inter-Core Memory Coherence* module (*ICMC*) that orchestrates the execution of Anaphase threads. For complexity reasons, the Anaphase hardware support only allows the execution of Anaphase threads from the same application inside the same tile. This simplifies a lot the design, without limiting too much scalability, because the *ICMC* only has to handle activity from cores on a given tile.

The *ICMC* of every tile is in charge of controlling the activity of the cores inside the tile when they execute in *cooperative* mode. This piece of logic interferes very little with the cores. Hence, the cores fetch, execute and retire instructions from the speculative threads in a decoupled fashion most of the time. A subset of the instructions is sent to the *ICMC* after they retire in order to properly update the architecture state and perform the validation of the execution. The set of instructions considered by the *ICMC* is limited to memory and some control instructions. We refer to those instructions as *ordering instructions*.

The *ICMC* receives the *ordering instructions* and it handles them through the three structures shown in Figure 38. The main tasks performed by the *ICMC* are the following:

- The *ICMC* sorts *ordering instructions* in order to: (1) make changes made by the multi-threaded application visible to the other tiles as if it would have been executed sequentially; and (2) detect memory dependence violations among the threads running on the cores of the tile. When the *ICMC* detects a memory violation, it rollbacks to a previous consistent state and the software redirects execution towards the original sequential version of the code. For the purpose of reconstructing the memory sequential order the *ICMC* implements one FIFO queue (*memFIFO*) per core as shown in Figure 38. These queues keep *ordering instructions* when they retired from the associated core. The *ICMC* then globally retires instructions from these FIFO queues in the original program order specified by some marks associated with the

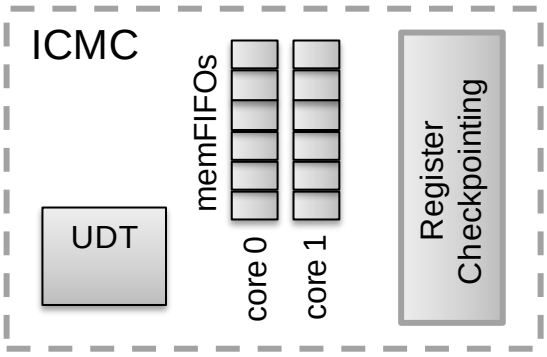


Figure 38. *ICMC* main structures for a 2-core tile

ordering instructions. Therefore, one of the duties of the *ICMC* is to reconstruct the original program order of the sequential application. The following Section 3.5.1 describes this software / hardware mechanism in detail.

- The *ICMC* and the extended memory hierarchy inside a tile allow each core running in *cooperative* mode to update its own memory state, while still committing the same state that the original sequential execution will produce. This is accomplished by: (1) allowing different versions of the same line in multiple *DCs*; and (2) avoiding speculative updates to propagate outside the tile. Section 3.5.2 describes in detail the memory state management.

- The proposed scheme requires some form of checkpointing to roll back the state to a correct state when the *ICMC* detects a misspeculation. Frequent checkpoints should be taken in order to keep the penalty due to a misspeculation small, without increasing too much the overhead to create them. The *ICMC* implements a novel scheme described in Section 3.5.3 that is able to take frequent checkpoints (every few hundreds of instructions) of the architectural state while allows a core running in a *cooperative* mode to retire instructions, reclaim execution resources and keep doing forward progress even when other cores are stalled.

It is important to remark that in *cooperative* mode the cores inside a tile need to synchronize very little among them. The only points where synchronization occurs between the cores are: (i) when an inter-thread dependence is satisfied by an explicit communication instruction pair, (ii) when a core fills up its FIFO queue in the *ICMC*, and (iii) when all speculative threads complete and they exit the optimized region. In the second case, one of the other cores has still to locally retire the oldest instruction/s in the system; hence, the core must wait until this happens and the *ICMC* frees up some of its FIFO queue space.

Finally, the architecture provides a mechanism to communicate values through memory between threads with explicit *send* / *receive* instruction pairs (a *send* is implemented through a special type of store, while a *receive* through a special type of load). Such a communication only blocks the receiver when the datum is not ready yet: the sender is never blocked. Although these communications happen through the *L2* cache, the decoupled nature of the cores requires that the sender is at the head of the reorder buffer (*ROB)* before sending the datum to memory. Hence, a communication is often a quite expensive operation.

In the following subsections we will describe in more detail the aforementioned tasks of the *ICMC* and the Anaphase hardware support.

3.5.1 Reconstructing Memory Sequential Order

In any SpMT system, each speculative thread updates its local view of the memory speculatively and independently. However, the architectural memory state (that is visible by the whole system) must be updated in program order to satisfy the consistency model and the detection of memory ordering violations among the threads. The order of memory instructions can be easily reconstructed in traditional speculative multithreading paradigms in which each thread executes a big chunk of consecutive instructions (for example a loop iteration or a function call) [5][30][58][98]. For instance, in the case of parallelizing loop iterations, instructions belonging to iteration *i* executed by thread 0 are older in sequential order than those belonging to iteration *i+1* and executed by thread 1.

However, in the case of Anaphase we are looking at a finer granularity speculative multithreading paradigm that is a superset of other traditional paradigms. Remind that in Anaphase, a sequential code can be decomposed into threads at instruction granularity and that some instructions may be assigned to more than just one thread (referred to as replicated instructions). As we have already pointed out, this new threading paradigm brings more flexibility when constructing the threads, but at the same time, poses new challenges in designing efficient mechanisms to handle control flow and the management of the architectural state. In this section, we focus on how to reconstruct the sequential order of instructions that have been arbitrarily assigned to speculative threads. More precisely, since we are interested on detecting inter-thread memory violations and updating the architectural memory state correctly, the mechanism only needs to target memory instructions.

The memory sequential order is reconstructed using special marks called Program Order Pointer (*POP*) bits that are included by the compiler in *ordering instructions*. Remind that, since we want to

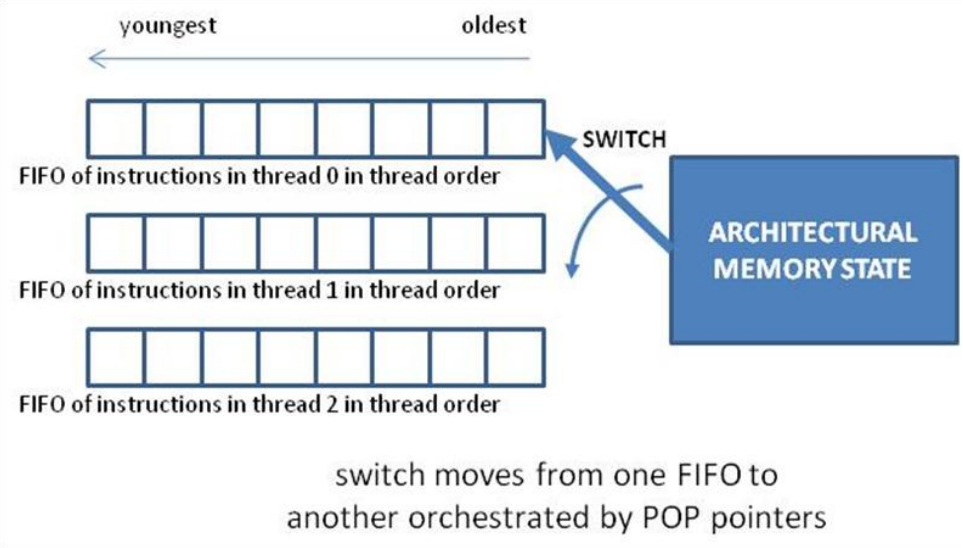


Figure 39. Mechanism to reconstruct *program order* base on *POP* marks

reconstruct the sequential memory order, *ordering instructions* are limited to memory and some control instructions. The *POP* bits of every *ordering instruction* indicate the speculative thread where the next memory instruction in the original sequential stream was assigned. *POP* marks are *N* bits wide (where $N = log_2 M$, *M* being the number of threads).

We define *thread order* as the order in which a thread sees its own assigned instructions, and *program order* the order in which all instructions looked like in the original sequential stream. *Thread order* can be easily reconstructed because each thread fetches and retires its own instructions in order. Hence, *thread ordering* can be satisfied by putting all retired instructions by a thread into a FIFO queue: the oldest instruction in *thread order* is the one at the head of the FIFO, whereas the youngest is the one at the tail. *Program order* is regenerated by marking *ordering instructions* with a *POP* value indicating what thread ordering FIFO queue must be accessed next in order to follow the desired sequential order. A global picture of this behavior is shown in Figure 39.

In the following subsections we give more details of how the software and hardware of this hybrid mechanism to reconstruct the original memory sequential order works.

3.5.1.1 Software

The Anaphase compiler is charge of adding the *POP* marks to *ordering instructions* after decomposing a region into speculative threads. Loads, stores, and unconditional branches are marked with a single *POP* value. In case of conditional branches, two *POP* marks are included; one is used when the branch is taken

and the other when the branch is not taken. In fact, it is not necessary to mark all *ordering instructions*; only those with a *POP* value different than the thread they are assigned to.

The algorithm to include *POP* marks in *ordering instructions* works as follows:

- Given a memory instruction (load or store) M_i assigned to thread T_i
  - M_i will be marked with a *POP* value T_k if there exists a memory instruction M_k following M_i assigned to thread T_k with no branch in between, being T_k and T_i different.
  - M_i will be marked with a *POP* value T_k if there is no other memory instruction M between M_i and the next branch B assigned to thread T_k, being T_i and T_k different.
  - Otherwise, there is no need to mark memory instruction M_i.
- Given a conditional branch B_i assigned to thread T_i
  - Mark B_i with a *POP* value T_k in its taken *POP* mark if the next ordering instruction when the branch is taken (it can be a branch or a store) is assigned to T_k, being T_i different than T_k. Otherwise, there is no need to assign a taken *POP* mark to B_i.
  - Mark B_i with a *POP* value T_k in its fallthru *POP* mark if the next ordering instruction when the branch is not taken (it can be a branch or a store) is assigned to T_k, being T_i different than T_k. Otherwise, there is no need to assign a fallthru *POP* mark to B_i.
- Given an unconditional branch B_i assigned to thread T_i
  - Apply the same algorithm as a conditional branch but only computing the taken *POP* value.

Indirect branches are specially handled because they can have many destinations. This scenario can be handled by forcing the first *ordering instruction* of all known destinations to be assigned to the same thread. Alternatively *nops* are used as *ordering instructions* in those destination paths where the first instruction fits better in a different thread.

Finally, for those *ordering instructions* that are replicated (i.e. they are assigned to multiple threads) the order among them is not important as long as the order with respect to the rest of the instructions is guaranteed. Hence, the algorithm chooses any arbitrary order among the individual instances of a replicated instruction.

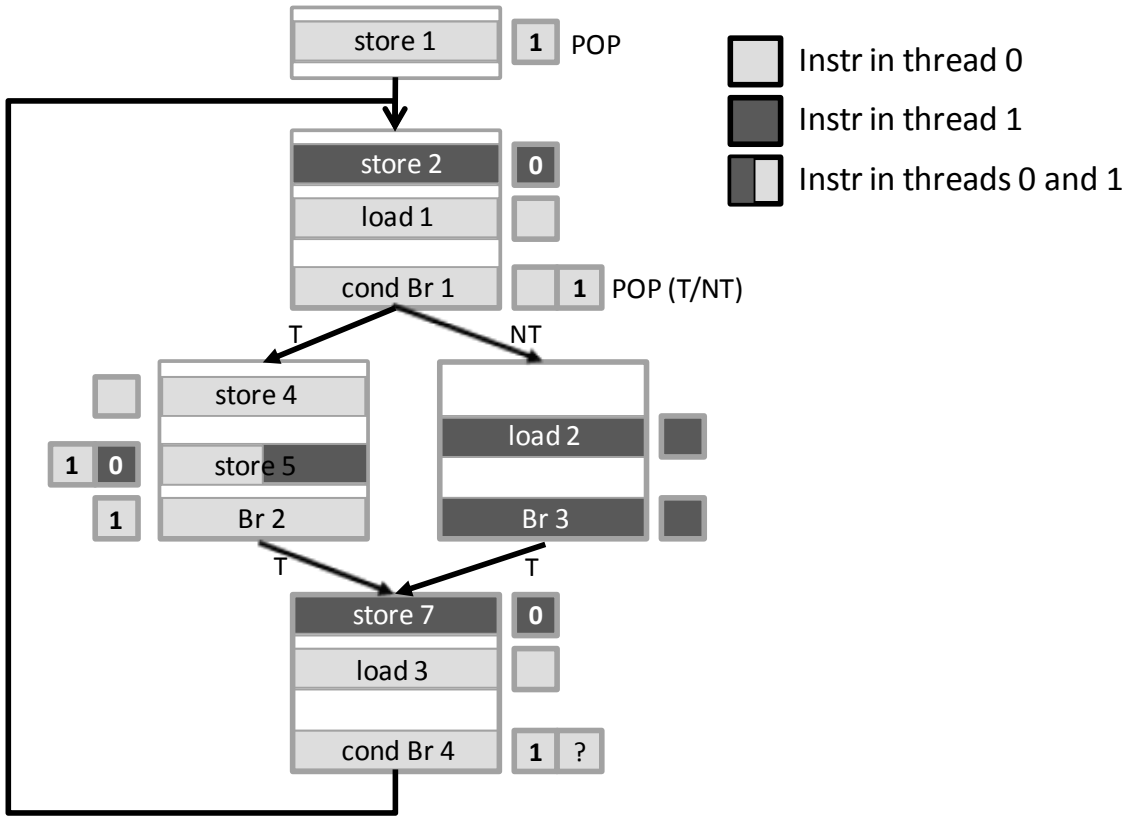


Figure 40. Example of *POP* marks in an optimized region decomposed into two threads

An example of how these *POP* marks are assigned to *ordering instructions* is shown in Figure 40. Note that for the sake of simplicity, in this example we consider that the optimized region is decomposed into two threads, so *POP* marks are just 1 bit wide. In the example "?" is used when there is no information about the following instructions because the whole control flow graph is not shown.

