EFFICIENT HARDWARE/SOFTWARE CO-DESIGNED SCHEMES FOR LOW-POWER PROCESSORS

Dissertation submitted for the degree of

Doctor of Philosophy in Computer Architecture

PhD Student: Pedro López Muñoz

PhD Supervisor: Fernando Latorre Salinas

PhD Co-supervisors: Enric Gibert Codina, Antonio González Colás

Universitat Politècnica de Catalunya

Programa de Doctorat en Arquitectura de Computadors

Barcelona, 2014

Contents

List of Figures

List of Tables

Abstract

Nowadays, we are reaching a point where further improving single thread performance can only be done at the expenses of significantly increasing power consumption. Thus, multi-core chips have been adopted by the industry and the scientific community as a proven solution to improve performance with limited power consumption. However, the number of units to be integrated into a single die is limited by its area and power restrictions, and therefore the thread level parallelism (TLP) that could be exploited is also limited. One way to continue incrementing the number of core units is to reduce the complexity of each individual core at the cost of sacrificing instruction level parallelism (ILP). We face a design trade-off here: to dedicate the total available die area to put a lot of simple cores and favor TLP or to dedicate it to put fewer cores and favor ILP. Among the different solutions already studied in the literature to deal with this challenge, we selected hybrid hardware/software co-designed processors. The objective of this solution is to provide high single thread performance on simple low-power cores through a software dynamic binary optimizer (a.k.a. software layer) tightly coupled with the hardware underneath. For this reason, we believe that hardware/software co-designed processors is an area that deserves special attention on the design of multi-core systems since it allows implementing multiple simple cores suitable to maximize TLP but sustaining better ILP than conventional pure hardware approaches. In particular, this thesis explores three different techniques to address some of the most relevant challenges on the design of a simple low-power hardware/software co-designed processor.

The first technique is a profiling mechanism, named as LIU Profiler, able to detect hot code regions. It consists in a small hardware table that uses a novel replacement policy aimed at detecting hot code. Such simple hardware structure implements this mechanism and allows the software to apply heuristics when building code regions and applying optimizations. The LIU Profiler achieves 85.5% code coverage detection whereas similar profilers implementing traditional replacement require a 4x bigger table to achieve similar numbers. Moreover, the LIU Profiler only increases by 1% the total area of a simple low-power processor and consumes less than 0.87% of the total processor power. The LIU Profiler enables improving single thread performance without significantly incrementing the area and power of the processor.

The second technique is a checkpointing scheme aimed to support code reordering and aggressive speculative optimizations on hot code regions. It is named HRC and combines software and hardware mechanisms to checkpoint and to recover the architectural register state of the processor. When compared with pure hardware solutions that require doubling the number of registers, the proposal reduces by 11% the area of the processor and by 24.4% the register file power consumption, at the cost of only degrading 1% the performance.

The third technique is a loop parallelization (called LP) scheme that uses the software layer to dynamically detect loops of instructions and to prepare them to execute multiple iterations in parallel by using Simultaneous Multi-Threading threads. These are optimized by employing dedicated loop parallelization binary optimizations to speed-up loop execution. LP scheme uses novel fine-grain register communication and thread dynamic register binding technique, as well as already existing processor resources. It introduces small overheads to the system and even small loops and loops that iterate just a few times are able to get significant performance improvements. The execution time of the loops is improved by more than a 16.5% when compared to a fully optimized baseline. LP contributes positively to the integration of a high number of simple cores in the same die and it allows those cores to cooperate to some extent to continue exploiting ILP when necessary.

Acknowledgments

Desarrollar y escribir una tesis es una tarea laboriosa que requiere años de dedicación y esfuerzo. En este largo periodo de tiempo son multitud las vivencias y anécdotas acontecidas. En mi caso, y por fortuna, la mayor parte de ellas las he vivido en compañía. El Dr. William H. Stone, científico genetista, dijo en una entrevista en La Vanguardia que lo mejor de él mismo son las personas que le rodean. Durante el desarrollo de esta tesis estas palabras me han sobrevenido continuamente por diversos motivos y no puedo más que hacer acopio de ellas porque he estado rodeado en todo momento de magníficas personas. Me gustaría agradecerles a todos ellos su apoyo, comprensión y ayuda para sacar adelante este trabajo. Espero haberles transmitido mi gratitud fuera de estas palabras que ahora escribo, así como también espero que pueda seguir haciéndolo después de haber cerrado esta importante etapa de mi vida.

Me gustaría comenzar agradeciendo a Antonio González la confianza depositada en mí, desde el día que me contrató para trabajar en Intel así como cuando me ofreció la posibilidad de realizar esta tesis. Me siento muy afortunado de haber podido disfrutar de sus amplios conocimientos en arquitectura y de haber podido aprender de él los principios básicos de la investigación. Además, también me gustaría agradecerle su comprensión, su interés y su paciencia cuando por motivos personales he necesitado priorizar mi vida personal por encima de la profesional.

Similares palabras puedo dedicar a mis otros dos directores de tesis, Fernando y Enric, con los que además desde hace muchos años comparto una bonita amistad. A ambos les admiro profundamente y les considero verdaderos ejemplos de lo que debe ser un gran investigador. Me han ayudado mucho guiándome en las investigaciones, así como en la escritura de los artículos y en la preparación del documento final de la tesis. Durante este tiempo han sabido enseñarme y transmitirme sus conocimientos para ir mejorando poco a poco como investigador. Me siento muy afortunado de haber podido contar con ellos para realizar esta tesis y les agradezco mucho su seriedad y sacrificio para tirarla adelante.

No puedo olvidarme de los integrantes del primer grupo de investigación en el que comencé a trabajar en Intel ya que con ellos los primeros trabajos de esta tesis comenzaron a cobrar vida. Especialmente me gustaría agradecerle a Josep María su ayuda para sacar adelante el capítulo 4, ya que sin él habría sido imposible acabarlo. A Pepe por ejercer de maestro y enseñarme a diseñar un simulador temporal desde cero, que a la

postre ha sido el eje de gran parte de las simulaciones requeridas en mis experimentos. Y por último, pero no menos importante, a Alex Piñeiro por guiarme mientras daba mis primeros pasos por la investigación y por compartir conmigo largas charlas sobre arquitectura y temas varios que me ayudaron mucho a formarme dentro de la gran familia de Intel.

Sin salirme de Intel y aunque no trabajáramos en proyectos comunes, también me gustaría agradecer a Grigoris, Pedro Marcuello y especialmente a Jesús por sus contribuciones en el desarrollo del capítulo sobre el Hot Code Profiler. Me siento muy honrado de haber podido disponer de vuestro tiempo y de vuestros enormes conocimientos para realizar este trabajo.

También me gustaría agradecerle al resto de compañeros de Intel el que hayan soportado mis disquisiciones sobre el desarrollo de la tesis y mis peleas personales internas cuando los ánimos flaqueaban y la consecución de acabar la tesis se convertía en un objetivo demasiado complicado y lejano.

Fuera de laboratorio, me gustaría darles las gracias a mis amigos del CA ComeKM Molins por motivarme a correr y ayudarme a evadirme de la dura tarea del trabajo y del desarrollo de la tesis. Sin ellos los caminos siempre son más duros y han sido las liebres perfectas para allanarme las complicaciones. Aprovecho para pedirles perdón por mis chistes malos, pero no les aseguro que si algún día llego a ser doctor deje de hacerlos.

Me gustaría hacer una mención muy especial a toda mi familia por el infinito apoyo que he recibido de ellos durante este tiempo. Nada de lo que he realizado durante este periodo habría sido posible sin su ayuda. Ellos son los principales responsables de que yo haya tenido fuerzas y ánimos para continuar luchando en mi desarrollo personal y por lo tanto de que haya encontrado tiempo para finalizar esta tesis.

Me gustaría dedicar esta tesis a mis abuelos, Pedro y María. A pesar de lo dura que ha sido su vida, jamás se han rendido en luchar por salir adelante y tirar del carro ayudando a toda la familia. El abuelo siempre nos ha transmitido que el saber no ocupa lugar y la abuela que vivir la vida es todo un privilegio que hay que disfrutar se tenga la edad que se tenga. Ambos, son mis referentes y el mejor espejo en el que mirarme. No es de extrañar que uno de los mayores placeres que he experimentado en estos años de tesis haya sido el sentarme con ellos para hablar y sentirme arropado por su inmenso cariño. En estas charlas, son muchas las veces que me han preguntado por mis estudios y otras tantas las

que me han insistido en que no los abandone nunca. Muchas gracias abuelos por quererme tanto y por estar a mi lado.

También quería hacer una mención especial para mi padre, al que siempre he considerado mi primer maestro. Desde bien pequeño supo transmitirme su pasión por la electrónica y la informática haciéndome partícipe de sus experimentos. No es de extrañar que quisiera seguir sus pasos y que decidiera convertirme en investigador. Esta tesis no tendría sentido sin él, por su constante apoyo, su motivación y por creer en mi más de lo que yo mismo hago. Sé que él habría sido capaz de hacer unas cuantas tesis si hubiera tenido las oportunidades que él me ha procurado. Así que no puedo más que agradecerle que me haya facilitado tanto las cosas.

En esta lista no puede faltar mi madre. Son tantos los sacrificios que ha realizado por mí que no encuentro palabras para expresar mi gratitud como ella se merece. Siempre me ha facilitado las cosas para realizar mis estudios. Ha sido la primera en ponerme los pies en el suelo, en controlar mis nervios y en darme fuerzas para afrontar todos los problemas que surgían. Sin duda, de ella he aprendido las mejores cosas de esta vida y me ha ayudado a ser la persona que soy. Una parte muy importante de esta tesis le pertenece por derecho propio.