As can be seen, *store 0* has a *POP* bit of "1" because the next *ordering instruction* in sequential order (*store 1*) was assigned to speculative thread "1". *Load 1* does not need a *POP* mark because the next *ordering instruction* in sequential order (*cond Br1*) was assigned to the same thread 0. Similarly, *cond Br1* does not need a taken *POP* mark because when the branch is taken, the next *ordering instruction* was assigned to the same thread 0. However, it needs a fallthru *POP* mark because when the branch is not taken, the *next ordering* instruction *load 2* was assigned to thread 1. In this case, the mark is "1". As another particular case, *store 5* was assigned to both threads (it was replicated). In this case, we do not care about the order between its two instances. In the figure, we have marked the instance of *store 5* in thread 0 to go before the instance in thread 1 by not assigning a *POP* pointer to *store 4* in thread 0 and by

assigning *POP* pointers "1" and "0" to *store 5* instances in thread 0 and 1 respectively. However the algorithm could have chosen the other way around.

3.5.1.2 Hardware

The hardware of the ordering mechanism is placed in the *ICMC*. The *ICMC* is in charge of committing all memory instructions in the original *program order* using the *POP* bits. In particular, when a core retires an *ordering instruction*, the instruction is stored in the *memFIFO* associated to the core (see Figure 38). Then, the *ICMC* process and removes the instructions from the *memFIFOs* based on the *POP* bits. The value of the *POP* bits of the last committed instruction identifies the head of the *memFIFO* where the next instruction to commit resides[5].

We say that an instruction *retires* when it becomes the oldest instruction in a core and do the retirement. By contrast, we say that an instruction *globally commits*, or *commits* for short, when the instruction is processed by the *ICMC* because is the oldest in the tile. Note that memory instructions are committed by the *ICMC* when they become the oldest instructions in the system in original sequential order. Therefore, this is the order in which inter-thread memory dependences can be checked and store operations can update the shared cache levels and be visible outside the tile.

Note that *memFIFOs* allow each core to fetch, execute and retire instructions independently. The only synchronization happens when a core prevents the other core to retire instructions. A core may eventually fill up its *memFIFO* and stall until its retired instructions can leave the *memFIFO*. This situation occurs when the next instruction to be globally committed comes from a different core and this instruction has not retired yet.

3.5.2 Memory State Management

The Anaphase multi-core processor presents a memory hierarchy aimed to execute fine-grain speculative threads. The key distinguishing features of the memory hierarchy are the following:

- The design is specially targeted to fine-grain speculative threads. Since all SpMT paradigms explored so far use the coarse-grain approach, the proposed configuration is unique. Previous

[5] We have measured that a switch between FIFOs is performed each 1.7 instructions on average. This clearly shows the fine-grain granularity of the Anaphase approach.

memory hardware proposals for speculative multithreading [3][16][32][33][44][54][78] do not support the execution of speculative threads at the finer granularity provided by Anaphase.

- The design requires minimal changes in the core (just *V* bits per *L1* cache line), since most of them reside in the uncore. In fact, the design introduces little hardware complexity to a multi-core processor (see Section 3.6.2.1).
- The small overheads associated with committing or squashing the speculative state make this configuration suitable for SpMT approaches in which frequent checkpoints are taken. Frequent checkpoints are important for power and performance reasons.

In Figure 41 the memory hierarchy from the point of view of a single tile is shown. For simplicity the figure shows the memory subsystem for a 2-core tile multi-core processor but the hardware easily scales for more cores. The Anaphase hardware support involved in managing the memory state is highlighted in grey in the figure. As can be seen, it is composed by the Inter-Core Memory Coherence module (*ICMC*) and a few bits in *L1* and *L2* caches.

The memory subsystem works as a regular multi-core memory hierarchy when executing in *independent* mode. That is, the traditional cache coherence protocol (MESI [76] or any other) propagates and invalidates cache lines as needed. In this mode, the *ICMC* and the bits attached to the cache memories are not used. This section focuses on how the memory hierarchy works when speculative threads are executed in *cooperative* mode and what actions are taken when transitioning from one mode to the other.

In *cooperative* mode, the *L1* caches work on their own, each having a potential version of each piece of data. Therefore, modified values are not propagated to upper levels and do not invalidate cache lines in other cores. The *L2* cache is the level at which merging is performed and it is done following the original sequential program order orchestrated by the *ICMC*, as explained in previous Section 3.5.1. The *ICMC* is also responsible for checking that no memory violations occur. A memory violation occurs when a load executing in one core needs data generated by a store executed in another core. In the presence of a memory violation, a squash signal is propagated to all the components. In addition, the system decides to take a checkpoint at regular intervals so that it can rollback to a previous consistent state in case a memory violation is found. The frequency of checkpoints and how the architectural register state is kept is described in Section 3.5.3. When speculative threads finish their execution, or are squashed, and the tile transitions from *cooperative* to *independent* mode, the valid lines only reside at the *L2* cache. Hence, the speculative lines are cleared in the *L1* caches.

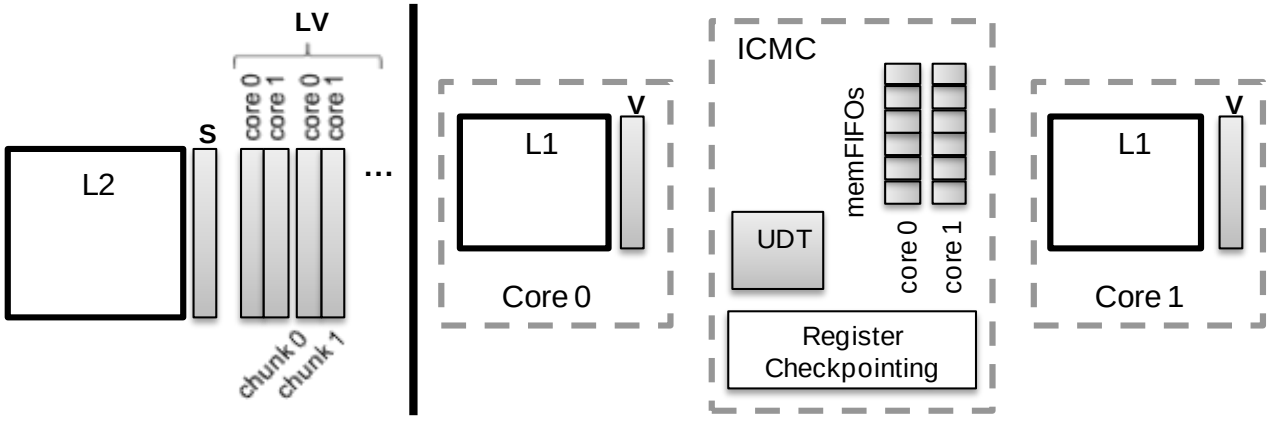


Figure 41. The memory hierarchy of a 2-core tile. Anaphase support is in grey.

In the following subsections the modifications in the *L1* cache, *L2* cache and the mechanisms of the *ICMC* are detailed.

3.5.2.1 L1 Cache

The *L1* caches are extended with a versioned bit (*V*) per line that is only used in *cooperative* mode and when transitioning between modes. This bit identifies a line that has been updated while executing the current speculative code region, that is, since the last transition from *independent* to *cooperative* mode.

In *cooperative* mode, the *V* bit of a line in one core is set when a store instruction executes in that core and updates that line similar to [65]. In this mode, *L1* data caches do not invalidate other *L1* caches when a line is updated. This allows versioning: each L1 cache may have a different version of the same datum. Moerover, speculative updates to the *L1* are not propagated (writtenthrough) to the shared *L2* cache. Store operations are sent to the *ICMC* which is in charge of updating the *L2* cache in the original order when they globally commit. Similarly, when a line with its *V* bit set is replaced from the *L1* it is not propagated to upper levels and its contents are discarded.

When transitioning from *independent* mode to *cooperative* mode, the *V* bits are unset. On the other hand, when the cores transition from *cooperative* to *independent* mode, all the *L1* lines with the *V* bit set are invalidated since the correct data resides in the *L2* and the *ICMC*. We have seen that invalidating all dirty speculative lines has a negligible impact on performance because our simulations show that there are very few mode transitions and squashes (see Section 3.6.2.2).

3.5.2.2 L2 Cache

The shared *L2* cache is extended with a speculative bit (*S*) per line and a set of *N* last version bits (*LV*) per chunk of information, where *N* is the number of cores in a tile (in Figure 41 since it is a 2-core tiled multi-core there are two *LV* bits per chunk). A chunk is the granularity at which memory disambiguation among speculative threads (and hence, memory violations) are detected, and ranges between a byte and a line. In the presented design we assume it is a byte.

The *S* bit indicates that a *L2* line contains speculative state. It is cleared when a checkpoint is performed and the memory is safe again. On the other hand, the *LV* bits indicate which core performed the last change to each chunk. In *cooperative* mode, these bits are set as stores globally retire in the original program order in the *ICMC* and they are not cleared until there is a transition back to *independent* mode (as opposed to the *S* bit, which is cleared between checkpoints). This is necessary to ensure that the system can capture memory violations that may happen at many checkpoints of distance.

When a store globally retires, it updates the corresponding *L2* line and sets its *S* bit to 1. As mentioned, such *S* bit indicates that the line has been modified since the last checkpoint. Once a new checkpoint is taken, the *S* bits are cleared. In case of a misspeculation, the threads are rolled back and the lines with an *S* bit set are invalidated. Hence, when a non-speculative dirty line is to be updated by a speculative store, the line must be written back to the next memory level in order to have a valid non-speculative version of the line somewhere in the memory hierarchy. Since speculative state cannot go beyond the *L2* cache, an eviction from the *L2* of a line that is marked as speculative (*S*) implies rolling back to the previous checkpoint to resume executing the original application. In that case, the speculative threads are squashed because the current cache configuration cannot hold the entire speculative memory state since the last checkpoint. Since checkpoints are taken regularly, this happens rarely as observed from our simulations (see Section 3.6.2.2).

On the other hand, the *LV* bits indicate the core that has the latest version of a particular chunk (i.e. byte). When a store globally retires, it sets the *LV* bits of the modified chunks belonging to that core to one and resets the rest. If a store is tagged as replicated (executed by several cores), several cores will have the latest copy. In this case, the *LV* bits are set to 1 for all the cores that globally retire the replicated store.

Upon the global commit of a load, the *LV* bits of the corresponding chunks are checked to see whether the core that executed the load was the core having the latest version of the data. If any of such *LV* bits have a value of 0 for the corresponding core a violation is detected and the speculative threads are

squashed. This is so because as each core fetches, executes and retires instructions independently and the *L1* caches also work decoupled from each other, the system can only guarantee that a load will read the right value if this was generated in the same core.

It is important to notice that in case a store misses in the *L2* cache, global retirement is stalled until the line is present in the *L2* cache. On the other hand, if a load misses in the *L2* cache a fill request is sent to the next level in the memory hierarchy and the load globally retires correctly.

3.5.2.3 ICMC

As already mentioned, the Inter-Core Memory Coherence module (*ICMC*) resides in the uncore of each of the multi-core tiles and is in charge of: (i) globally retiring memory instructions in the original program order, (ii) properly updating the *L2* cache, (iii) checking for memory violations among the speculative threads, and (iv) handling register and memory checkpoints. Figure 41shows the main structures of this module.

The *memFIFOs* receive the memory instructions of each core of the tile as they retire in *thread order*. The content of an entry in the *memFIFOs* is detailed in Figure 42(a). It consists on: 2 *TYPE* bits that identify the type of instruction (load, store, branch); *N POP* bits (where $N = log_2 M$, *M* being the number of cores in a tile; in this example two cores per tile so *N* is 1 bit); 64 bits for memory address (@); 32 bits for the *value* in case of a store; 2 bits to describe the *size* of the memory access; and 1 bit to mark replicated (*rep*) instructions. As has been described in the previous subsection, replicated instructions are marked to avoid the *ICMC* to check for a dependence violation for these instructions.