Por último, pero no por ello menos importante, me gustaría mencionar muy especialmente a Helena que me haya acompañado en todo momento en estos últimos años. Ambos decidimos comenzar nuestras tesis por la misma época y ambos pretendemos acabarlas en fechas cercanas. Así que aparte del día a día en pareja, durante estos años también hemos compartido multitud de situaciones consecuencia del desarrollo de la tesis. Me gustaría agradecerle principalmente dos cosas. La primera, que además tiene un impacto directo en el presente documento, son sus múltiples aportaciones derivadas de sus consejos ayudándome a diseñar y corregir mis escritos. La segunda, cuyo impacto es más indirecto, es agradecerle su vitalidad y su coraje para afrontar los retos que ha tenido que superar. Se ha convertido en un ejemplo para mí y me consta que para muchas personas de nuestro entorno. Luchar por merecerme su cariño es la fuerza que me ha guiado en todas las situaciones complicadas con las que he tenido que lidiar en estos últimos años e imaginar un mejor futuro, como fruto de nuestro esfuerzo, ha sido el mejor sueño por el que pelear.

Chapter 1

Thesis Introduction

In the last years the industry and the scientific community have adopted the multi-core chips as a proven solution to obtain significant performance improvements [1] [2] [3] [4] [5] [6] [7]. The integration of multiple core units into a single chip allows executing multiple programs in parallel and also executing multiple threads belonging to the same application simultaneously, which highly improves the thread level parallelism (TLP) that the full processor can exploit. However, multi-core chips present area and power restrictions [8] that can be solved by reducing the complexity of each individual core. This increments the number of units per chip with the aim of exploiting the TLP of the applications. However, reducing the complexity of each unit has a negative impact on the execution of single-threaded applications since less instruction level parallelism (ILP) can be exploited.

Sequential code applications, where instruction level parallelism is still important, are widely used in nowadays processors. In addition, parallel applications contain significant sections of sequential codes. Therefore, exploiting ILP and TLP together is one of the biggest challenges for current and future processor designs. Among the different proposals existing in the literature to deal with this [9] [10] [11] [12], we believe that the most promising are those that rely on hybrid hardware/software co-designed architectures. In these approaches, a software component, known as the software layer, adapts the code of the running applications to be executed more efficiently by taking advantage from the particularities of the hardware. The software layer is implemented very tightly with the hardware and it has full access to its resources. Moreover, the software layer may implement some functionalities of the hardware, which allows reducing the complexity of the processor without negatively impacting its performance (for both single-threaded and multi-threaded applications) [13] [14] [15].

However, most of the hardware/software co-designed proposals require additional hardware resources (for instance by duplicating the register file [15] [16]) and/or introduce significant software overheads (for instance for detecting the code regions to be optimized [17] [18]), which in turns complicates the adoption of processors based on this technology to be implemented in multi-core chips.

In this thesis we target some of the most important design decisions that need to be considered when designing a hardware/software co-designed processor. We propose solutions that simplify the hardware requirements of these processors and also we propose solutions for significantly increase their performance. Thus, the proposals described in this thesis further exploit TLP since the complexity reduction of each core allows implementing a larger number of them in a multi-core system. Moreover, they will continue delivering competitive single-thread performance when TLP is scarce.

1.1 Thesis Objectives

The main goal of this thesis is to offer solutions in order to develop a small (from an area point of view) and low power hardware/software co-designed processor capable of offering a performance close to the one obtained by more complex processors (i.e. out of order cores [19] [20]) but requiring less hardware complexity. This goal would allow incrementing the number of core units per die and, therefore, it would be possible to continue exploiting application thread level parallelism without sacrificing instruction level parallelism.

However, designing a hardware/software co-designed processor is a complex task that requires the interaction of several groups of engineers with different areas of expertise, including deep knowledge about the hardware and the software components. The design of the software layer, the communication interfaces between the hardware and the software, and their interactions, are additional tasks to be considered on top of the already complex task of designing a processor completely in hardware. However, such processors are very suitable for multi-core designs because the introduction of the software layer in the system allows reducing the complexity of the hardware components by moving some of the processor functionalities to the software without impacting the overall performance of the system [15] [21] [16].

In Figure 1.1, we show an example of a few hardware functionalities that can be transferred to the software layer. In the left part of the figure we show a high-block diagram of a traditional hardware processor. The different components that form the

processor are highlighted with different colors. In the right part of the figure we show some of the functions that may form part of the software layer. In this particular example, we highlight that part of the functionalities of the instruction scheduler and the register renaming of the traditional hardware processor can be transferred to the software layer in the form of code functions. The software layer can take care of these functionalities and it is responsible for communicating with the hardware to ensure correct application execution. By moving functionalities from the hardware to the software layer, it is possible to reduce processor area and power consumption at the cost of executing additional instructions.

The software layer can also be employed to improve the performance of the processor without incrementing its hardware complexity [21] [22]. Additional features can be implemented within the software layer to make a more efficient usage of the already existing hardware resources of the processor [23]. For instance, the most common way to improve performance is through dynamically optimizing the code of the application by employing the software layer optimizer. Note that the software layer has full access to the processor hardware resources because it is implemented very tightly with it. In fact, both the software layer and the hardware are seen as a single unit for the entire software stack [13]. However, the software layer introduces additional overheads to the processor that may compromise the benefits obtained from the optimization of the code. The detection of the code suitable to be optimized, its optimization, and the mechanisms to ensure its correct execution introduce additional overheads either by requiring the execution of additional instructions or by requiring specialized hardware resources.

Software layer overheads should be kept as low as possible to maximize the efficiency of hardware/software co-designed processors over more traditional approaches. The techniques presented in this thesis are aimed at reducing these overheads and, consequently, to improve the overall implementation of the hardware/software co-designed processors.

In addition, in this thesis, in order to fulfill the expected simplicity and efficiency, the processor design starting point is to use a simple in-order processor with very constrained area and power requirements. On top of this simple design and taking advantage from the software layer, extra functionalities are implemented to improve its performance without significantly incrementing its hardware complexity.

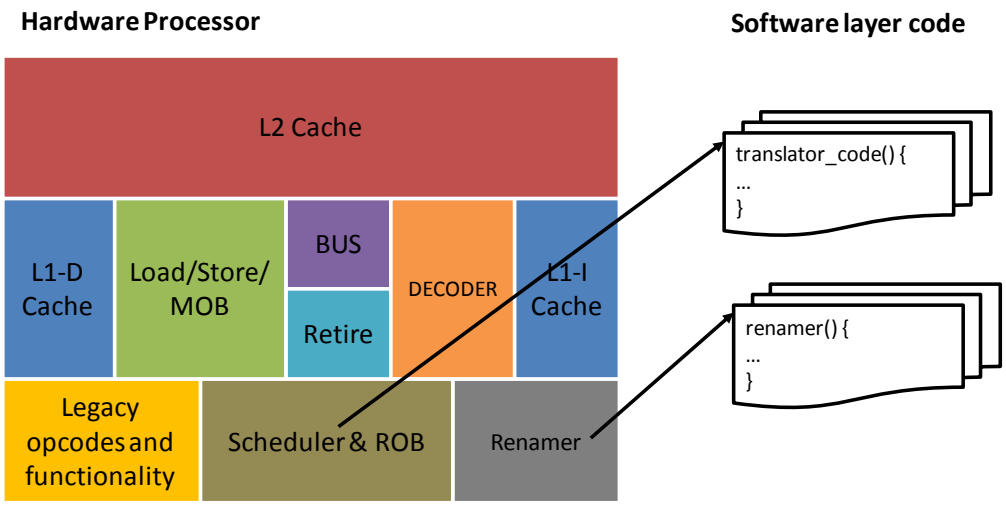


Figure 1.1: Example of possible hardware components to be implemented in software in a Co-designed Hardware/Software processor.

Given that a full processor re-design is too ambitious to be covered in this thesis, the goal of this work is focused on reducing overheads and improving performance by targeting three key points of the main operational flow of hardware/software co-designed processors. These are the mechanism for detecting code regions suitable to be optimized, the mechanisms to guarantee correct region execution, and the optimizer itself. This has resulted in three original contributions: (1) a novel hot code profiler able to detect the most promising pieces of code to be optimized, (2) a novel software register checkpointing technique to support precise interrupts and aggressive optimizations with minimum hardware and software overheads, and (3) a loop parallelization scheme able to significantly improve the ILP of the processor. The two first schemes are aimed at reducing current overheads of hardware/software co-designed processors, whereas the third technique is aimed at improving performance without requiring significant hardware changes. Apart from the main objective of the thesis, each one of these solutions has its own objectives as described in the next sections.

1.1.1 Code Profiler Design Objectives

The code profiler component of a hardware/software co-designed processor is in charge of detecting the most appealing regions of code to be optimized. These regions are the most frequently executed and in turn, the ones that offer higher performance gains when optimized by the software layer (a.k.a. Dynamic Binary Optimizer or DBO for short) [24]. Therefore, the profiler has a big impact on the performance that could be achieved in a hardware/software co-designed processor.

An accurate and efficient code profiler should be capable of detecting the most frequently executed regions, discarding the non-promising regions, and providing up-to-date information about application execution. It should introduce negligible overheads (if any) to the system and should not interfere with the execution of the applications. The potential overheads introduced by the profiler should be widely compensated by the execution of the optimized regions. Moreover, the design of the profiler should not compromise the processor area and power constraints and neither the software design requirements.

The aforementioned objectives for the profiler should guide its design to use less hardware and software resources while keeping similar or better accuracy at detecting the most appealing code regions to be optimized when compared with other proposals from the literature [25] [26] [17] [27] [18].

The design of the code profiler as the detector of the hot regions for the dynamic binary optimizer is a good example of a technique oriented to simplify the hardware requirements of current hardware/software co-designed processors while keeping competitive performance when compared with more complex processors.