3.5.2.3.1 UDT

The Update Description Table (*UDT*) is a table in the *ICMC* that describes the *L2* lines that are going to be updated by store instructions located in the *memFIFO* queues. The purpose of the UDT is to delay any fill from the shared *L2* cache to the *L1* cache as long as there are still some stores pending to update that line. This way we avoid filling an *L1* with a stale line from the *L2*. In particular, a fill to the *L1* of a given core is delayed until there are no more pending stores in the *memFIFOs* for that particular core (there is no any entry in the *UDT* for the line tag).

Note that there is no need to wait for stores from other cores that access the same line, since in case of a memory dependence the *LV* bits will already detect it, and in case that the cores access different parts of the same line, the *ICMC* will properly merge the updates at the *L2*.

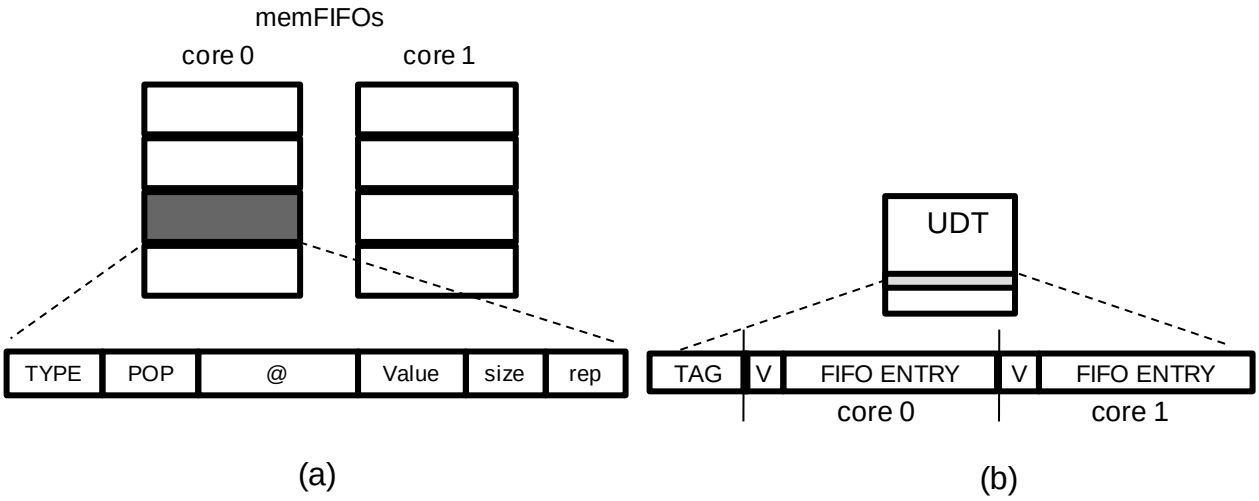


Figure 42. (a) Contents of the *memFIFOs* and (b) the UDT in the *ICMC* of a 2-core tile

An *UDT* entry as shown in Figure 42(b) has: the *TAG* identifying the *L2* line, plus a valid bit (*V*) attached to a *memFIFO* entry id for each core. The *memFIFO* entry id is the entry number of the last store that updates that line. This field is updated every time a store is appended to a *memFIFO*. If a store writes a line without an entry in the *UDT* then it allocates a new entry. By contrast, if a globally retired store is pointed by the *memFIFO* entry id then its valid bit is set to false; and if all valid bits are false then the entry is removed from the *UDT*. Finally, when transitioning from *cooperative* to *independent* mode, the *UDT* is cleared.

3.5.3 Register State Management

In the Anaphase multi-core processor, the architectural register state is distributed between the cores physical register files of a tile. In order to allow cores running in *cooperative* mode to work asynchronously, the *ICMC* is in charge of merging and building the register architectural state. Therefore, a core does not have to generate the complete architectural state. Instead, this can be partially computed by multiple cores.

An important feature of this design is the checkpointing mechanism, which is a hybrid software / hardware scheme specially designed to support a fine-grain threading scheme. Checkpointing is performed by hardware at the places decided by the compiler through a *CKP* instruction. This instruction marks the place where the checkpoint can be taken. We consider inserting a *CKP* instruction at the beginning of any loop belonging to optimize regions. Thanks to this mechanism each core takes a partial register checkpoint at regular intervals. From these partial checkpoints, the core that is retiring the oldest

instructions in the system can recover a complete valid architectural register state. These regular checkpoints allow the system to normally throw away very little work when a rollback occurs.

More specifically, the inter-core architectural checkpointing scheme that we propose has the following key benefits:

- It allows a core to retire instructions, reclaim execution resources and keep doing forward progress even when other cores are stalled. This is a clear advantage versus other schemes like the one proposed by Core Fusion [36] where the retirement of all cores involved in the execution of the same application works in lock-step. Thus, a stalled core prevents other cores from retiring instructions, which limits their forward progress.
- Although Core Fusion could improve its forward progress when a core stalls by enlarging the Reorder Buffers (*ROB*s) of the cores, its forward progress would still be limited because it would run out of physical registers very soon. Note that the physical register files are hardware components that are less scalable than the *ROB*. By contrast, this scheme allows safe early register reclamation so that it allows forward progress increasing very little the pressure on the register files. Our experiments show that this early register reclamation removes outliers that would get up to 35% slowdown when a conservative register reclamation scheme such as that of Core Fusion is implemented.
- Checkpoints are taken very frequently (every few hundreds of instructions) so that the amount of wasted work is very little when we have to rollback because either an interrupt or data misspeculation.
- A core does not have to generate the complete architectural state as it happens in previous SpMT schemes but the architectural state can be partially computed by multiple cores instead. Thus, this design is specially targeted to fine-grain threading schemes like Anaphase.
- Cores do not have to synchronize in order to get the architectural state at a specific point. This mechanism allows cores to work asynchronously. The technique merges and builds the architectural state.

Figure 43 shows a conceptual view of the checkpointing mechanism. This mechanism conceptually creates a *ROB* where instructions are stored in the order they should be globally committed. However, since the threads execute asynchronously, the entries in this conceptual *ROB* are not allocated sequentially. Instead we have areas where we do not know either how many nor the kind of instructions to

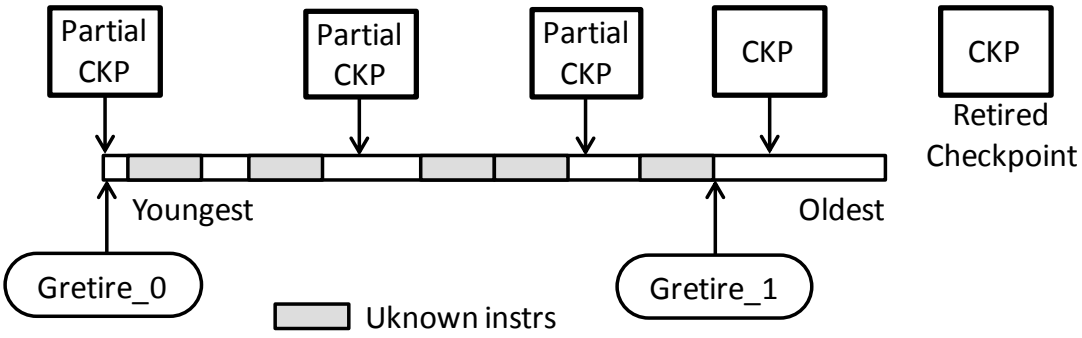


Figure 43. Conceptual view of the register checkpointing mechanism

be allocated there. In this example, with two speculative threads running on a 2-core tile, this situation may happen if for instance core0 is executing a region of code that should be committed after the instructions executed from core1. In Figure 43, *GRetire_C* points to the last instruction retired by core *C*. As it can be seen, core0 goes ahead of core1 so that there are gaps (shown as shaded regions) between *Gretire_0* and *Gretire_1*.

Checkpoints taken by the core that retires the youngest instructions in the system are always *partial checkpoints*. We cannot guarantee that this core actually produces a whole architectural state. By contrast, checkpoints taken by the core that does not retire the youngest instruction in the system are *complete checkpoints* because it knows the instructions older than the checkpoint that other cores have executed. Therefore, it knows where each of the architectural values resides at that point.

The reason why a core takes periodic checkpoints even when they are partial, as core0 in the example, is because all physical registers that are not pointed by these *partial checkpoints* are reclaimed. This feature allows this core to make forward progress with little increase on the pressure over its register file. Moreover, when other core reaches the checkpoint, core1 in the example, it is guaranteed that the registers containing the values produced by core0 that belong to the architectural state at this point have not been reclaimed so that we can build the complete checkpoint with the information from core1. On the other hand, registers being allocated in core0 that did not belong to the checkpoint because they were overwritten by core1 can also be released.

Note that the core that goes ahead is not always the same. This role changes depending on the decomposition of instructions among threads. Therefore, the role of complete checkpointing is moving among the cores.

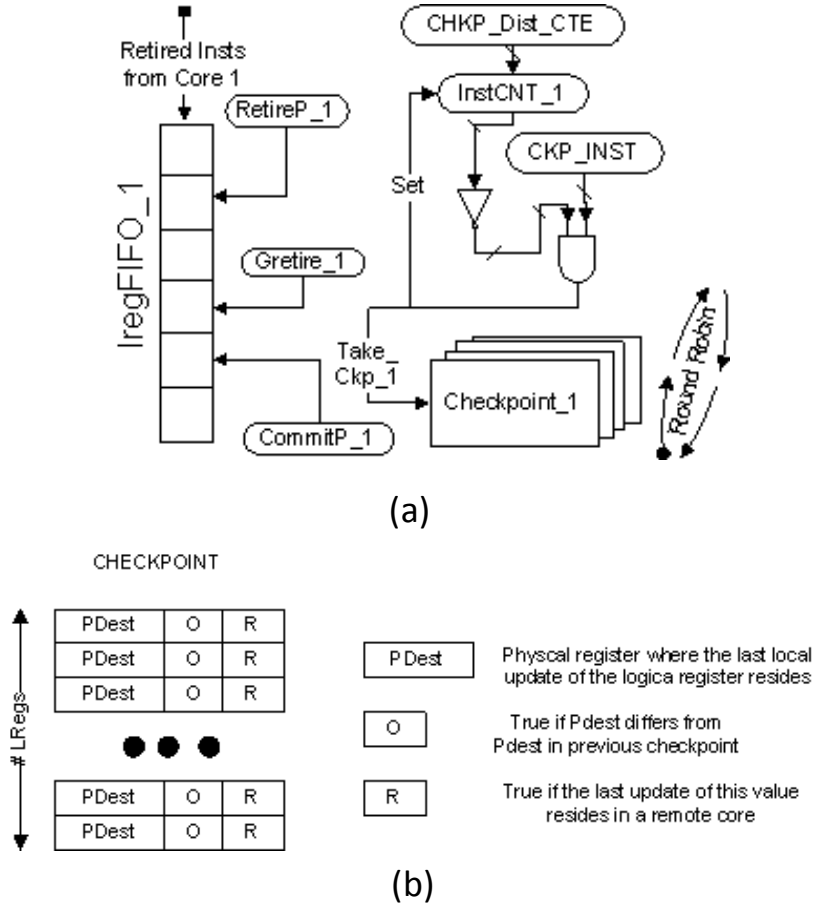


Figure 44. (a) Register checkpointing mechanism (per core) and (b) contents of a checkpoint

At a given time, a complete checkpoint has pointers to the physical registers in the register files (in any core of the tile) where the value resides for each logical register. A checkpoint can be released and its physical registers reclaimed when all instruction have been globally committed and a younger checkpoint becomes complete.

A checkpoint is taken when a *CKP* instruction inserted by the compiler is found, and at least a minimum number of dynamic instructions have been globally committed since the last checkpoint (*CKP_DIST_CTE*). This logic is shown in Figure 44(a). This *CKP* instruction has the IP of the recovery code which is stored along with the checkpoint, so that when an interrupt or data misspeculation occurs, the values pointed by the previous checkpoint are copied to the core that will resume the execution of the application. We consider that this copy is done by the checkpointing hardware. However, it could also be done by software as the beginning of a service routine.

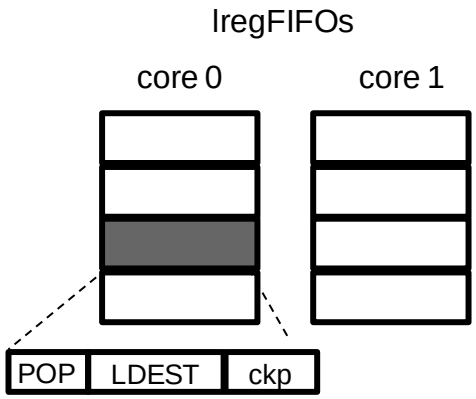


Figure 45. Contents of the *lregFIFOs* in the *ICMC* of a 2-core tile

A checkpoint includes the IP of the instruction where the checkpoint was created, the IP of the rollback code, and an entry for each logical register in the architecture. As shown in Figure 44(b), each of these entries have: the physical register (*PDest*) where the last value produced prior to the checkpoint resides for that particular logical register; the overwritten bit (*O*) which is set to 1 if the *PDest* identifier differs from the *PDest* in the previous checkpoint; and the remote bit (*R*) which is set to 1 if the architectural state for the logical register resides in another core.