1.1.2 Register CheckPointing Design Objectives

In order to support precise exceptions and aggressive code optimizations, the hardware/software co-designed processor requires executing regions atomically [28]. The processor architectural state may not be fully correct during the execution of an optimized region as long as it matches the original architectural state at the end of the region. A common solution to correctly update the architectural state is to checkpoint the state of the memory and the registers before the execution of the region. In case that during the execution of the region there is an exception (including those that are consequence of aggressive and/or speculative optimizations) the state of the processor can be recovered from the saved values and the execution of the region can be restarted using a less or non-optimized version of the code. These saved values for the memory and the registers are what it is commonly known as an architectural checkpoint. In this thesis, we are only focused on register checkpointing.

Proposals from the literature tend to duplicate processor hardware resources [16] and/or to implement costly software algorithms that introduce significant overheads to the system [29] [30]. An efficient hardware/software register checkpointing scheme should minimize the hardware requirements of the processor and should not introduce important

software overheads for saving and restoring the register values. Moreover, the solution should not penalize the common case that is to correctly execute the regions without problems (otherwise optimizing the code regions would not make sense). Finally, supporting speculative aggressive optimizations is a must to further exploit ILP in the processor and therefore, the checkpointing scheme should not penalize miss speculations in order to maximize the performance that the processor is able to achieve.

1.1.3 Loop Parallelization Scheme Design Objectives

Simple in-order processors, like the one selected as baseline for this thesis, are not able to exploit instruction level parallelism. The objective of the loop parallelization scheme is to improve the execution of loop code regions by increasing the instruction level parallelism that could be exploited when executing them in such simple in-order processors. Note that loops are the most predominant parts of the applications and that performance may significantly increase by optimizing them.

In order to achieve this objective, loop parallelization scheme distributes the loop iterations into multiple threads that can be executed simultaneously in the processor[1]. In fact, although the execution of the instructions within each thread is still in order, the instructions belonging to different threads are executed out of order.

Since instructions are executed in an out of order manner, the proposed scheme keeps track of the dependences among them in order to guarantee the correct execution of the codes. Note that contrarily to out of order cores where this functionality is already implemented, simple in-order cores do not require it. Therefore, additional mechanisms to deal with such dependences need to be implemented.

The proposal should avoid the usage of complex hardware solutions in order to keep the design as simple and efficient as possible and it should rely on the software layer to prepare and optimize the codes to be parallelized, removing these responsibilities from the hardware.

The loop parallelization scheme is a good example of a hardware/software co-designed technique oriented to improve the performance of applications without increasing significantly the area and the dissipated power of the processor.

[1] The simple in-order processor considered as baseline for this thesis is based on an Intel® Atom® processor that supports Simultaneously Multi-Threading (SMT) execution [111].

1.2 Outline

This document is organized into 6 chapters. Chapter 1 contains the introduction to this thesis, with the motivation of the work, its main objectives, and the description of the outline of the document, in which these words are found. Chapter 2 describes the basic concepts necessary for proper understanding the rest of the document and the hardware/software co-designed scheme that has been used for developing the techniques proposed in this thesis.

The main contributions of this thesis are described in Chapter 3, Chapter 4, and Chapter 5. In particular, Chapter 3 describes a novel efficient hardware/software profiler solution for fast and accurate code detection. Chapter 4 presents a very efficient hardware/software register checkpointing scheme designed to support aggressive code optimizations in the co-designed system, and Chapter 5 describes a novel technique designed for improving the performance of loop regions even when it is implemented on top of a very simple processor.

Finally, in Chapter 6, we present the main conclusions of the thesis, including the contributions and some recommendations for future work.

Chapter 2

Background

The aim of this chapter is to explain the basic concepts that are necessary for a proper understanding of the rest of the chapters in this thesis. Basically, this chapter describes the basic concepts about hardware/software co-designed virtual machines and presents the hardware/software co-designed approach that has been employed to develop the techniques proposed in this thesis

2.1 General Introduction to Virtual Machines

Modern computers are built around three main components: the hardware, the operating system (OS), and the applications [31]. Computers require the interaction of these components in order to operate. These interactions are done through established interfaces, which are the most effective way to build complete systems made of independent components. Figure 2.1 shows a common scenario describing how the three main components interact in a computer. As it is described, the applications and the OS, which together are also known as the software, communicate with the hardware component. Applications may communicate with the hardware directly or indirectly through the OS whereas the OS always communicates directly. Although the case study presented in Figure 2.1 corresponds to the more common computer design, other design scenarios are also possible depending on how the three components are combined and how they communicate though the defined interfaces.

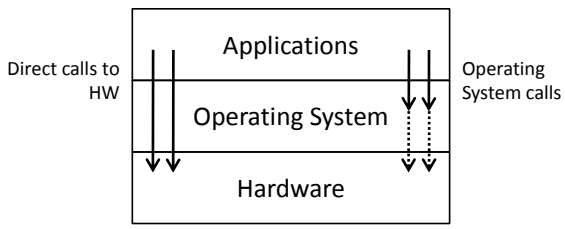


Figure 2.1: Today's computer components.

The main advantage of this design is that each component can be designed independently of the others. For this reason well defined communication interfaces among them become crucial. Actually, the fast evolution of both hardware and software (including applications and OS) compromises the evolution of this design mostly incurring in the following three main problems. First, the three components only work correctly when they interact in a very strict manner. For instance, software applications compiled for a particular set of instructions do not work on hardware that does not support them. Second, the need for backward compatibility with previous designs forces components to support legacy constraints that complicate their implementation. In fact, many current processors maintain obsolete instructions in order to guarantee backward compatibility with older applications [32]. For instance, the Intel® x86 64 bit processors can execute legacy codes written for 32 bit processors by using special executions modes [32]. Third, although the components are independent by definition, their design is limited by the interactions among them. Therefore, components are usually designed in a constrained manner, having into account the other components, in order to facilitate the interaction among them. For instance, most of the traditional OS have been designed to take maximum advantage of particular processors. A real example is the Windows OS that it was only supporting Intel® x86 Instruction Set Architecture until recently [33].

Virtualization is a technique proposed to overcome the design problems previously described. Virtualization is the simulation of the software and/or the hardware component upon which other component runs [34]. The simulated component is usually called virtual machine [13]. Virtualization is often achieved by introducing a new software component within the three component computer design presented previously that connects directly or indirectly the other components [13]. This software layer is able to adapt the components in order to work together by filling or emulating the differences among them [13]. Moreover, it is able to offer compatibility among them guarantying their independent development if needed [30] [35]. Finally, it is also able to define the required interfaces in a very efficient manner getting maximum performance from the interaction of the components [13].

Besides of the aforementioned benefits of introducing the software layer in the three component design, virtual machines are also employed for improving general system performance [36] [37] and simplifying the hardware resources [14] [16] among others.

The software layer acts like a barrier in the design, separating the native components of the system from the ones that communicate through the software layer with the native

components. The term used when referring to the native components is "host" component, whereas the term used to refer to a component that communicates through the software layer is "guest". For instance, an operating system employed directly on top of a specific hardware is called "host OS", whereas one that is employed on top of a virtual machine is called "guest OS".

Virtualization by means of a software layer can be done at two different levels within the three components design described in Figure 2.2. On one hand, virtualization can be applied at the Instruction Set Architecture (ISA) level, which is the communication interface between the applications and the OS with the hardware. This interface includes privileged and non-privileged instructions, the memory interface and its programming model, and the interactions with devices. On the other hand, it can be applied at the Application Binary Interface (ABI) level, which communicates the applications with the OS and the hardware. In this case, applications are built according to a specific OS and hardware.

Smith and Nair proposed a classification of the virtual machines (VMs) [31] based on where the software layer is introduced within the three component computer design. If the software layer is introduced at the "ABI interface" level then the virtual machines are categorized as *Process VMs*. By contrast, if the software layer is introduced at the "ISA interface" then the virtual machines are categorized as *System VMs*. Moreover, within these two classifications, virtual machines are also classified depending on whether guest and host components use the same ISA. The classification is shown in detail in Figure 2.3. As it can be observed, there are different names for referring to each type of virtual machine. Multi-programmed Systems and Dynamic Optimizers are *Process VMs* on which guest and host share the same ISA. Dynamic Translators and High Level Languages (HLL) are also *Process VMs* but guest and host do not share the same ISA. On the other hand, the *System VMs* where guest and host share the same ISA are the Classic OS and the Hosted VMs, and the ones where guest and host do not share the same ISA are the Whole System and the Co-designed VMs.

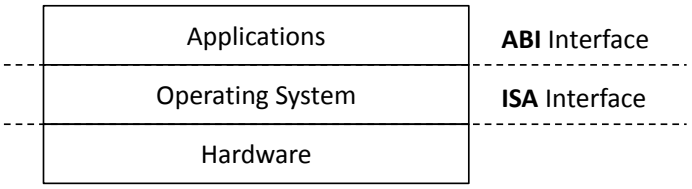


Figure 2.2: One possible VM software layer integration in the three component computer design.

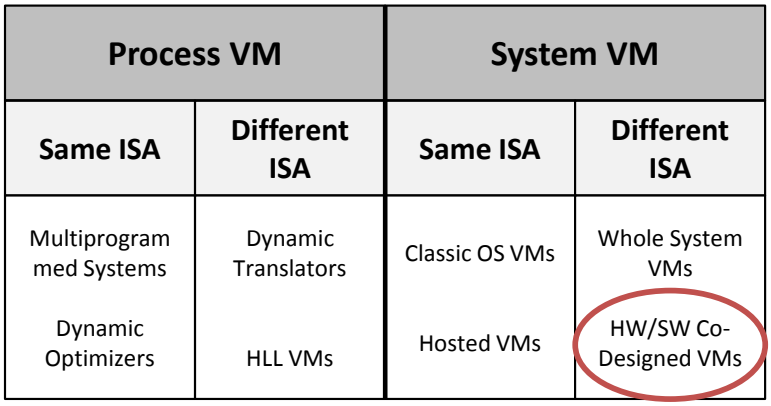


Figure 2.3: J. Smith's Virtual Machines classification.

In this thesis, we focus on the implementation of hardware/software co-designed virtual machines. We have chosen this category because its capability of improving the efficiency of simple processors without requiring the addition of significant hardware complexity. As commented in Section 1.1, the objective of this thesis is to propose mechanisms to implement an efficient and simple hardware/software co-designed processor that could be implemented in a multi-core chip in order to further exploit TLP and ILP.

The proposals described in this thesis can also be implemented in other type of virtual machines. However, we only describe its implementation in a hardware/software co-designed virtual machine approach in this manuscript.

The rest of the chapter is organized as follows. In Section 2.2, we provide a detailed overview of hardware/software co-designed virtual machines and its benefits compared to traditional processor designs. Later, in Section 2.3, we describe how these virtual machines work from a system level point of view, and in Section 2.4, we describe how they work from an internal point of view. We conclude this chapter in Section 2.5, presenting different alternatives to improve the application execution performance by implementing a hardware/software co-designed virtual machine.

2.2 Hardware/Software Co-Designed Virtual Machines

Smith and Nair call hardware/software co-designed virtual machines to the ones that implement the software layer at the "ISA interface" level in a system where guest and host do not use the same ISA set of instructions (see Figure 2.3 for more details). The software layer is placed between the hardware and the software computer components as shown in Figure 2.4. The software layer is tightly implemented with the hardware in such a way that both are perceived as a single entity for the upper components in the system.

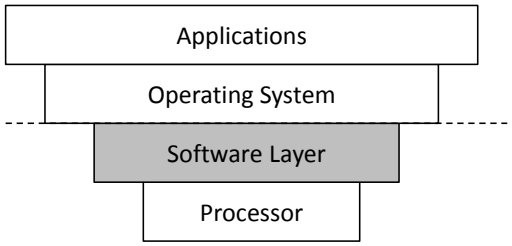


Figure 2.4: Co-designed Virtual Machine approach.

In fact, the software layer has full access to the hardware resources and it is transparent to the entire software stack. Since the hardware and software layer work in a collaborative manner, these designs are also known as hybrid hardware/software processors.

Hardware/software co-designed virtual machines present important advantages when compared with more traditional system designs. Among other things, the co-designed approach can be used to simplify the processor hardware components, to guarantee backward compatibility with previous designs, to improve application execution performance, to reduce product time to market, and to develop specialized solutions to the costumers.

The simplification of the hardware is achieved by passing functionalities from the hardware to the software layer. This hardware simplification usually translates into area and power reductions, without losing performance. For instance, Transmeta® processors perform instruction decoding in software without any hardware component involved in the process and they cache these decoded regions for later reuse [16].

The hardware and the software stack do not interact directly. Instead, they communicate through the software layer. This facilitates the backward compatibility with older designs, since the software layer may implement legacy functionalities and remove them from the hardware component. In order to accomplish this, the software layer includes a Dynamic Binary Translator (DBT) [24] [38] that transforms the original application code from the guest ISA to the host ISA. In addition, the software layer can also implement a Dynamic Binary Optimizer (DBO) [24] with the objective of improving the execution of the application in the processor. In this case, the optimizer also transforms the original application code from the guest ISA into the host ISA, but it is also able to optimize the code in order to take full advantage from the hardware specifics. The software layer is implemented very tightly with the hardware and it has direct access to certain components which allows it to adapt the application code in a very efficient manner. In

the next Section 2.5.1, we will cover the different optimizations than can be applied to the code in more detail.

Designing and manufacturing a traditional processor is a very complex and time consuming task that requires efforts from different groups of people (hardware architects, silicon manufacturers, validation engineers, among others). This scenario is even more complex when the different components in the system have to be developed in a collaborative manner. Therefore, by breaking the dependences among the components by the introduction of the software layer, the time required to develop the processor can be reduced significantly. This also translates in a time reduction for the product to reach the market. Moreover, the software layer can be modified and improved in less time than the hardware, which is also an advantage for fixing possible bugs in the processor implementation.

Finally, hardware/software co-designed virtual machines are also very appealing in order to offer specialized products to the costumers. Actually, not all users employ the processor characteristics in the same way, and not all the resources are equally utilized. Even by not modifying the hardware resources, the software layer can be adapted according to the user's needs, transforming the processor to fulfill the user experience. Applications can be adapted by the software layer during the time they are running on the system. Moreover, the software layer itself can also be modified based on the different user roles. In such a way, the processor of a graphical designer may include a software layer configured differently to the one that a biomedical researcher may require. Furthermore, since the software layer can be developed faster than the hardware, it is possible to incrementally include more functionalities if the user requires them.

2.3 Operation of a Co-Designed VM at System Level

The booting process of a hybrid hardware/software co-designed processor is different than in a traditional processor. In this case, the first component to be loaded in memory is the software layer and it is also the first component to be executed. The software layer code is stored in a dedicated ROM, normally placed in the motherboard [39]. The content of this ROM may be compressed and encrypted for security reasons. The software layer is stored in a protected memory area called the concealed memory [13]. This memory area is totally under the control of the software layer and it is non-visible for other applications and for the operating system. The rest of the memory in the system is used in a similar manner than in traditional systems. Once the software layer is loaded, the software layer takes control of the system and it executes specialized routines to proceed with the normal

booting of the system. Therefore, the next step is to load the operating system code to memory. From this point on, the booting process is similar to the one of a traditional processor, with the difference that the software layer controls all the activity of the processor driving the execution of all instructions to be executed by the hardware.

Since the software layer is also an application that runs over the processor, its instructions and data need to be stored in the concealed memory. Moreover, information to optimize/translate the original code of the application is also stored in the concealed memory. Therefore, the hardware/software co-designed processor requires storing the information about instructions and data being executed in the processor, as well as other three types of information: (1) information about the optimized/translated code, (2) information about the software layer itself, and (3) information about the execution of the instructions (meta-data). Most of the existing hardware/software co-designed systems implement internal memory structures by using software approaches or by combining hardware with software to store this information [30] [35] [36] [37]. Figure 2.5 details the distribution of the aforementioned types of information within system main memory. Note that all data internal to the software layer is stored in the concealed memory area whereas the rest of information from the applications is placed in the OS-visible memory area.

2.4 Operation of a Hardware/Software Co-Designed VM at Internal Level

In order to control the execution of the applications and the operating system, the software layer employs the three following internal operation modes: *native*, *emulated* and *interruption/exception* modes. The *native* mode is employed for executing the applications and the operating system instructions in a non-optimized manner. The code is simply translated from the guest ISA into the host ISA by the hardware or the software. Moreover, its semantics are not modified at all since no optimizations are applied, so that this mode is often used when the software layer needs to guarantee forward progress. This mode is also employed when there are guest ISA constraints that do not allow optimizations, as for instance, codes that handle peripheral input/output activity [40]. The *emulated* mode is used for executing the code belonging to the software layer itself and also the optimized versions of code belonging to the original application. In this case, depending on the optimizations applied to the code, forward progress is not guaranteed. By contrast, by using this mode the application performance can be improved. Finally, the *interrupt/exception* mode is in charge of executing the code of interrupt and exception handlers that may be raised during the execution of the instructions. The software layer

uses this mode to communicate the interruptions and the exceptions to the applications and the operating system in order to satisfactorily solve them. The software layer decides in which mode it should work depending on the execution state of the application. Figure 2.6 shows the transitions that the software layer realizes to move across the three operational modes. As it can be observed, on interrupts or exceptions code events, the software layer always transitions to the *interrupt/exception* mode no matter which is the original state. However, once the interrupt or exception event is solved, the software layer may transition to the *emulated* or to the *native* operational mode depending on the previous state before the initial transition to the *interrupt/exception* operational mode. To enter in *emulated* mode it is required to execute optimized or software layer code, otherwise the system works in *native* mode. On the other hand, if it is necessary to execute non-optimized code when the software layer is in *emulated* mode, then the software layer transitions to *native* operational mode.

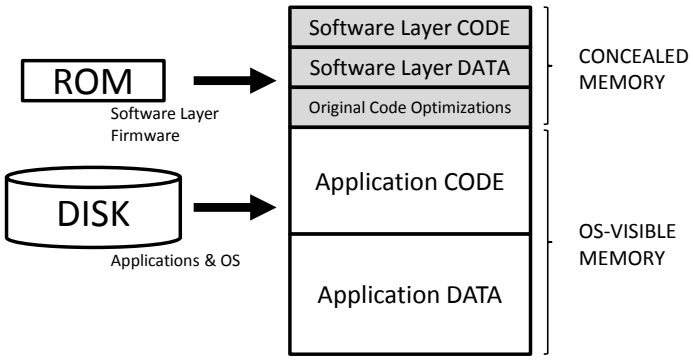


Figure 2.5: Memory distribution of a hardware/software co-designed system.

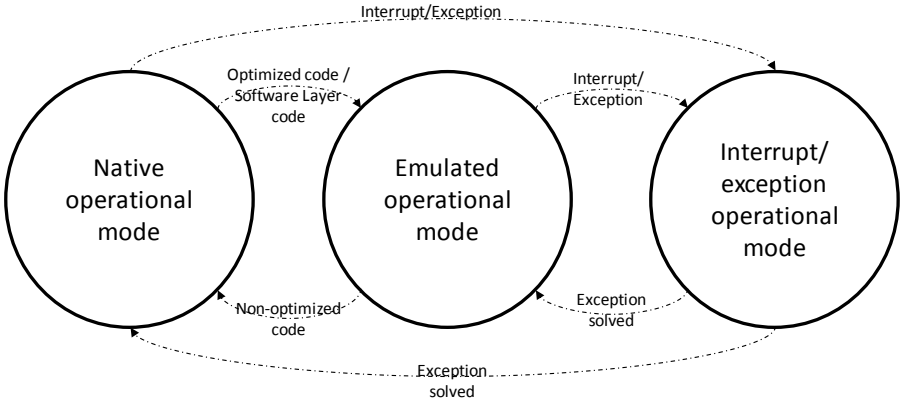


Figure 2.6: Hardware/Software co-designed virtual machine internal operational modes.