The hardware components of the register checkpointing mechanism are placed in the *ICMC* and comprise:

- One FIFO queue (*lregFIFO*) per core where all retired instructions that writes a logical register allocate an entry. Each entry, as shown in Figure 45, consists of: 1-bit field named *ckp* that is set to 1 in case there is an architectural state checkpoint associated to this entry; the identifier of the logical register written by the instruction (*LDest*); and the *POP* bits to identify which thread contains the next instruction in program order (see Section 3.5.1 for more details).
- A set of pointers per *lregFIFO*: (1) a *RetireP* pointer to the first unused entry of the *lregFIFO*, where new retired instructions allocate an entry; (2) a *CommitP* pointer to the oldest allocated entry in the *lregFIFO* which is used to deallocate the *lregFIFO* entries in order; (3) a *Gretire* pointer to the last entry in the *lregFIFO* we visited in order to build a complete checkpoint.
- A pool of checkpoint tables per *lregFIFO*. The number of these tables defines the maximum number of checkpoints we can have in-flight. Each pool of checkpoints works as a FIFO queue where checkpoints are allocated and reclaimed in order.

The mechanism works as follows. Every time a core retires an instruction that produces a new architectural register value, a new entry is allocated in the corresponding *lregFIFO*. Then, it reads the entry in the active checkpoint for the logical register it overwrites. In case the *O* bit is set, the *PDest* identifier stored in the entry is reclaimed. Then, the *O* bit is set and the *R* bit unset. Finally, we update the *PDest* field with the physical register allocated by the retired instruction. Once we found the starting point of a new checkpoint, we copy the current checkpoint to the next checkpoint making this next checkpoint the active checkpoint. In such case, we reset all *O* bits in the new active checkpoint.

When *GRetire* pointer does not match *RetireP*, we do not have to take any action, because we are the youngest instruction in the system. Otherwise, if the *GRetire* pointer matches the *RetireP* pointer means that this instruction is not the youngest instruction and so we are performing complete checkpoints. For that, we check the *POP* bit and in case it points to other core *C*, we use the *GRetire* pointer of *C* to walk over its *lregFIFO* until we find an entry with a *POP* pointer pointing to us again. For every entry we visit, we read the *LDest* value and update our active checkpoint: in case the *O* bit is set, we reclaim the physical register identifier written in *PDest*. Then, in any case we reset the *O* bit, set the *R* bit and update the *PDest*. When an entry with the *ckp* bit set to 1 is visited, the partial checkpoint with the information of our active checkpoint is completed. This merging involves reclaiming all *PDest* which in the partial checkpoint has the *O* bit set and the *R* bit in our active checkpoint is reset. Then, we update our active checkpoint resetting the *O* bit of these entries.

3.5.4 Related Work

In the area of multi-core architectures, a significant amount of research efforts have been devoted to come up with appropriate design points to deal with ILP and TLP. These include alternatives comprising either a large number of tiny cores [6], a limited number of big cores [67], or heterogeneous designs [39]. Some researchers propose adaptive architectures and reconfigurable hardware where the characteristics of the architecture dynamically adapt to the parallelism of the applications. We could say that Anaphase is a hybrid scheme that combines the benefits of speculative multithreading (i.e. exploiting speculative TLP) with the adaptability of such architectures. In that sense, the techniques more closely related to Anaphase are those that pursue the idea of combining or fusing cores [36][48][107].

Core Fusion is a hardware only scheme that can be seen as a natural evolution of clustered microarchitectures [11][18]. In a clustered microarchitecture, instructions are distributed among the clusters statically at compile time, or dynamically through a steering logic. Instead, Core Fusion involves the task of distributing instructions in cores rather than clusters. Thus, starting from a single-threaded

code, Core Fusion decomposes a dynamic stream of instructions into different hardware contexts. Therefore, Core Fusion exploits the concept of non-speculative multithreading execution where instructions are distributed in a fine-grain fashion. However, the parallelism exploited by core fusion techniques is limited to ILP. In contrast, Anaphase is able to exploit TLP and MLP, in addition to ILP.

The closest work similar to our scheme is Fg-STP [81], which is a purely hardware based scheme. Fg-STP is able to exploit TLP and do thread partitioning at instruction granularity, like Anaphase. In Fg-STP, the hardware is responsible for thread decomposition, detecting violations, recovery on violations and committing in the original program order. Although this has the additional benefits of being able to run unmodified program binaries which are compiled for single thread execution, the decomposition algorithm and the exploited heuristics are much less powerful than the ones in Anaphase.

3.6 Experimental Evaluation

3.6.1 Framework

This section describes the framework that has been used for conducting the performance evaluation of the Anaphase architecture. The whole architecture has been modeled in detail, including the Anaphase compiler and the Anaphase multi-core processor.

The Anaphase speculative thread decomposition scheme has been implemented on top of the Intel® production compiler (*icc*), whereas the performance of the optimized binaries has been evaluated through a detailed cycle accurate simulator. The simulator models the tiled x86 multi-core architecture described in section 3.5. For this evaluation only one tile has been used. Two types of cores have been considered to represent different CMP scenarios. One out-of-order core, called *Medium*, that is similar to current CMP cores [67], and another called *Tiny*, where main structures have been reduced to half the size, which represents a likely core on a many-core environment. The main parameters that we have considered are shown in Table 6. The shadowed fields on the table are per core. Regarding the *ICMC* structures, our studies show that 1024 entries per *mem/lreg FIFO* and 64 entries in the *UDT* for the *Medium* core gives us almost the maximum performance we can get with the threading scheme we are evaluating.

For evaluating the proposed architecture, we have selected the SPEC2006 benchmark suite compiled with *icc –O3*. In total, 12 SpecFp and 12 SpecInt benchmarks have been optimized with Anaphase using the train input set for profiling information. Representative loops of the train execution have been selected to be speculative parallelized with Anaphase based on 20M instruction traces

Parameter	*Tiny* Core	*Medium* Core
Fetch, Retire and Issue width	2	4
Pipeline stages	7	7
Branch Predictor GShare history bits/table entries	12/4096	12/4096
Sizes ROB/Issue Queue/MOB	48/16/24	96/32/48
Miss Status Holding Registers per core	8	16
L1 and ICache size/ ways/ line size/ latency	16KB/4/32B/2	32KB/4/32B/2
Mem Fifos / Ireg Fifos Size	512	1024
UDT Size	32	64
L2 size/ ways/ line size/ latency (round trip)	4MB/8/32B/32	
Memory latency	328	
Explicit communication penalty	32	
Overhead spawn / commit (cycles)	64 / 64	

Table 6. Anaphase multi-core processor configuration parameters

generated with the PinPoint tool [75]. In these traces, those outer loops that account for more than 500K dynamic instructions have been selected for thread decomposition. Although Anaphase can decompose a loop into any number of speculative threads, for the studies presented in this thesis we have focused to partitioning each loop into two threads and evaluate the performance on the 2-core tile multi-core processor described before. We believe that focusing in two thread decomposition is enough to show the benefits and characteristics of the proposed model and at the same time is an interesting scenario on current multi-core systems. However, studying the scalability of the system for more than two threads is part of the future work, as will be described in the following chapter.

For each of the selected loops, we have generated multiple partitions using different replication thresholds and loop unrolling factors, as described in Section 3.4.2. In particular, we have considered up to 7 different thresholds for limiting the replication: 0 (meaning that all inter-thread dependences are communicated), 48, 96, 128, 256, 512 and an unbounded threshold (meaning that all the inter-thread dependences are pre-computed). Furthermore two unrolling factors have been considered: (i) no unrolling; and, (ii) unroll by the number of thread (i.e. 2 in our experiments). Hence, we have generated 14 different versions for each loop and have chosen at compile time the best one based on the objective function described in Section 3.4.2.2.3.

In order to conduct the performance studies, for each benchmark we have randomly selected traces of 100M instructions of the reference input set execution starting with the head of each of its optimized loops. Results for each benchmark are then reported as the addition of all its simulated traces. On average, about 10 traces have been generated per benchmark. We have measured that on average the optimized loops found in these traces cover more than 90% of the whole reference input set execution.

Finally, in our studies we have considered two configurations for comparing against Anaphase:

1) A coarse-grain speculative loop parallelization scheme that assigns odd iterations to one thread and even iterations to the other. This scheme is similar to traditional SpMT schemes, where big chunks of consecutive instructions (i.e. loop iterations) are assigned to different threads. In this partitioning scheme, inter-thread data dependences have been equally handled with the techniques proposed in the Anaphase decomposition algorithm (see Section 3.4.2.2.2). For our performance studies, the resulting threads run on the same hardware support than Anaphase threads.

2) A scheme based on dynamic non-speculative fine-grain thread decomposition that closely models the Core Fusion proposal [36]. This hardware-only scheme has been implemented on top of the 2-core tile multi-core processor with the parameters shown in Table 6. In this scheme, inter-core communication is performed by specialized hardware and we have assumed a latency of 2 cycles.

3.6.2 Results

In the following sections we present the results of several studies that we conducted with the objective of evaluating and characterizing the Anaphase architecture.

3.6.2.1 Optimized Regions Characterization

We have characterized the regions we have selected to be optimized for each benchmark. Table 7 shows the average size of the regions (i.e. loops) decomposed with Anaphase expressed in instructions excluding extra instructions added through replication.

Note that the average static size of the regions ranges from a few hundreds of instructions to thousands of instructions. This large number of instructions in the PDG advocates for the use of smart algorithms and heuristics to shred the code into speculative threads. This explains some of the decisions exposed in Section 3.4.2 for reducing the search space.

Spec FP	Static	Dynamic	Spec INT	Static	Dynamic
bwaves	526	2.4M	astar	682	1.2M
gamess	11.5K	28K	bzip2	946	6.3M
GemsFDTD	331	361K	gcc	11K	14K
lbm	467	2M	gobmk	17.6K	3K
leslie3d	10.6K	3.5M	h264ref	6K	4.3K
milc	345	4K	hmmer	775	26K
namd	1.2K	3.8M	libquantum	207	25M
povray	16.6K	94M	mcf	1.1K	600K
soplex	6.5K	900K	omnetpp	13.6K	13.7K
sphinx	228	1.5M	perlbench	8.3K	218K
tonto	5K	8.4K	sjeng	7.7K	11K
wrf	1.8K	80K	zeusmp	1.2K	3.2M

Table 7. Average static and dynamic size of optimized regions from Spec2006 benchmarks

The dynamic size of a region is the amount of dynamic instructions committed between the time the threads are spawned and the time the threads finish (either correctly or through a squash). As we can see, this number tends to be very large and may tolerate bigger overheads for entering and exiting an optimized region (we currently assume 64 cycles in each case). However, although regions are dynamically big, the amount of work that is thrown away when a region is rolled back (in number of dynamic instructions) is much smaller because the hardware takes regular checkpoints, as we discuss later in this section.

3.6.2.2 Performance Characterization

3.6.2.2.1 Fine-grain versus Corse-grain Thread Decomposition

The objective of the following study is to assess the benefit of fine-grain thread decomposition. Figure 46 shows the performance of the Anaphase fine-grain decomposition scheme compared to a coarse-grain speculative loop parallelization scheme that assigns odd iterations to one thread and even iterations to the other. Threads run on a 2-core tile multi-core processor with *Medium* cores (see Table 6). Performance is reported as speed-up over execution on a single core. For this comparison, the same loops have been parallelized with both decomposition schemes and inter-thread data dependences have been equally handled with the techniques proposed in our decomposition algorithm.