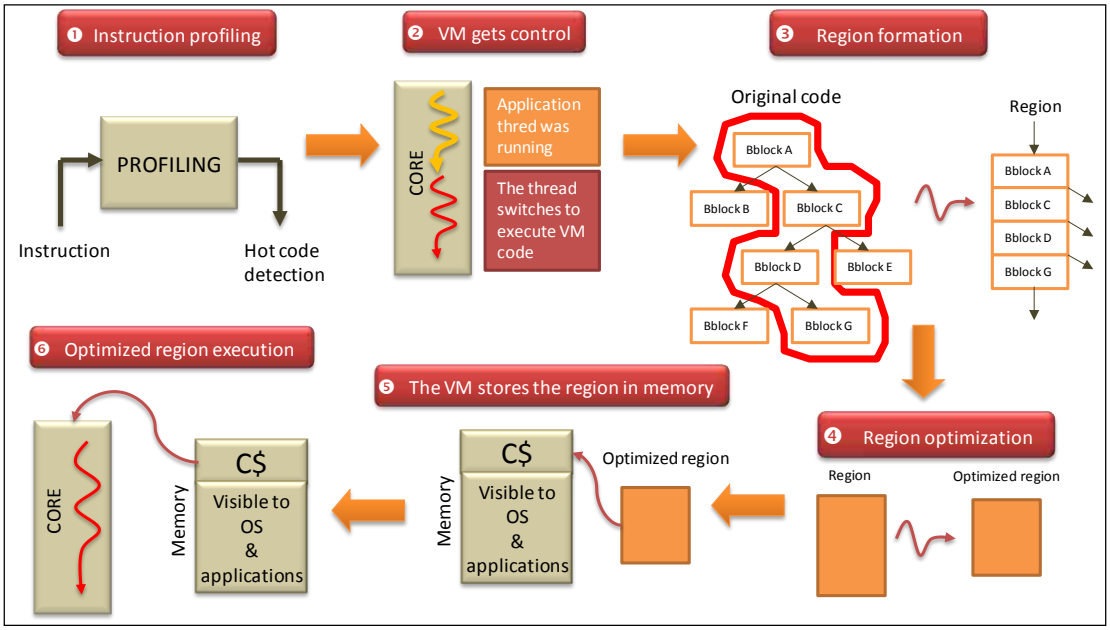


Figure 2.7: Execution Life Cycle of a Co-designed VM.

The *native* and the *emulated* modes accumulate most of the execution of the applications in the system. Deciding which regions of code should be executed in one mode or the other is one of the key points of the hardware/software co-designed virtual machine execution. This process is commonly called the execution life cycle of a code region and it is represented in Figure 2.7. At the beginning, the system does not contain optimized versions of code and the software layer executes the applications in the *native* mode. During this time, it analyzes the application execution with the main objective of detecting the instructions of code that are more suitable to be optimized. As soon as the software layer determines that one region of code is suitable to be optimized then it gets the control of the processor by switching from the currently being executed application code to a special software layer routine that will be in charge of applying the optimizations to the code. Since this code is from the software layer itself, the *native* mode cannot be used and then the software layer switches execution to the *emulated* mode. During the analysis of the applications, the software layer builds an internal representation (normally by using basic blocks) of each one of the original guest ISA code regions. It also stores statistical information about how the code is being executed within the processor. The process of optimizing a region takes this internal representation with the statistical information and builds a new optimized region but in the host ISA instruction set. The new representation of the code is then placed inside the concealed memory in an internal memory structure called the code cache (C$) [13]. From this point on, the software layer uses the optimized representation of the code stored in the C$ in

order to improve application execution when possible. The code from the C$ is always executed in the *emulated* operational mode of the software layer.

From a memory point of view, the execution life cycle of a region requires a structure for detecting the regions of code to be optimized, a structure for placing the optimized versions of code and a structure to store the information required to switch processor execution from a non-optimized region to an optimized version of it. In the rest of this section we will cover all these structures in more detail. Note that the other information required for the execution life cycle of a region is stored and managed as simple data within the virtual machine concealed memory.

2.4.1 Detecting Regions to be Optimized

The software layer uses profiling for detecting the regions of code that are more frequently executed in the system [13]. The profiling requires storing information about the number of times each instruction has been executed. Periodically, the software layer analyzes this information and, based on the number of executions of the instructions it decides which codes should be optimized. A region of code that has been executed enough times to be considered for optimization is commonly called *Hot Code Region*. These regions can be basic blocks, superblocks [41], routines, and loops, among others types. It is a decision of the software layer designer to select the more appropriate type of region, normally based on the optimizations to be applied on top of them. In Chapter 4 and Chapter 5, we use superblocks to represent code regions. Superblocks are formed by a sequence of basic blocks as shown in the example of Figure 2.8. Superblock regions may have multiple exit points but only one entry point is allowed. More details about superblocks are given in Section 5.2.1.

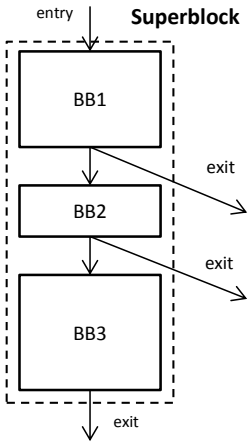


Figure 2.8: Example of a superblock code region.

The most appealing hot regions to be optimized are those that significantly improve the performance of the application without incurring in big overheads to the system. Therefore, detecting these regions is not a straight forward problem. In the literature, we can find different examples about how to detect hot code regions. These proposed solutions can be classified as hardware profilers and software profilers. Hardware profilers [26] [17] [42] [43] usually implement a hardware table that keeps track of the number of times each static instruction has been executed. Since the table is implemented in hardware its size is usually small, which compromises the accuracy of the profiler. By contrast, its management is fast and does not introduce significant overheads to the system. On the other hand, software profilers [16] [30] [35] [36] [44] [45] [46] rely on memory table structures. Special software instructions are executed in the processor to keep track of the execution of the information and for updating the data of these tables. The execution of these instructions adds an additional overhead to the system since they are executed intermixed with the original instructions of the application. However, the table size is not constrained as in the hardware profiler schemes and, therefore, software solutions normally present higher accuracy.

In an efficient implementation of a hardware/software co-designed virtual machine, the profiler should introduce low cost and complexity to the system. Moreover, it should introduce minimum overheads in order not to compromise the benefits that can be obtained by the execution of the optimized regions. Therefore, pure software and pure hardware based solutions are normally discarded because the overheads are high in the formers, and the accuracy is low in the latters. Fortunately, a hybrid hardware/software solution is possible in a co-designed virtual machine [17] [27] [18]. For instance, this may be done by just letting the hardware to count instructions and the software to analyze the information for later building and optimizing the hot code regions.

2.4.2 Storing Optimized Regions

Once a code region has been optimized, it is stored in memory for further reuse to avoid the overheads of re-optimizing the region again. Code regions are stored in the code cache (C$). The main design decisions that determine the effectiveness of the C$ are the identification of the regions within the structure, the region placing and replacing mechanisms, the size of the structure, and the persistence of the information.

The identification of the regions is done by the indexing mechanism. This mechanisms in charge of identifying where the information within the C$ resides. Since the optimized regions stored in the C$ are modifications of the original regions, the indexing mechanism

requires to identify the relation between them. This identification must be fast, otherwise every time a region needs to be searched it would be very costly to use it and the possible benefits of its optimization would be compromised. In the literature, there are different studies focused on reducing the costs of the indexing mechanisms, like for instance Indirect Branch Handling Mechanisms [47] and Region Chaining [48]. These techniques avoid returning to the software layer code once a region has been executed if it is known that another optimized region is the next to be executed.

The placing and replacing mechanisms are in charge of determining the best position within the C$ for the optimized regions and which region or regions are the best candidates to be removed from the structure when there is no free space. Since optimized regions are inserted, deleted and replaced from the code cache and their sizes are not fixed, we may have fragmentation problems in the C$ [49]. Actually, fitting regions of variable size in a limited space is not a straight forward problem [50].

For an efficient design of the C$, the aforementioned indexing, placing, and replacing C$ mechanisms should satisfy the following requirements: (1) to introduce low overhead, avoiding extra costs to the system, (2) to maximize the data temporal locality by keeping the most frequently executed regions in place, and (3) to effectively utilize the resources avoiding fragmentation.

The size of the C$ is one of the most important design decisions for maximizing the effectiveness of the structure and for determining the most convenient implementation of the indexing, placing, and replacing mechanisms. The C$ size is dependent on the instruction footprint of the applications. Actually, its size tends to grow up to five times the size of the footprint of the executed application [51]. Over-dimensioning the size of the C$ may imply a waste on the memory resources and it complicates the implementation of the required mechanisms to correctly handle the structure. By contrast, a small C$ is also counterproductive because optimized code regions won't fit in it and re-optimization will occur more often. The C$ size should meet a compromise between the efficiency of its handling mechanisms and retaining the most frequently executed regions in place. Several works focus on the management of the C$ [37] [49] [51] [52] [53].

The last C$ design decision refers to the persistence of the information across different application executions. The basic C$ design is to remove all the content of the structure on every application context switch. This solution rely on the fact that the optimized code between context switches is small and hence flushing C$ during context switches is enough

to make room for the necessary regions [36]. However, it is possible to save the information of the C$ in other memory structure to recover it in a future application execution [35].

Finally, the C$ management is also cumbersome because it has to deal with several corner cases to guarantee correctness [54]. First, some systems have special situations where regions cannot be directly removed from the C$. For instance, when software layer exceptions[2] are being served in the middle of the execution of a region, the software layer requires the stored version of the region to remain in the C$ in order to know the returning address once the exception completes. Second, when the OS decides to force C$ region evictions because a page is de-allocated from main memory, the management mechanism have to be conveniently informed, otherwise the placing and replacing mechanisms would use incorrect information and would incur in incorrect decisions.

Finally, regions stored in the code cache could become obsolete in ISAs like x86 that support Self Modifying Code (SMC) if the original code is modified. In these cases, C$ management has to prevent the execution of obsolete optimized versions of the modified code.

2.4.3 Using the Optimized Regions

Once a code region has been detected, optimized, and stored inside the C$, it should be used every time we want to execute the original code it represents. Therefore, the software layer accesses the C$ looking for the optimized version of the original code that is going to be executed. In the case that the optimized region is not present, then the software layer will re-optimize it again (if the region is still considered as hot). On the other hand, if the region is in the C$ then the software layer executes its optimized version. The searching mechanisms may incur in significant execution overheads [55].

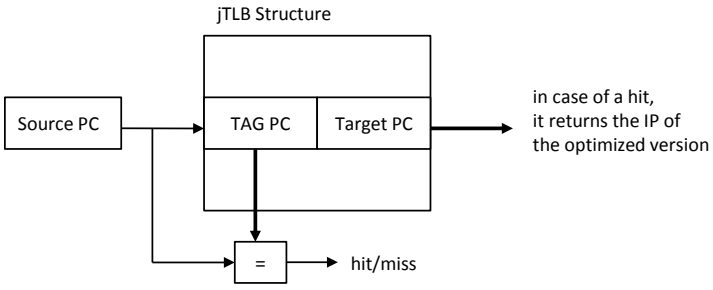


Figure 2.9: jTLB implementation.

[2] Software layer exception does not require updating the architectural state of the processor.

Some hardware/software co-designed virtual machines implement their search mechanisms by software [14]. This software approaches use lookup hash tables. However, other co-designed virtual machines implement a dedicated hardware structure called the jump TLB (jTLB) to minimize the execution overheads. The jTLB keeps track of the relation between the original region identifier and the optimized region identifier within the C$ [31]. Every time an original code region is going to be executed, the jTLB is queried in order to determine if there is an optimized version for it. The lookup procedure uses the original region identifier to index the table and if it exists then a hit and the identifier of the optimized version of the code is reported. In case that the data is not in the table, a miss is reported with no identifier associated. Figure 2.9 shows an example of the functionality of a generic jTLB. As it can be observed, it is indexed by the starting PC of the original region that acts as the original region identifier. In case of hit, the table returns the target PC of the associated optimized region that acts as the identifier of the region. Note that a hit always implies that there is also an optimized region in the C$, whereas a miss does not imply that there is no an optimized version. In this latter case, the software layer still needs to access the C$ to see if there is the region. Note that the jTLB structure is very dependent of the C$ and both structures need to work in a cooperative manner.

2.5 Improving Program Execution via a Co-Designed Virtual Machine

2.5.1 Code Optimizations

An optimization consists on transforming the original application code into a new version that can be executed in a faster manner or using fewer system resources. The software layer dynamically analyzes the execution of the instructions executed by the processor and applies optimizations to them in order to improve their performance execution.

The optimizations applied to the codes are classified as platform dependent and platform independent and the co-designed virtual machine can perform both of them indistinctly. The platform dependent optimizations take advantage of the architectural characteristics of the system. Actually, since the software layer knows in detail the components that form the underlying hardware, it can apply very precise optimizations to the codes to take advantage from the specifics of the architecture. For instance, the software layer can apply optimizations to the code in order to improve the accuracy of the branch predictor [56]. Platform independent optimizations are not designed to take advantage from any

particular system and they are effective on most of the systems. In the literature, there exist different types of independent optimizations that the software layer can adopt in order to improve the application performance execution [57]. In particular, there are optimizations oriented to improve loop regions [57], such as loop fission, loop fusion, and loop unrolling. Other optimizations are focused on improving data-flow execution, such as common sub-expression elimination [58], constant folding propagation [59] or induction variable recognition and elimination [60]. There are also SSA-based and code generation optimizations, like register allocation [61], instruction scheduling, and reordering computations [60]. Finally there are optimizations like dead code elimination [60], macro compression [62], and reduction of cache collisions that can be broadly applied to different types of codes.

The benefits obtained from the different presented optimizations are variable. This variability is not only dependent on the type of optimization but also on the characteristics of the codes on which they are applied. The same optimization applied to different codes may not offer the same benefits to all of them. There is not a perfect rule for knowing in advance the best optimization to be applied to the codes. Moreover, optimizations can be combined together, including dependent and independent ones, which complicates even more the selection of the optimizations to be applied. It is responsibility of the software layer to find the best trade-off between the cost of applying the optimizations and the benefits that can be obtained by using them. The software layer is able to analyze the code execution before and after applying the optimizations and, moreover, it is also able to dynamically re-optimize the code, even by changing the type of the optimizations, if the benefits are not the expected ones.

The general rule is that code optimizations must not affect the correct execution of the application. However, it is possible to apply very aggressive speculative optimizations in order to get higher performance improvements. The main idea of these techniques is to start doing work in advance before knowing whether it will be really required or optimizing the code making certain assumptions that are not always true. Allowing speculative code execution requires special mechanisms to avoid corruption of the architectural state in case that the optimization jeopardizes the correctness of the execution. Basically, any outcome of the speculative execution cannot become part of the architectural or memory state until the correctness of the execution is validated. In next section, we cover in more detail the required mechanisms that a hardware/software co-designed virtual machine requires in order to allow speculative code execution.

2.5.2 Mechanisms to allow speculative code execution

Hardware/software co-designed virtual machines are allowed to use speculative optimizations to achieve high performance improvements. However, speculation implies generating values that may be incorrect and may compromise the correctness of the execution. In this case, the software layer needs to guarantee that the architectural state of the system is not updated until the optimized code is correctly executed. Correct execution means producing the same values as if the code was executed in the original sequential order. In order not to promote incorrect speculative values, the software layer prevents any change on the architectural state to become visible outside the core until the region completes execution. If the region cannot be executed completely for any reason, the work is undone and the architectural state is recovered to the point it was at the beginning of its execution. The execution of a region is also cancelled if an interrupt or an exception occurs since they have to be handled as if they were produced in the original sequential application execution [63]. On the other hand, if the execution of the region is correct, then the architectural state is updated accordingly and the speculative values are considered correct and visible for other cores. The regions are executed atomically since all instructions within them retire or any one does [64]. When a problem occurs during the execution of the speculative region, we say that the region does a *Rollback*. By contrast, when the execution is correct, we say that the region does a *Commit* [65].

The architectural state of the system comprises the registers and the memory. The speculative execution of a region cannot produce values updating these structures directly because other cores could see incorrect values and proceed generating wrong results. The software has to guarantee that these both structures are updated conveniently guarantying application execution correctness [63]. The most common solution for storing correctly the architectural state of the processor registers is by using a Shadow Register File [15] [16]. As it is described in Figure 2.10, the shadow register file is used by the software layer to make a copy of the value of the registers just at the beginning of the execution of the code region. In fact, this is a checkpointing mechanism that makes a safe copy of the architectural state of the registers at this point. Therefore, there are two copies of the architectural registers coexisting simultaneously in the system. One containing the checkpointed values (shadow register file) and another containing the values produced during the execution of the region (normal register file). In case of rollback, the shadow register file values are copied into the normal register file, recovering the architectural state of the registers at the beginning of the execution of the region.

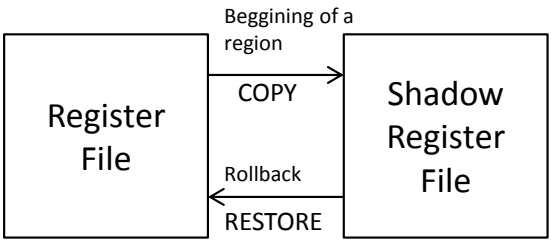


Figure 2.10: Shadow Register File.

In traditional processors, memory-write operations (stores) are buffered in a hardware structure, called Store Buffer, before their promotion to main memory [66]. This allows the store operations to be executed even if previous operations have not still retired and the data is promoted to main memory only when the store operation correctly retires on the processor. Co-designed virtual machines normally take this idea and extend it to handle the problem of updating correctly the architectural memory state of the processor. The extension consists on adding a gate barrier to the store buffer to avoid the promotion of stores that belong to code regions that are still pending of commit (these stores are executed speculatively). Note that region commit also requires all stores within the region to correctly retire. The buffer structure with the additions is commonly called the Gated Store Buffer [67]. In Figure 2.11, we illustrate how the structure works by using a simple example. Within the buffer, there is the data of store operations belonging to different regions. The data of stores that belong to code regions that have already committed are called senior stores. In the figure, they are located at the right part of the buffer and highlighted in dark gray color. The store data of regions that have not committed are called non-senior stores. In the figure, they are located in the left part and highlighted with light gray color. Between both types of stores, it is placed the gate barrier. This barrier separates stores that can proceed to main memory from those that no. The gate barrier position changes every time a region commits and it moves to the first store of a region that has not already committed (or to the last store belonging to a last region that has already committed). In case of rollback, all non-senior stores are directly removed from the buffer and they do not update the main memory content. Therefore, all senior stores before the gate barrier can be promoted and all non-senior stores after the gate barrier need to wait for their regions to be committed.