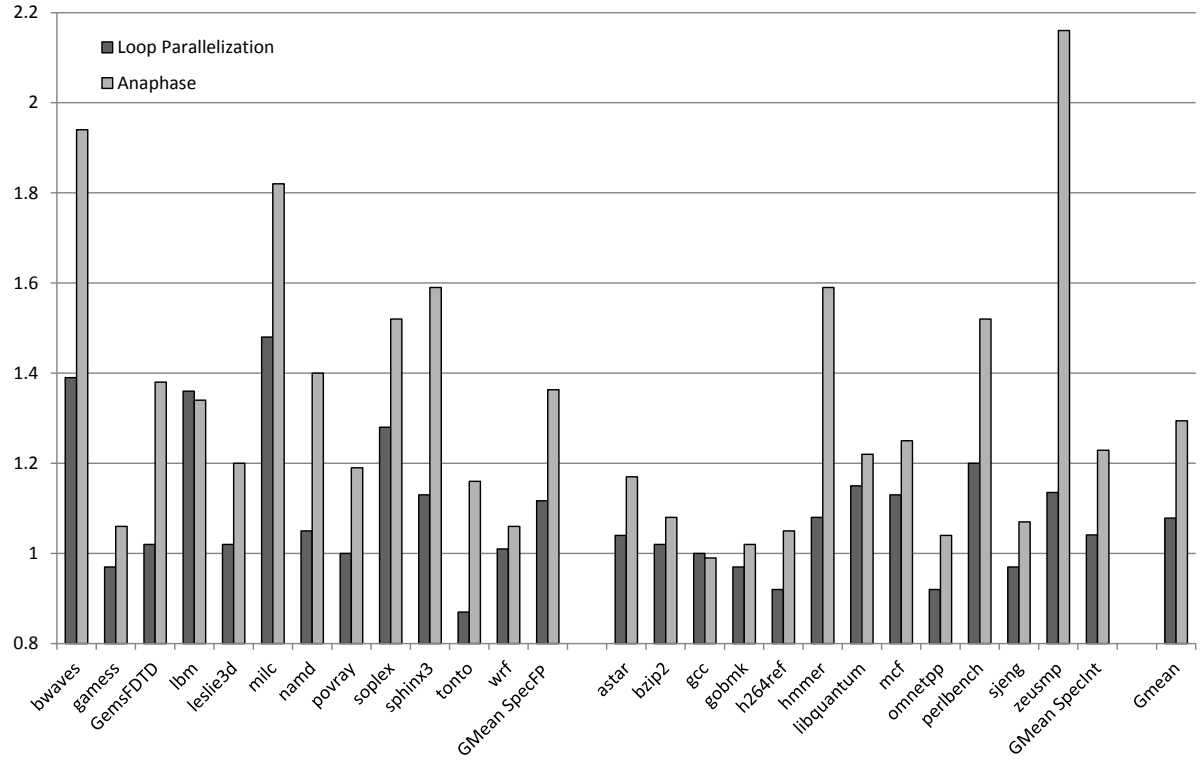


Figure 46. Anaphase performance compared to coarse-grain loop parallelization

As can be seen, Anaphase clearly outperforms the loop parallelization scheme. An average speed-up of 30% is observed, whereas loop parallelization just achieves 8%. For very regular benchmarks, like *bwaves*, *lbm*, *milc*, *sphinx3*, *perlbench*, and *zeusmp*, where loop parallelization performs well, Anaphase significantly benefits from exploiting more MLP thanks to its fine-grain decomposition and the delinquent load heuristic. Moreover, Anaphase is able to exploit TLP on those irregular and hard to parallelize benchmarks, like *astar*, *bzip2*, *gobmk*, *sjeng*, among others, where conventional coarse-grain decomposition schemes fail due to the presence of many inter-thread dependences. On the other hand, results for *gcc* show a slight slowdown with respect to loop parallelization and single-thread execution. This is mainly due to a very low coverage as we will explain later in Subsection 3.6.2.2.3. Furthermore, note that in a particular case (*zeusmp*) super-linear speed-up is achieved by a combination of doubling computation resources with respect to single core and a better exploitation of memory-level parallelism (MLP).

Overall, results show that Anaphase fine-grain thread decomposition is a very effective technique to speed-up single thread execution on regular and irregular applications, and mitigates the constraints of traditional coarse-grain SpMT schemes.

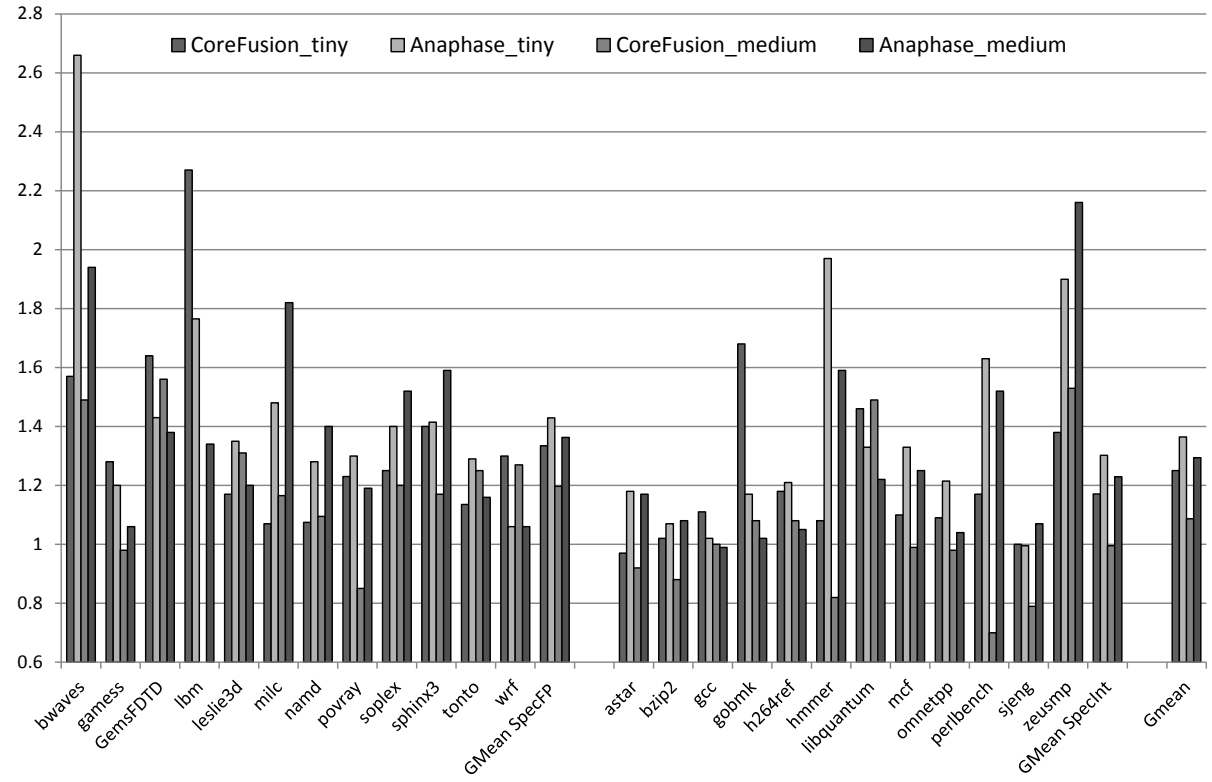


Figure 47. Anaphase performance compared to Core Fusion

3.6.2.2.2 Anaphase versus Core Fusion

Another scheme that exploits fine-grain thread decomposition techniques to boost single thread performance on CMPs is Core Fusion [36] (see Section 1.3.4 for more details). In order to compare Anaphase against Core Fusion, we have implemented a scheme based on dynamic non-speculative fine-grain thread decomposition in the Anaphase multi-core processor that closely models the Core Fusion proposal.

Figure 47 shows the performance of the Anaphase scheme compared to Core Fusion. Performance for both schemes is reported as speed-up over execution on a single core. For this comparison, both *Tiny* and *Medium* core configurations have been evaluated.

As can be seen, Anaphase clearly outperforms the Core Fusion scheme on both core configurations. An average speed-up of 37% and 30% is observed for the *Tiny* and *Medium* core respectively, whereas Core Fusion achieves 25% and 9%. Note that on a CMP with small cores, where the amount of ILP each core is able to exploit is limited by resources, Core Fusion performs significantly better than on a CMP with bigger cores. Other experiments have shown its higher potential on smaller cores [4][12]. On the other hand, Anaphase performs well on both CMP environments since it better exploits different sources

of parallelism: ILP, TLP, and MLP. Regarding TLP, we have measured that for these benchmarks Anaphase is able to speculate on a window comprised of several thousands and even a few millions of instructions. This is measured as the distance between the youngest instruction and the most senior instruction (not yet globally retired) in the system. Regarding MLP, we have observed that exploiting this parallelism is key for some benchmarks like *bwaves*, *lbm*, *milc*, *hmmer*, and *zeusmp*. As can be seen in this study (Figure 47) and the previous one (Figure 46), Anaphase significantly benefits from exploiting more MLP thanks to its fine-grain decomposition and the delinquent load heuristic. On the other hand, differences between the train and reference input sets affect the profiling information used by Anaphase and may cause that available MLP remains to be exploited. This effect can be observed in the *lbm* benchmark, where Core Fusion decomposition ends up in a better distribution of load misses.

Other features that may affect Anaphase performance compared to Core Fusion are inter-thread communications and the amount of replication due to control instructions. Regarding communications, although we have observed that for some benchmarks inter-thread communication latency is up to a few thousand cycles on average, the Anaphase decomposition scheme is able to adapt to it. On the other hand, we have verified that for some benchmarks, like *libquantum*, Core Fusion benefits a lot from having a dedicated communication mechanism with only 2 cycles latency. In addition, Core Fusion requires a tight synchronization in the front-end in order to steer instructions among cores in sequential order. Anaphase allows asynchronous execution of each of the threads and thus no modifications on the front-end at the expenses of having to replicate some control instructions. We have observed that the impact on performance of this control replication is about 5% on average, and for some benchmarks like *gcc* and *gobmk* this is one of the causes why Anaphase performs worse than Core Fusion.

Finally, in some benchmarks where Core Fusion performs better than Anaphase, like *gemsFDTD*, *wrf*, *gcc*, *gobmk*, *h264ref*, and *libquantum*, Anaphase suffers from low coverage of the optimized regions, as we will explain in the following subsection.

3.6.2.2.3 Anaphase Activity Breakdown

In order to have more insight of the execution of Anaphase threads, we have performed a study showing how execution cycles are spent. Figure 48 shows the active time breakdown for the execution of the different benchmarks optimized with Anaphase. We show results for the *Medium* core; results for the *Tiny* core are not shown but lead to the same conclusions.

The first thing to notice is that the overhead of the Anaphase execution model is extremely low for all benchmarks, even though we conservatively assume 64 cycles for entering and exiting an optimized

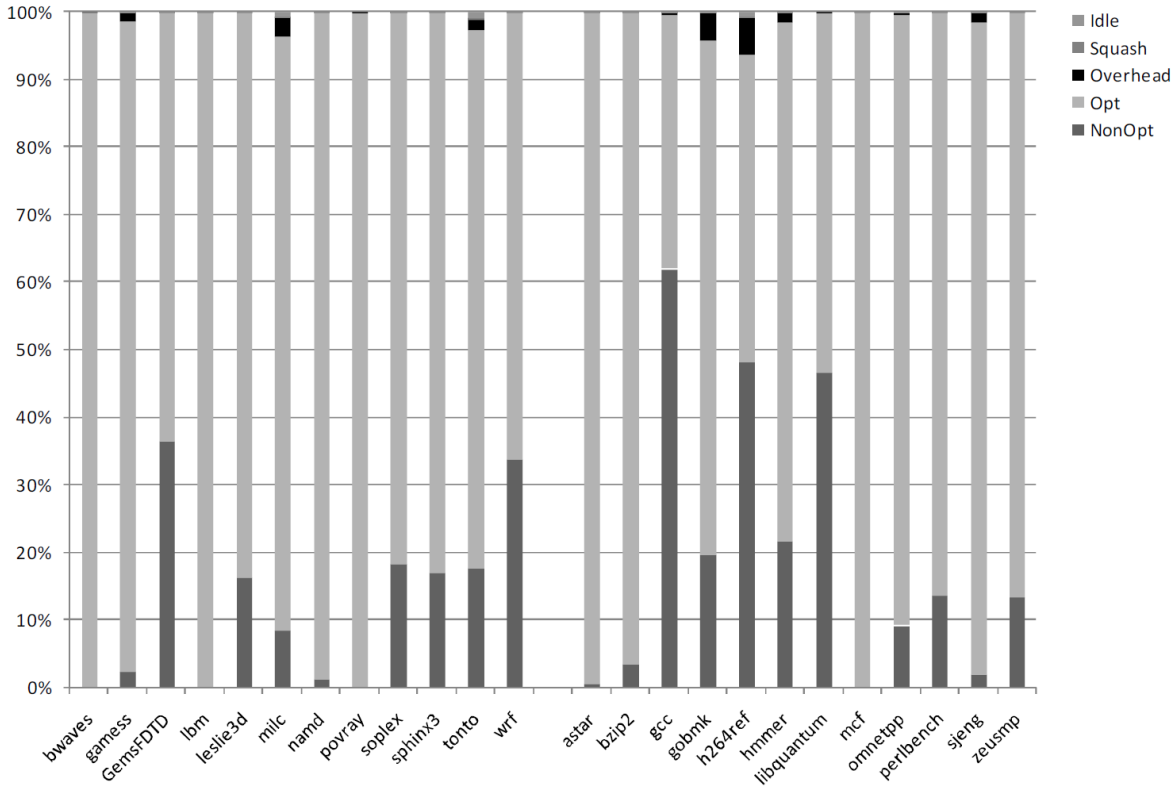


Figure 48. Anaphase activity breakdown

region, due to the spawn and join operations. Benchmarks *gobmk* and *h264ref* present the larger overhead, 4% and 6% respectively, because regions are smaller and more frequently spawned than for the rest of benchmarks.

Thanks to the workload balance heuristic and the checkpointing support, the Anaphase scheme is also very effective on reducing the idle time of the cores and the amount of work that is thrown away on a squash. For all benchmarks both sources of overhead represent less than 1% of the active cycles. It is worth to point out that on average less than 2% of the optimized regions turn out to be squashed due to a memory misspeculation at some point during their execution. However, the percentage of squashed regions is close to 95% for some benchmarks like *bwaves* and *lbm*. Notice that even in these cases speed-up is still achieved. This is so because such squashes occur long time after entering the region (i.e. the loop) and the proposed fine-grain speculative paradigm takes checkpoints regularly. Hence, although for these benchmarks there are very high chances that a dependence not observed during profiling that requires a squash arises at some point during the execution of a loop, the last valid checkpoint is close enough to have negligible effects on performance.

In addition, we have measured that on average more than 50% of the squashes are due to evictions of speculative lines on the *L2* cache. This strengthens the importance of having a fine-grain checkpointing mechanism as the one proposed for Anaphase. Our evaluations show that in order to take a checkpoint every 200 original instructions on average, less than 8 live checkpoints are required for our system.

As expected, the time spent in the optimized loops is very high in almost all the benchmarks. However, for some benchmarks like *gcc*, *h264ref*, and *libquantum*, the time spent in non-optimized code is greater than 40% due to the low coverage of the optimized code. Coverage is mainly a caveat of our research infrastructure. Since our analysis works with traces and not with the complete binaries, the hottest loops chosen with the train input set do not sometimes correspond with the hottest parts of the traces used with the reference input set. This explains the poor performance of these benchmarks.

3.6.2.2.4 Anaphase Thread Decomposition Overheads

As we have seen in the previous study (see Figure 48), benchmarks spend the majority of time running instructions from optimized regions. However, as described in Section 3.4.2.2.2, the Anaphase thread decomposition scheme incurs on overheads in form of additional instructions for managing inter-thread dependences and intra-thread control.

The amount of extra instructions introduced by the Anaphase decomposition scheme to solve inter-thread dependences and to handle intra-thread control is shown in Figure 49. These extra instructions are divided into two groups: replicated instructions and communications. The former includes all instructions that are replicated in order to satisfy a dependence locally (through a *p-slice*), plus all instructions that are replicated to manage the control. The latter, on the other hand, are additional instructions because of explicit communications. On average, about 30% of additional instructions compared to single-thread execution are introduced. However, for some benchmarks like *povray*, *gobmk*, and *sjeng*, the Anaphase decomposition scheme introduces more than 70% of replicated instructions. Although, this large amount of replication code does not imply a slowdown in performance, it may imply an increase in energy. One important thing to notice is that in Anaphase *p-slices* are conservative and do not include any speculative optimization. Previous work, including the Mitosis proposal presented in Chapter 2, has shown that *p-slices* can be significantly reduced through speculative optimizations with a slight impact on accuracy [30][107].

On the other hand, explicit communications, only account for 3% of additional instructions on average. This short amount of extra instructions has proven to be a very effective technique to handle inter-thread dependences. We have verified this fact with a scheme that does not allow explicit

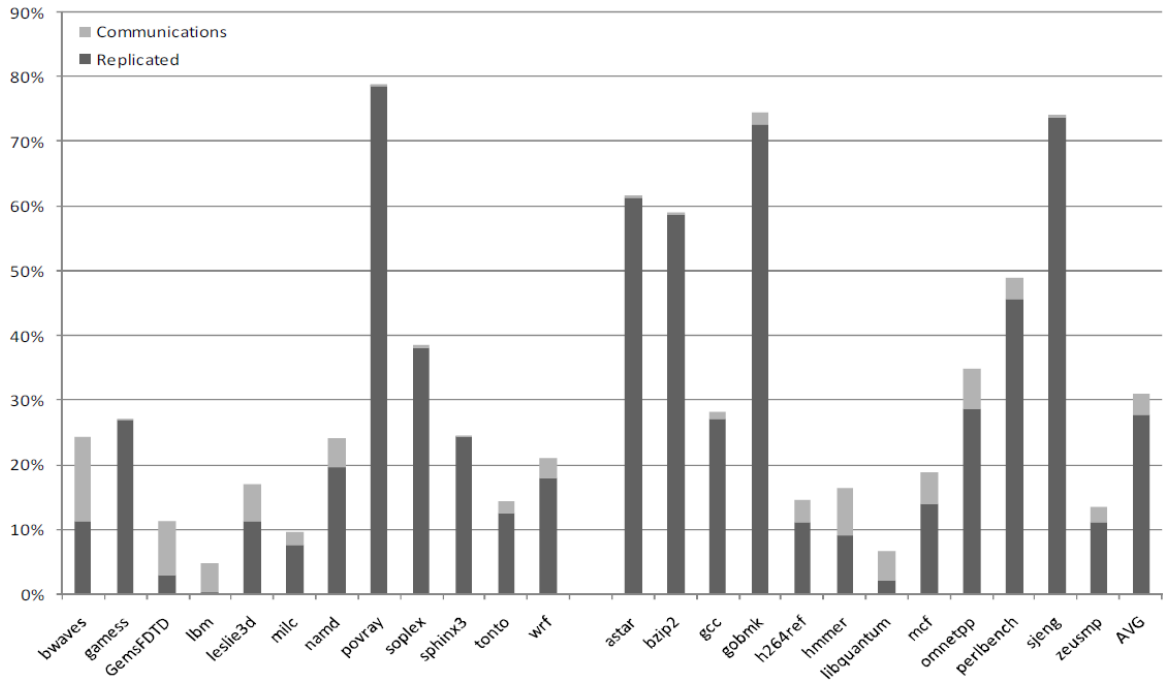


Figure 49. Additional instructions introduced by Anaphase thread decomposition

communications. In this case, the performance speed-up of Anaphase for the studied benchmarks drops from 32% to 20% on average.

3.6.2.3 Hardware Complexity

Finally, we have evaluated the complexity of the Anaphase hardware support in terms of extra area. We have used CACTI 5.3 [93] to measure the area required by our proposal in a contemporary CMP.

For this study we have considered a processor with only one tile based on the parameters shown in Table 6 (*Medium* core), which is similar to an Intel® Core™ 2 Duo processor. On 65 nm technology, the studied base processor has an area of 143 mm^2. We have measured that our proposal, the *ICMC* and the extensions to *L1* and *L2* caches, increases the area up to 153.4 mm^2 (7.2% increase).

3.7 Conclusions

In this chapter we have presented and evaluated the Anaphase architecture, a hybrid software/hardware platform that exploits fine-grain speculative thread decomposition and leverages multiple cores to boost single-thread performance on CMPs. The main novelty of the proposed technique is its fine-grain thread decomposition algorithm, which is able to shred a region of code into multiple threads at instruction granularity. The fine-grain decomposition provides many benefits compared to traditional coarse-grain

SpMT shemes thanks to the flexibility to distribute individual instructions among threads. In that sense, Anaphase is able to effectively exploit ILP, TLP, and MLP, generating threads that better adapt to the available parallelism of the applications, have better workload balance, and fewer inter-thread dependences. In addition, inspired by the Mitosis architecture, it leverages pre-computation to handle inter-thread dependences, which has been shown to be an effective software value prediction mechanism for SpMT in previous Chapter 2.

We have described the whole Anaphase architecture. On the software side, we have presented the Anaphase compiler and its fine-grain thread decomposition algorithm based on a multi-level graph partitioning technique. The two steps of the algorithm, coarsening and refinement, have been described with special emphasis on the different heuristics and the objective function used to generate threads that provide the aforementioned benefits. We have seen that exploiting both compute and memory parallelism, reducing the critical path, and properly selecting how to deal with inter-thread dependences in order to reduce the communication overhead, are key elements for obtaining good partitions.

On the hardware side, the proposed architecture features a cost-effective hardware support for executing Anaphase threads generated at compile time. The key microarchitecture components of the Anaphase multi-core processor have been presented. In particular, we have proposed a novel hardware component, named Inter-Core Memory Coherency Module (*ICMC*), which is in charge of updating the memory state in the original program order, detecting memory violations, and implementing checkpointing and recovery mechanisms. The ICMC is placed in the uncore, it interferes very little with the cores, and allows each core to execute decoupled of the others. In addition, the proposed checkpointing mechanism is able to take frequent checkpoints without incurring to large overheads. This feature has proven to be crucial for keeping the misspeculation penalty small.

Results reported in this thesis strongly validate the effectiveness of Anaphase for boosting single-thread performance on multi-core architectures, resulting from exploiting ILP, TLP and MLP, with a high accuracy, and low overheads. We have shown that, while previous coarse-grain SpMT schemes fail to exploit TLP on hard to parallelize applications, Anaphase is able to extract a larger amount of the parallelism available in these applications. Overall, Anaphase is able to improve single-thread performance by 30% and 37% on average for the Spec2006 suite and up to 2.15x and 2.6x for some selected benchmarks, using medium sized cores and small sized cores respectively.

Finally, we have compared Anaphase to other techniques pursuing the same goal of boosting single-thread performance on multi-cores systems. For this comparison, two techniques have been chosen

due to its high relevance on the literature: (i) hardware-only Core Fusion and (ii) a loop parallelization scheme based on coarse-grain SpMT. We have shown that Anaphase outperforms Core Fusion by more than 10% on average on the Spec2006 suite for all configurations, and coarse-grain SpMT loop parallelization working at iteration level by more than 20%.

These results confirm that Anaphase effectively overcomes the main constraints of previous SpMT schemes and adaptive multi-core systems, and it is a promising architecture for boosting single-thread performance on current and future CMPs. However, as pointed out in this chapter, we have identified a few areas for improvement that need further work. Examples of these areas are: (i) better region selection scheme and methodology in order to increase the coverage of optimized regions, (ii) apply *p-slice* speculative optimizations, like the ones proposed in Mitosis, to reduce pre-computation overheads, and (iii) study the scalability of the Anaphase architecture for more than two cores. We plan to address all these issues as part of the future work presented in the next chapter.

Chapter 4

Conclusions and Open-Research Areas

In this Chapter, the main conclusions of this thesis are presented. Moreover, some open-research areas in the topic of this thesis are pointed out.

4.1 Conclusions

In the era of multi-core architectures or Chip Multi Processors (CMPs), sequential and lightly threaded applications are still prevalent in many computation areas due to the burdens of parallel programming and the limitations of automatic parallelizers. These applications have traditionally relied on architectures which exploit more Instruction Level Parallelism (ILP) from the additional transistor budget ensured by Moore's law. Unfortunately due to the complexity and power constraints designers have shifted to multi-core processors which improve the performance of applications by exploiting Thread Level Parallelism (TLP) instead of ILP. Hence, sequential applications would not benefit from these multi-core designs by the conventional execution paradigms. In some cases the individual cores in a multi-core architecture are even simpler than the traditional single core processors and hence exploit lesser ILP. In such cases, sequential applications may even suffer a performance drop on these architectures. Given this scenario, speeding-up sequential and lightly threaded applications on multi-core processors has become of paramount importance in order to keep enjoying the performance benefits of current and future architectures.

In this thesis, we have proposed and evaluated two novel schemes to speed-up sequential or lightly threaded applications in current and future multi-core processors. These new systems rely on both Speculative Multithreading (SpMT) [3][63][89][96][106] techniques and the concept of adaptive CMP architectures, like Core Fusion [36], to leverage the additional resources of multi-core processors and exploit the ILP and TLP available in these applications. These novel proposals build up on many previous research efforts on these areas during the last decade, which so far showed marginal performance improvements.

In that sense, we have first identified the fundamental challenges of previous techniques that constrained the exploitable parallelism and the overall performance gains and we have proposed in this thesis schemes to effectively address them. In a nutshell, main challenges that we have addressed are: (i) achieve high parallelization coverage on irregular applications, (ii) extract speculative threads from hard to parallelize codes that adapt to the available parallelism, have good workload balance, and few inter-thread dependences, (iii) manage inter-thread dependences with high accuracy and low overhead, and (iv) achieve good performance benefits keeping hardware complexity low.

As mentioned, to address these challenges we have proposed two systems. The first system we have proposed, the Mitosis architecture [30][54], is a hybrid software/hardware SpMT system. Its main distinguishing feature is that it is able to parallelize irregular applications and achieve good performance

even in the presence of frequent dependences among threads. This is achieved thanks to: (i) a general thread partitioning scheme implemented in the Mitosis compiler that is able to place *spawning pairs* [58] in any point of the program and identify the most effective points to spawn speculative threads, and (ii) a powerful software value prediction technique to manage inter-thread dependences. This value prediction technique is based on predicting/computing the thread input values via software through a piece of code, the pre-computation slice (*p-slice*), that is generated at compile time and added at the beginning of each thread. The accuracy of a *p-slice* is higher than other prediction schemes because it is constructed from the original program instructions. In addition, we have proposed several aggressive *p-slice* optimizations to reduce the overhead without sacrificing too much accuracy.

As described in Chapter 2, Mitosis threads execute on a conventional multi-core architecture provided with a cost-effective hardware support that manage the speculative register and memory state, and implement effective validation and recovery mechanisms. Novel schemes to support the execution and validation of *p-slices* have been proposed in this thesis.

Results obtained by the Mitosis architecture with four cores for a subset of the Olden benchmarks [13] show significant performance improvement over single-thread execution. It outperforms the single-threaded execution by 2.2 times and provides more than a 1.75 speed-up over a double-sized out-of-order processor. A similar speed-up is achieved over a processor with perfect memory. These results confirm that there are large amounts of available TLP for code that is resistant to conventional parallelism techniques. However, this parallelism requires highly accurate dependence prediction and efficient data communication between threads, as provided by the Mitosis architecture.