The main problem of the Gate Store Buffer is that regions with more memory store operations than the size of the structure cannot be executed in their optimized version because the amount of speculative data generated cannot be temporarily stored on it. This problem is exacerbated when the software layer operates with big regions (for instance

regions with more than 100 instructions which imply approximately 30 store operations on average per region) or it allows multiple threads running simultaneously on the same processor. There are other approaches, based on transactional memory that can also be employed in order to prevent the promotion of speculative data to the memory [10] [68] [69] [70] [71] [72] [73]. The fundamental of these techniques is to use the regions as if they were atomic transactions [74]. A transaction is handled as a sequence of instructions that execute atomically (as regular code regions from the co-designed virtual machine definition do). In general, transactional memory techniques use the memory cache structures of the processor to keep the generated data by the transactions before promoting it to main memory. In such a way, the space for keeping the speculative data is bigger than in the Gate Store Buffer proposal which makes these solution more scalable when regions are big or when multiple threads run simultaneously. By contrast, these solutions are more complex from a hardware point of view.

The common scenario is to commit the regions and only rolling them back very rarely. Therefore, the more appealing memory transactional techniques to be adopted by the software layer are those that introduce minimum overhead to the system when regions execute correctly, even if they have higher overheads when rollbacks occurs. This is the case of well-known techniques such as LogTM-SE [72] and others based on it. However, low rollback overheads are also helpful since they enable the utilization of more aggressive optimizations.

Transactional memory as described in this section is still under research to be fully adopted by co-designed systems. In particular, the Transmeta® processors use the Shadow Register File in both Crusoe® and Efficieon® for the registers, and for memory, the Gated Store Buffer in Crusoe® [16] and a simplified transactional memory based mechanism in Efficieon® [75]. However, there are no real commercial processors using the aforementioned transactional proposals

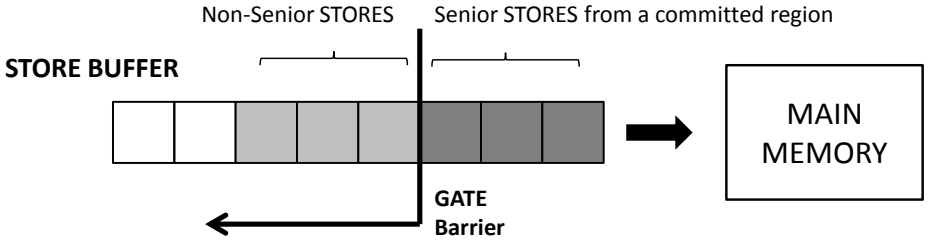


Figure 2.11: Gated Store Buffer implementation.

2.5.3 Trade-Offs

Co-designed virtual machines employ a set of actions that introduce an additional cost compared to traditional systems. These actions are: the execution of the application code in native mode, the detection and optimization of the hot code regions, the storage and search of the optimized regions, the correct execution of the regions in the processor, and depending on the processor hardware, the emulation of legacy functionalities. Moreover, the software layer uses the processor resources for its execution. This means that it competes for resources with the applications that are being executed and therefore, the applications do not make forward progress while the software layer executes. Thus, the optimizations applied to the regions of code need to improve the application execution performance, at least the minimum necessary to compensate the overheads introduced by the execution of the software layer. However, optimizing the code is one of the most costly actions performed by the software layer. Thus, it is very important to find the best trade-off between the benefits that can be obtained by applying the optimizations and the overheads of making them possible. For instance, the best optimization to be applied to a particular code may require high overheads that may not be covered by the benefits obtained. On the other hand, a simple optimization may give huge benefits if it incurs in very low overheads.

Overall, the software layer has to be a highly tuned code where all components should perform their actions as efficient as possible. All of these components cannot be tuned in an isolation manner since all of them are inter-related. For instance, detecting hot code regions fast and accurate is crucial to maximize the benefits of the optimizations. If the regions are not frequently executed then the optimizations are useless. Moreover, maintaining the optimizations in the C$ is necessary to avoid re-optimizing the code multiple times, and implementing a fast mechanism to get the regions from the C$ is important to execute as soon as possible the optimized code. Therefore, all the software and hardware components involved in the aforementioned software layer actions must be designed with the objective of reducing the overheads and maximizing the benefits that can be obtained by their use. As examples, in this thesis, we present a very efficient implementation of a fast and accurate hot code detector and an efficient implementation of a hybrid hardware/software register checkpointing mechanism.

2.6 Baseline Hardware/Software Co-designed Processor

In this thesis, we have considered a hardware/software co-designed processor which its main operational flow matches that presented in Section 2.4. This processor has been

employed as the baseline for all our implementations. It uses hardware profiling for detecting code suitable to be optimized and relies on the software layer to build regions by using superblocks. Optimized regions are stored in a C$ and are accessed by employing a jTLB. The software layer applies optimizations such as dead code removal, instruction reordering, etc. In order to guarantee the correct update of the architectural state of the processor we use a Gated Store Buffer for the memory and a Shadow Register File for the registers.

Chapter 3

Detecting Hot Code

The ability of analyzing the code executed by the processor with the purpose of dynamically improving performance/power is one of the main characteristics of hybrid hardware/software co-designed systems. Such improvements come from either dynamically adapting the code to the particularities of the internal architecture or from applying aggressive optimizations to eliminate unnecessary computations or reducing the critical paths. The application is analyzed from different points of view in order to gather information related to its execution. For instance, the detection of memory operations with high cache miss-ratios can be used for enhanced explicit prefetching strategies or for improving instruction scheduling in in-order processors, while the extraction of conditional branches behavior can be used to create better code regions for optimization or to convert biased branches into assert instructions. However, this information extraction process introduces an additional cost to the system. First, the code has to be carefully analyzed and second, the application code has to be transformed and/or optimized by a software layer referred to as the optimizer. Both steps are time and resource consuming. Thus, the cost can be amortized only by improving significantly the execution of the transformed code. One way for reducing these costs is to only optimize the most frequently executed code regions of the application, also known as hot code regions. Transforming only these regions is less costly than optimizing the whole application code and they still represent the major part of the execution of the application. This chapter presents a novel hardware/software co-designed mechanism for detecting the most frequently executed application regions in a very efficient manner.

This chapter is organized as follows. First, we introduce how a co-designed system is able to collect information from the application execution. In Section 3.2, we describe how to detect the most frequently executed code regions during program execution in detail. In

Section 3.3, we propose a hardware/software mechanism for detecting these code regions and later, in Section 3.4 we evaluate the proposal. Finally, in Section 3.5 we conclude the chapter by presenting the conclusions and the future work.

3.1 Profiling

3.1.1 Description

Profiling in computer engineering is a technique used to analyze the execution of an application when it is running in the processor. This analysis is done by studying information that is collected at runtime and that it is related to particular application characteristics such as the memory usage requirements, the amount of branch instructions correctly predicted, or the amount of arithmetic operations processed, among others. The components required to collect the information and analyze it form what is commonly called the profiler.

Application profiling can be done in many different ways and this leads to multiple types of profiler designs. However, all of them share a common structure depending on two basic aspects. The first is related to "how they collect the information at runtime". The second is related to "when the information is processed". In this chapter, we will start describing the differences related with "when the information is processed" by introducing the concepts of online and offline profiling. Later, we will cover "how the information is collected" by describing the concept of program events and also by introducing the two more typical approaches to roll up such events: by hardware and by software.

3.1.2 Online versus Offline Profiling

Profiling is always performed while the application is running on the system. However, such profiling is classified as online or offline profiling depending on when the collected information is processed. Online profiling collects and processes the information while the application is being executed in the system. That is, both collection and processing are performed at runtime. This type of profiling is also known as dynamic profiling. On the other hand, offline profiling collects the information in a first execution of the program and it processes such information once the execution is finalized. Processed information is often used to generate better versions of the program for later executions. This type of profiling is also known as static profiling. In the rest of this thesis, we will use online/dynamic and offline/static terms indistinctly.

The main advantage of dynamic over static profiling is that the collected information in the former can be used to enhance the behavior of the application using the same input

set. In other words, the application can be transformed at runtime using information from the same execution. By contrast, in the static approach the input set used to collect profiling information may be totally different than the input sets used for later executions. In order to avoid collecting meaningless data, one needs to use representative input sets. It is important to note that, although in some cases it is possible to use the same input sets in both executions, there are particular programs, such as games or multimedia applications, where this is not feasible because their input sets are unknown until the user executes them.

However, a drawback of online profiling is that the use of the collected profiling information is limited to the ambit of execution of the application itself. This implies that accurate profiling information needs to be collected fast in order to spend a significant amount of time executing the enhanced version of the application. Furthermore, profiling information needs to be collected every time the program is executed.

One general inconvenient in profiling is that application execution may pass through different execution phases. In other words, one application could behave totally different from one moment in time to another. This has implications for static and dynamic profiling. For the latter, phases may imply that collected information is obsolete once it is used. For this reason, it is important to review periodically the profiling information and react when significant changes are observed. On the other hand, phases may be totally different from different input set executions in static profiling. In addition, phases may be hidden in offline profiling as average profiling statistics are often generated.

In general, static profiling is used in traditional compilers, whereas dynamic profiling is commonly used in dynamic binary optimizers and translators such the ones implemented in hardware/software co-designed systems.

3.1.3 Gathering Information

3.1.3.1 Program Events

As we have commented, profiling is based on gathering information about particular application characteristics. This information is collected via events. An event describes a particular architectural or micro-architectural situation that happens during the execution of an application. As example of events once can enumerate data cache misses, branch miss-predictions, registers usage, among others. It is the responsibility of the computer architect to define what events are required by the co-designed system in order to apply transformations/optimizations to the code.

In general, profiling information is gathered by counting the number of times an event occurs. There are two ways of processing the information once it has been collected. On one hand, there is the event-based alternative where the analysis of the information starts when a particular counter exceeds a fixed number of occurrences (threshold). In this case, a control flow disruption occurs (e.g. an exception) and the counters are analyzed (see Section 3.1.3.2). On the other hand, there is the statistical alternative, where particular event counters are analyzed after fixed periods of time. In this second alternative, the time between two different reads of the same counter/s can be programmed. It is important to note that event counters are normally exposed and they can be read at any time during program execution.