Mitosis, thanks to its novel features, like the management of inter-thread dependences based on *p-slices*, is able to effectively parallelize irregular sequential applications. However, the parallelism that is exploited is constrained by the kind of threads that are extracted. Like any conventional SpMT system, Mitosis has a coarse-grain thread decomposition scheme, where threads are formed by large chunks of consecutive instructions. Finding this kind of threads on hard to parallelize applications may not be always possible or may end up on suboptimal thread partitions. Many hard to parallelize applications require fine-grain decomposition schemes in order to better adapt to the available parallelism.

Our second proposal, the Anaphase architecture [55][56], is a hybrid software/hardware SpMT system that combines the best of pure Speculative Multithreading schemes, like Mitosis, and the concept of fusing cores introduced in adaptive CMP architectures like Core Fusion [36]. In contrast to traditional SpMT schemes, the main distinguishing feature of this novel scheme is that it effectively exploits ILP,

TLP and Memory Level Parallelism (MLP), thanks to its unique and powerful decomposition algorithm that decomposes an application into threads at instruction granularity. The flexibility that this fine-grain decomposition provides has many benefits. As a result, Anaphase is able to generate threads that better adapt to the available parallelism of the applications, have better workload balance, and fewer inter-thread dependences.

Anaphase builds up on the techniques proposed in Mitosis and the observations we extracted from it. In that sense, Anaphase leverages pre-computation to handle inter-thread dependences, which has been shown to be an effective software value prediction mechanism for SpMT in our previous proposal. In addition, the Anaphase architecture implements a cost-effective hardware support on top of a conventional multi-core processor that allows the execution of Anaphase threads with low hardware complexity.

Results reported in this thesis strongly validate the effectiveness of Anaphase for boosting single-thread performance on multi-core architectures, resulting from exploiting ILP, TLP and MLP, with a high accuracy, and low overheads. We have shown that, while previous coarse-grain SpMT schemes fail to exploit TLP on hard to parallelize applications, Anaphase is able to extract a larger amount of the parallelism available in these applications. Overall, Anaphase is able to improve single-thread performance by 41% on average for the Spec2006 suite and up to 2.6x for some selected benchmarks. Compared to other techniques, Anaphase outperforms Core Fusion by more than 10% on average on the Spec2006, and a coarse-grain SpMT loop parallelization system working at iteration level by more than 20%.

These results confirm that Anaphase effectively overcomes the main constraints of previous SpMT schemes and adaptive multi-core systems, and it is a promising architecture for boosting single-thread performance on current and future CMPs.

4.2 Open Research Areas

4.2.1 Anaphase Improvements

In the previous chapter, we identified several areas for improving the Anaphase proposal that require additional work. These areas are:

- *Region selection.* A better region selection scheme is required in order to increase the coverage of the optimized regions. As described in Section 3.4.1, in Anaphase we have focused on outer loops as a first solution to select regions that cover most of the execution time of applications, without

increasing too much the complexity of this step. This is a good compromise for the vast majority of applications but for some applications it may constraint the parallelization coverage. In fact, results presented in the previous chapter confirm that some irregular applications suffer from this problem. For these cases, a more in-depth analysis is required for selecting those regions that have high coverage and provide the best opportunities for parallelization. In addition, as mentioned for the Mitosis architecture, a selection mechanism that is not tied to any high-level code structure (i.e. loops) is also desirable. Given that the Anaphase thread decomposition algorithm is able to decompose any kind of code region, it would be possible to implement a flavor of the region selection scheme proposed in Mitosis (see Section 2.4.1). We believe that the synergy of the Mitosis selection scheme with the flexibility of the Anaphase fine-grain thread decomposition would provide significant performance benefits.

- *P-slice optimizations*. Results presented in the previous chapter, reveal that for some applications Anaphase may introduce too many replicated instructions. This is due to the fact that in the current implementation of Anaphase *p-slices* are not specially optimized. It would be possible to apply all the *p-slice* optimizations present in the Mitosis architecture in order to reduce the replication overhead. However, some of these speculative optimizations require the validation of the pre-computed values. Additional hardware similar to the one proposed in Mitosis for *p-slice* validation would be needed. Alternatively, it may be possible to perform the required checks by software.

- *Scalability*. The software and hardware components of the Anaphase architecture have been designed to support decomposing an application into multiple threads and running them in parallel. Although in the previous chapter we have shown results for only two threads, there is no fundamental reason that prevents us to evaluate the system for more than two threads. This decision was made purely due to constrains on the simulation infrastructure. On the other hand, focusing in two thread decomposition is enough to show the benefits and characteristics of Anaphase and at the same time is an interesting scenario on current multi-core systems. Nevertheless, we believe that it would be interesting to study the scalability of the Anaphase architecture for more than two cores.

4.2.2 Power Reduction

Both the approaches studied in this thesis use aggressive speculation as a means of extracting more parallelism in the programs. One of the biggest concerns regarding speculative techniques in general and

speculative multithreading in particular is about power. Since a wrong speculation often leads to wastage of work leading to wastage of power, as well as some side effects like cache pollution and so on, effective throttling of speculation would be a necessary mechanism in future architectures wanting to use speculation for performance improvements.

Further, the design of the hardware support for running speculative threads on a multi-core architecture has its implications not only on the performance overheads but also on power usage. New complex hardware structures which are needed to support speculation may have high power consumption. We believe that designing these structures with the power as first hand design parameter is going to be an important criterion for any future acceptability of these designs.

Many hardware components proposed in this thesis have been designed with the spirit of keeping hardware complexity low. In that sense, we believe that we are not far from having a cost-effective hardware support for our two proposed architectures. However, we understand that power metrics and better complexity measures should be used in order to evaluate the current proposals and envision possible future enhancements.

4.2.3 Workloads

As it is well known in the research community, architecture designs are driven by the workloads that are used in evaluating them. Speculative Multithreading research has been highly focused on using SPEC benchmarks or other small benchmarks that may not be fully representative of real use scenarios.

Commercial benchmarks, for instance in the area of desktop or multimedia workloads, still have many sequential applications. The research on multi-core architectures that focuses on sequential or lightly threaded applications should study some more representative workloads which might provide newer insights in designing such systems.

List of tables

List of figures

References

[1] A. Agarwal and M. Levy, "*The Kill Rule for Multicore*", in Proc. of the 44th Annual Design Automation Conference, pp. 750-753, 2007

[2] R. Allen and K. Kennedy, "*Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*", Morgan Kaufmann, 2002

[3] H. Akkary and M.A. Driscoll, "*A Dynamic Multithreading Processor*", in Proc. of the 31st Int. Symp. on Microarchitecture, 1998

[4] A. S. Al-Zawawi,, V. K. Reddy, E. Rotenberg, and H. Akkary, "*Transparent control independence (TCI)*", in Proc. of the 34th Int. Symp. on Computer Architecture, 2007

[5] S. Balakrishnan and G. Sohi, "*Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs*", in Proc. of the Int. Symp. on Computer Architecture, pp. 302-313, 2006

[6] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "*Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing*", in Proc. of the 27th Int. Symp. on Computer Architecture, pp. 282-293, 2000

[7] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, "*Parallel Programming with Polaris*", in Proc. of Computer, pp. 78-82, December 1996

[8] B. Bloom, "*Space/Time Trade-Offs in Hash Coding with Allowable Errors*", in the Communications of the ACM, July 1970

[9] S. Breach, T.N. Vijaykumar, and G.S. Sohi, "*The Anatomy of the Register File in a Multiscalar Processor*", in Proc. of the Int. Symp. on Microarchitecture, pp. 181-190, 1994

[10] M. G. Burke and R. K. Cytron, "*Interprocedural Dependence Analysis and Parallelization*", in Proc. of the Intl. Conf. on Programming Language Design and Implementation, pp. 139-154, 1986

[11] R. Canal, J. M. Parcerisa, and A. Gonzalez, "*A Cost-effective Clustered Architecture*", in Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 160–168, 1999

[12] R. Canal, J. M. Parcerisa, and A. Gonzalez, "*Dynamic Cluster Assignment Mechanisms*", in Proc. of the 6th Int. Symp. High-Performance Computer Architecture, 2000

[13] M. C. Carlisle and A. Rogers, "*Software Caching and Computation Migration in Olden*", in Proc. of the 5th Symp. on Principles and Practice of Parallel Programming, p.29-38, 1995

[14] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "*Bulk Disambiguation of Speculative Threads in Multiprocessors*", in Proc. of the 33rd Int. Symp. on Computer Architecture, 2006

[15] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "*Simultaneous Subordinate Microthreading (SSMT)*", in Proc. of the Int. Symp. on Computer Architecture, pp. 186-195, 1999

[16] M. Cintra, J.F. Martinez, and J. Torrellas, "*Architectural Support for Scalable Speculative Parallelization in Shared-Memory Systems*", in Proc. of the 27th Int. Symp. on Computer Architecture, 2000

[17] M. Cintra and J. Torrellas, "*Eliminating Squashes through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors*", in Proc. of the 8th Int. Symp. High-Performance Computer Architecture, 2002

[18] J. D. Collins and D. M. Tullsen, "*Clustered Multithreaded Architectures - Pursuing Both Ipc and Cycle Time*", in Proc. of the Int. Parallel and Distributed Processing Symp., 2004

[19] 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 Proc. of the 28th Int. Symp. on Computer Architecture, 2001

[20] L. Codrescu and D. Wills, "*On Dynamic Speculative Thread Partitioning and the MEM-Slicing Algorithm*", in Proc. of the Int. Conf. Parallel Architectures and Compilation Techniques, pp. 40-46, 1999

[21] C. Colohan, A. Ailamaki, J. Steffan and T. Mowry, "*Tolerating Dependences Between Large Speculative Threads Via Sub-Threads*", in Proc. of the 33rd Int. Symp. on Computer Architecture, 2006

[22] ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf

[23] Z.-H. Du, C-Ch. Lim, X.-F. Li, Q. Zhao, and T.-F. Ngai, "*A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs*", in Proc. of the Int. Conf. on Programming Language Design and Implementation, 2004

[24] P.K. Dubey, K. O'Brien, K.M. O'Brien, and C. Barton, "*Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading*", in Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, 1995

[25] E. Duesterwald, R. Gupta, and M.L. Soffa, "*Rigorous Data Flow Testing through Output Influences*", in 2nd Irvine Software Symposium, 1992

[26] J. Ferrante, K. Ottenstei, and J. Warren, "*The Program Dependence Graph and its Use in Optimization*", ACM Transactions on Programming Languages and Systems (TOPLAS), 9(3), 1987

[27] B. Fields, R. Bodík, and M. D. Hill, "*Slack: Maximizing Performance under Technological Constraints*", in Proc. of the 29th Int. Symp. on Computer Architecture, 2002

[28] M. Franklin and G.S. Sohi, "*The Expandable Split Window Paradigm for Exploiting Fine Grain Parallelism*", in Proc. of the 19th Int. Symp. on Computer Architecture, 1992

[29] M. Franklin and G.S. Sohi, "*ARB: A hardware mechanism for dynamic reordering of memory references*", IEEE Transactions on Computers 45, 5 (May), 552–571, 1996

[30] C. García, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. Tullsen, "*Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices*", in Proc. of the Int. Conf. on Programming Language Design and Implementation, 2005

[31] M. Garzarán, M. Prvulovic, J. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas, "*Tradeoffs in Buffering Speculative Memory State for Thread-level Speculation in Multiprocessors*", in ACM Transactions on Architecture and Code Optimization, Volume 2 , Issue 3, September 2005

[32] S. Gopal, T.N. Vijaykumar, J.E. Smith, and G.S. Sohi, "*Speculative Versioning Cache*", in Proc. of the 4th Int. Symp. on High Performance Computer Architecture, 1998

[33] L. Hammond, M. Willey, and K. Olukotun, "*Data Speculation Support for a Chip Multiprocessor*", in Proc. of the Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 1998

[34] M. Herlihy and J. E. B. Moss, "*Transactional Memory: Architectural Support for Lock-Free Data Structures*", in Proc. of the Int. Symp. on Computer Architecture, pp. 289-300, 1993

[35] R. Hower and M. D. Hill, "*Exploiting Episodes for Lightweight Race Recording*", in Proc. of the Int. Symp. on Computer Architecture, pp. 265-276, 2008

[36] E. Ipek, M. Kirman, N. Kirman, and J.F. Martinez, "*Core Fusion: Accommodating Software Diversity in Chip Multiprocessors*", in Proc. of the Int. Symp. on Computer Architecture, 2007

[37] T. Johnson, R. Eigenmann, and T. Vijaykumar, "*Min-Cut Program Decomposition for Thread-Level Speculation*", in Proc. of the Int. Conf. on Programming Language Design and Implementation, 2004

[38] R. Ju, S. Chan, and C. Wu, "*Open Research Compiler for the Itanium™ Family*", in Tutorial in the 34th Int. Symp. on Microarchitecture, 2001

[39] J. A. Kahle , M. N. Day , H. P. Hofstee , C. R. Johns , T. R. Maeurer , and D. Shippy, "*Introduction to the Cell Multiprocessor*", IBM Journal of Research and Development, v.49 n.4/5, p.589-604, July 2005

[40] G. Karypis and V. Kumar, "*Analysis of Multilevel Graph Partitioning*", in Proc. of the 7th Supercomputing, 1995

[41] B. Kernighan and S. Lin, "*An Efficient Heuristic Procedure for Partitioning of Electrical Circuits*", in Bell System Technical Journal, 1970

[42] P. Kongetira, K. Aingaran, and K. Olukotun, "*Niagara: A 32-Way Multithreaded Sparc Processor*", IEEE Micro 25(2): 21-29, 2005

[43] B. Korel and J. Laski, "*Dynamic Program Slicing*", Information Processing Letters, Volume 29, Issue 3, October 1988

[44] V. Krishnan and J. Torrellas, "*Hardware and Software Support for Speculative Execution of Sequential binaries on a Chip-Multiprocessor*", in Proc. of the Int. Conf. on Supercomputing, pp. 85-92, 1998

[45] V. Krishnan and J. Torrellas, "*A Chip Multiprocessor Architecture with Speculative Multithreading*", in IEEE Trans. Comput. 1999 Special Issue on Multithreaded Architecture

[46] R. Kumar, Keith I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "*Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction*", in Proc. of the 36th Int. Symp. on Microarchitecture, 2003

[47] J. R. Larus and R. Rajwar, "*Transactional Memory*", Morgan & Claypool Publishers, 2006

[48] F. Latorre, J. Gonzalez, and A. Gonzalez, "*Back-end Assignment Schemes for Clustered Multithreaded Processors*", in Proc. of the Int. Conf. on Supercomputing, pp. 316–325, 2004

[49] HQ. Le, et.al., "*IBM POWER6 microarchitecture*", IBM Journal of Research and Development, Volume 51, Issue 6 (November 2007)

[50] X. Li, Z. Du, Q. Zhao, and T. Ngai, "*Software Value Prediction for Speculative Parallel Threaded Computtions*", in Proc. of the First Value-Prediction Workshop, 2003

[51] A. W. Lim and M. S. Lam, "*Maximizing Parallelism and Minimizing Synchronization with Affine Transforms*", in Proc. of Parallel Computing, pp. 201-214, 1997

[52] M. Lipasti and J. P. Shen, "*Exploiting Value Locality to Exceed the Dataflow Limit*", in Proc. of Int. Journal of Parallel Programming, 1998

[53] W. Liu, et.al., "*POSH: a TLS compiler that exploits program structure*", in Proc. of the 11th Int. Symp. on Principles and Practice of Parallel Programming, 2006

[54] C. Madriles, C. García, J. Sánchez, P. Marcuello, A. González, D. Tullsen, H. Wang, and J. P. Shen, "*Mitosis: Speculative Multithreaded Processor based on Pre-Computation Slices*", in IEEE Transactions on Parallel Distributed Systems, 2008

[55] C. Madriles, P. López, J. M. Codina, E. Gibert, F. Latorre, A. Martínez, R. Martínez, and A. González, "*Boosting Single-Thread Performance in Multi-Core Systems Through Fine-Grain Multi-Threading*", in Proc. of the 36th Int. Symp. on Computer Architecture, 2009

[56] C. Madriles, P. López, J. M. Codina, E. Gibert, F. Latorre, A. Martínez, R. Martínez, and A. González, "*Anaphase: A Fine-Grain Thread Decomposition Scheme for Speculative Multithreading*", in Proc. of the 18th Int. Conf. on Parallel Architectures and Compilation Techniques, 2009

[57] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "*Smart Memories: a modular reconfigurable architecture*", in Int. Symp. on Computer Architecture, pp. 161–171, 2000

[58] P. Marcuello and A. González, "*Thread-Spawning Schemes for Speculative Multithreaded Architectures*", in Proc. of the Symp. on High Performance Computer Architectures, 2002

[59] P. Marcuello and A. Gonzalez, "*Exploiting Speculative Thread Level Parallelism on SMT Processors*", in Proc. of the 7th Int. Conf on High Performance Computer and Networking, 1999

[60] P. Marcuello and A. González, "*Clustered Speculative Multithreaded Processors*", in Proc. of the 13th Int. Conf. on Supercomputing, pp. 365-372, 1999

[61] P. Marcuello, A. González, and J. Tubella, "*Speculative Multithreaded Processors*", in Proc. of the 12th Int. Conf. on Supercomputing, 1998

[62] P. Marcuello, J. Tubella, and A. González, "*Value Prediction for Speculative Multithreaded Architectures*", in Proc. of the 32nd Int. Symp. on Microarchitecture, 1999

[63] P. Marcuello, "*Speculative Multithreaded Processors*," PhD dissertation, Universitat Politecnica de Catalunya, 2003

[64] T. Marr et al., "*Hyperthreading Technology Architecture and Microarchitecture*", Intel Technology Journal, vol. 6, no. 1, 2002

[65] J.F. Martinez, J. Renau, M.C. Huang, M. Prvulovic, and J. Torrellas, "*Cherry: Checkpointed Early Recycling in Out-of-order Microprocessors*", in Proc. of the Int. Symp. on Microarchitecture, 2002

[66] M. Mehrara, J. Hao, P-C Hsu, and S. A. Mahlke, "*Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory*", in Proc. of the Int. Conf. on Programming Language Design and Implementation, 2009

[67] A. Mendelson, J, Mandelblat, S. Gochman, A. Shemer, R. Chabukswar, E. Niemeyer, and A. Kumar, "*CMP Implementation in Systems Based on the Intel® Core™ Duo Processor*", in Intel Technology Journal, Volume 10, Issue 2, 2006

[68] P. Montesinos, L. Ceze, and J. Torrellas. "*DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently*", in Proc. of the Int. Symp. on Computer Architecture, pp. 289-300, 2008

[69] C. E. Oancea, A. Mycroft, and T. Harris, "*A Lightweight In-Place Implementation for Software Thread-Level Speculation*", SPAA '09. pp. 1–10, 2009

[70] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita, "*Pinot: Speculative Muti-threading Processor Architecture Exploiting Parallelism over a wide Range of Granularities*", in Proc. of the 38th Int. Symp. on Microarchitecture, 2005

[71] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang, "*The Case for a Single-Chip Multiprocessor*", in Proc. of the 7th Int. Symp. on Architectural Support for Programming Languages and Operating Systems, 1996

[72] J. Oplinger, D. Heine, and M. Lam, "*In Search of Speculative Thread-Level Parallelism*", in Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, 1999

[73] G. Ottoni and D. August, "*Communication Optimizations for Global Multi-threaded Instruction Scheduling*", in Proc. of the 13th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 2008

[74] V. Packirisamy, Y. Luo, W-L Hung, A. Zhai, P-C Yew, and T-F Ngai, "*Efficiency of Thread-Level Speculation in SMT and CMP Architectures - Performance, Power and Thermal Perspective*", in Proc. of the 26th Int. Conf. on Computer Design, 2008

[75] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "*Pinpointing Representative Portions of Large Intel® Itanium® Programs with Dynamic Instrumentation*", in Proc. of the 37th Int. Symp. on Microarchitecture, 2004

[76] M. S. Papamarcos and J. H. Patel, "*A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories*", in Proc. of the 11th Int. Symp. on Computer Architecture, pp.348-354, 1984

[77] I. Park, B. Falsafi, and T.N. Vijaykumar, "*Implicitly-Multithreaded Processors*", in Proc. of the 30th Int. Symp. on Computer Architecture, pp. 39-51, 2003

[78] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas, "*Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization*", in Proc. of the 28th Int. Symp. on Computer Architecture, 2001

[79] R. Rajwar and J. R. Goodman, "*Speculative lock elision: enabling highly concurrent multithreaded execution*", in Proc. of the 34th Int. Symp. on Microarchitecture, 2001

[80] R. Ranjan, P. Marcuello, F. Latorre and A. González, "*P-Slice Based Efficient Speculative Multithreading*", in Proc. of the 16th Int. Conf. on High Performance Computing, 2009

[81] R. Ranjan, P. Marcuello, F. Latorre and A. González, "*Fg-STP: Fine-Grain Single Thread Partitioning on Multicores*", in Proc. of the 17th Int. Symp. on High Performance Computer Architecture, 2011

[82] L. Rauchwerger and D. A. Padua, "*The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization*", in IEEE Transactions of Parallel Distributed Systems, 10(2): 160-180, 1999

[83] T. Reps and T. Bricker, "*Illustrating Interference in Interfering Versions of Programs*", ACM SIGSOFT Software Engineering Notes, Volume 14, Issue 7, November 1989

[84] A. Roth and G.S. Sohi, "*Speculative Data-Driven Multithreading*", in Proc. of the 7th Int. Symp. on High-Performance Computer Architecture, pp. 37-48, 2001

[85] P. Rundberg and P. Stenström, "*An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors*", The Journal of Instruction-Level Parallelism, 1999

[86] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "*Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture*", in Proc. of the Int. Symp. on Computer Architecture, pp. 422–433, 2003

[87] S. Sarangi, W. Liu, J. Torrellas, and Y. Zhou, "*ReSlice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing*", in Proc. of the 38th Int. Symp. on Microarchitecture, 2005

[88] Y. Sazeides and J. E. Smith, "*The Predictability of Data Values*", in Proc. of the 30th Int. Symp. on Microarchitecture, 1997

[89] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "*Multiscalar Processors*", in Proc. of the 22nd Int. Symp. on Computer Architecture, pp.414-425, 1995

[90] J. Steffan and T. Mowry, "*The Potential of Using Thread-level Data Speculation to Facilitate Automatic Parallelization*", in Proc. of the 4th Int. Symp. on High Performance Computer Architecture, pp. 2-13, 1998

[91] J. Steffan, C. Colohan, A. Zhai, and T. Mowry, "*Improving Value Communication for Thread-Level Speculation*", in Proc. of the 8th Int. Symp. on High Performance Computer Architecture, pp. 58-62, 1998

[92] J. Steffan, C. Colohan, A. Zhai, and T. Mowry, "*A Scalable Aproach to Thread-Level Speculation*", in Proc. of the Int. Symp. on Computer Architecture, 2000

[93] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. P. Jouppi, "*CACTI 5.1*", Technical Report HPL-2008-20, HP Labs

[94] C. Tian, M. Feng, V. Nagarajan, and R. Gupta, "*Copy or Discard Execution Model for Speculative Parallelization on Multicores*", in Proc. of the 41st Int. Symp. on Microarchitecture, 2008

[95] G. Tournavitis, Z. Wang, B. Franke and M. F. P. O'Boyle, "*Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping*", in Proc. of the Int. Conf. on Programming Language Design and Implementation, pp. 177-187, 2009

[96] J.Y. Tsai and P-C. Yew, "*The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation*", in Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, 1995

[97] D. M. Tullsen, S.J. Eggers, and H.M. Levy, "*Simultaneous Multithreading: Maximizing On-Chip Parallelism*", in Proc. of the 22nd Int. Symp. on Computer Architecture, pp. 392-403, 1995

[98] N. Vachharajani, R. Rangan, E. Raman, M. Bridges, G. Ottoni, and D. August, "*Speculative Decoupled Software Pipelining*", in Proc. of the Int. Conf. on Parallel Architecture and Compilation Techniques, pp. 49-59, 2007

[99] S. Vangal., et al., "*An 80-Tile 1.28TFLOPS Network-on-Chip in 65 nm CMOS*", in Proc. of the Int. Solid-State Circuits Conf., 2007

[100] T.N. Vijaykumar, "*Compiling for the Multiscalar Architecture*", Ph.D. Dissertation, Univ. of Wisconsin-Madison, 1998

[101] F. Warg and P. Stenström, "*Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms*", in Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, 2001

[102] M. Weiser, "*Program slicing*", in Proc. of the Int. Conf. on Software Engineering, 1981

[103] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, "*Compiler Optimization of Scalar Value Communication Between Speculative Threads*", in Proc. of the Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 171-183, 2002

[104] Y. Zhang, L. Rauchwerger, and J. Torrellas, "*Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors*", in Proc. of the 5th Int. Symp. on High Performance Computer Architecture, 1999

[105] C.B. Zilles and G.S. Sohi, "*Execution-Based Prediction Using Speculative Slices*", in Proc. of the 28th Int. Symp. on Computer Architecture, 2001

[106] C.B. Zilles and G.S. Sohi, "*Master/Slave Speculative Parallelization*", in Proc. of the 35th Int. Symp. on Microarchitecture, 2002

[107] H. Zhong, S. A. Lieberman, and S. A. Mahlke, "*Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications*", in Proc. of the Int. Symp. on High-Performance Computer Architecture, 2007

[108] H. Zhong, M. Mehrara, S. A. Lieberman, and S. A. Mahlke, "*Uncovering Hidden Loop Level Parallelism in Sequential Applications*", In Proc. of the 14th Int. Symp. on High Performance Computer Architecture, 2008