3.1.3.2 Hardware and Software Approaches

There are two basic approaches for counting events during program execution. On one hand, one can add specific hardware structures to track and count events. On the other hand, one can count events by inserting instructions in the original code. Since this approach does not use additional hardware, the profiled information is stored directly in main memory for its later analysis. Solution based on the first alternative are called hardware profilers, whereas the alternatives based on the second approach are called software profilers.

Both alternatives have their own pros and cons. On the one hand, the hardware based solutions require more processor area than the software based ones. On the other hand, the software based solutions may slow down the execution of the application and the additional instructions may impact the profiled behavior of the application. Note that although we have presented both alternatives in a separate way, there are no limitations for combining them in hybrid solutions.

3.2 Profiling for Hot Code Detection

3.2.1 Objectives

The main goal of this work is to design a profiler for detecting the most frequently executed pieces of code of an application. These pieces of code, also called hot code regions, are very appealing for dynamic binary optimizers because they normally represent a small fraction of the static code but they account for the majority of the dynamic instructions. Thus, optimizers could globally improve application execution by optimizing just these regions requiring less time and resources than optimizing the full program code.

Detecting hot code regions only represents one part of the total profiling information required for optimizing the code globally. Depending on the type of optimizations/transformations different profiling information may be required. In any case, the detection of hot code regions is the pillar upon which the hardware/software co-designed processors are designed, since these regions describe the scope where such optimizations are performed.

A profiler designed for detecting hot code in a co-designed system should satisfy the following characteristics:

- **Low overhead**: hot code detection is done during program execution. Thus, it should not interfere with the application and it should not impact its performance. All overheads, included the ones from the optimizer, should be compensated by the execution of the optimized versions of the code.
- **High accuracy**: events should be accurately counted by the profiler. Only most frequently executed regions should be detected since detecting regions that are not executed very often or missing regions that are executed very frequently would translate in suboptimal performance. In addition, the profiler should always provide up-to-date information from the application execution. Decisions taken in the past may not be correct in the future depending on how the application is being executed.
- **Low cost and complexity**: one of the main goals of a co-designed hardware/software system is to reduce hardware complexity without harming performance. Thus, resources dedicated to the profiler should not compromise the processor design hardware and the software requirements.

3.2.2 Related Work

Profiling for detecting hot code regions amenable for optimization has been investigated from different perspectives in the literature. In general, prior art proposals have been classified into the three following categories: software, hardware, and hybrid (hardware/software) based solutions.

Profiling in software is achieved by modifying the original program binary. Special profile instructions dedicated to count events are introduced between the original application code. Traditionally, these application modifications are done by a code instrumentation tool [76] [77]. Usually, these tools do not require recompilation in order to generate the binary with the profiling instructions attached to it.

Ball and Larus first demonstrated how to build code control-flow events in an efficient manner [78]. They use instrumentation techniques in order to break the dynamic program execution into acyclic and intra-procedural sequences of code, called BL (from Ball-Larus) regions. The number of BL regions executions is counted in order to determine their contribution to the total program execution. Thus, regions with higher profiled counter values are the most frequently executed ones. However, BL regions do not include loops and are also limited to procedure boundaries. These two region formation constraints limit the type of optimizations that can be applied to the code. Moreover, the required code for generating the profiled information introduces an overhead that ranges from 30% to 45% of total program execution time [78].

Later approaches focused on detecting more complex regions in order to extract more fine-grain information and they apply more complex code transformations. For example, Whole Program Path (WPP) technique produces an entire dynamic program control flow by combining acyclic regions, very similar to BL regions, which conforms a complete sequence of code [79]. However, these more complex regions require large amount of data to be stored in memory and the profiling required for its generation is also more expensive than the one required for generating simpler BL regions.

Tallam et. al also improved the initial BL region proposal by extending the code paths to consider regions across loop iterations and across procedure boundaries [80]. However, this technique incurred in 4 times the overheads introduced by the BL approach. On the other hand, several techniques have been directly designed on reducing the overheads of path profiling [81] [82] [83]. However, the overhead reduction is normally done at the cost of increasing the time required for producing representative profiling events. These techniques use less accurate profilers, and more time is required to gather events in order to get representative information from the application in consequence.

The overheads introduced by software profiling can be reduced by adding hardware support to the profiler. Early designs have taken advantage from existing branch handling hardware in order to generate profile information during program execution [84] [85]. However, the small size of the branch management hardware structures compromises the accuracy of the obtained profile information, and hence the type of optimizations that can be applied to the code are limited [25]. In order to overcome this problem, the Profile Buffer technique was proposed as a hardware profiler with higher accuracy than previous designs [26]. This profiling technique gathers the number of times a branch has been taken or not-taken and it stores this information into a set of hardware counters that are later

used to optimize the code by applying superblock scheduling [86]. Since the Profile Buffer does not assign events to specific static instructions, the later ProfileMe technique improved it by adding this feature and also by providing additional information about paths using historical outcome resolution from branches [17].

One different approach for building frequently executed code regions and for dynamically optimizing them is the Trace Cache [87]. In this case, a hardware structure stores dynamic basic block code regions forming straight line code sequences known as traces. As a consequence, the Trace Cache allows the processor to fetch such traces potentially spanning multiple basic blocks in a single cycle. Although several techniques have been proposed for optimizing these traces, they have traditionally been limited to classic optimization because of traces' small sizes and short lifetimes.

The Frame Cache was later proposed for solving these aforementioned drawbacks [88]. In this case, the Frame Cache generates longer traces than the Trace Cache by speculating on branches. Actually, regions are built considering high biased branches, so that region control-flow is re-directed to the most taken outcome of the branch. By contrast, an assertion is added to the less frequently taken outcome of the branch. Therefore, a fail in the speculation raises an exception with the objective of recovering the initial state of the machine (prior to the execution of the region). In order to guarantee forward progress execution, the region is then re-executed in a non-speculative form. In this approach, the cache line size and the amount of biased branches in the application limit the size of the detected regions.

Merten et al. proposed the Hot Spot Detector [18] with the objective of identifying longer regions than the ones detected by the Trace Cache without incurring in code speculation as the Frame Cache does. The Hot Spot Detector determines at runtime the set of most frequently executed branches that conforms a region called the hot spot. The profiler uses a table called Branch Behavior Buffer that monitors the behavior of application branches during a fixed amount of time. Once this time has elapsed, the buffer is used in order to generate a code region. Although the overhead required for trace generation is low, the required hardware is more complex than in previous approaches.

Apart from the aforementioned research proposals, processors hardware support for profiling events may be also used nowadays for detecting the hot code regions [89] [90]. In this case, the hardware profiling is combined with a software solution able to read the events and detecting the interesting code regions. Other proposals combining hardware

and software have been proposed, such as the aforementioned ProfileMe mechanism which does the region formation in software by calling a specialized OS routine [17]. Also the Relational Profiling introduced by Heil and Smith combines hardware and software by gathering events in hardware and by using a software application running on a dedicated co-processor in order to perform the region formation and the optimizations to the code [27].

As it is reflected in this related work summary, software profiling solutions for hot code detection suffer from important execution overheads whereas hardware solutions normally sacrifice profiling information accuracy and hardware costs. However, as also commented, hybrid hardware and software approaches overcome part of these two drawbacks by reducing the overheads by adding special hardware resources and by improving the accuracy by using dedicated software functions. Therefore, co-designed systems such the one described in this thesis, normally use hybrid hardware/software solutions for detecting hot code regions. In next sections, we describe in detail how a hybrid profiler should be designed for detecting in a very efficient manner the hot code regions when used in a co-designed system.

3.2.3 Application Characterization

3.2.3.1 Static and Dynamic Instructions

Static instructions is the term used to refer to the instructions that form the application executable binary. Static instructions executed by the processor are called dynamic instructions. Due to the program dynamic execution flow, a static instruction can be dynamically executed zero, one, or several times. Taking the example from Figure 3.1, a program formed by the static instructions represented on the left of the figure has two different dynamic executions depending on the outcome of the control instruction “INS 2”. These two case examples are represented in the figure as dynamic codes ‘a’ and ‘b’. In case ‘a’, the dynamic instruction control flow redirects the program execution to the loop of instructions formed by “INS 3” and “INS 4”. In this case, the loop iterates several times and “INS 5” is never executed. On the other hand, in case ‘b’, the loop is not executed and the control flow is redirected to instruction “INS 5”. Note that in case ‘a’, the 6 static instructions may contribute with several dynamic instructions (depending on loop conditions driven by control instruction “INS 4”), whereas in case ‘b’ the number of dynamic instructions is 4.

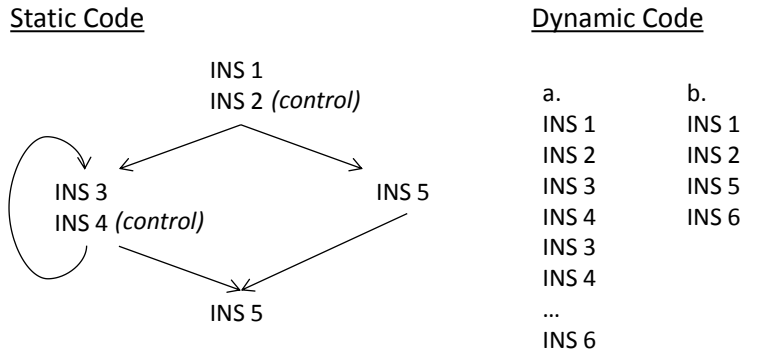


Figure 3.1: Static vs dynamic instruction code.

In current applications, the number of dynamic instructions is usually several orders of magnitude higher than the number of static ones due to loops and routines that are executed several times. In order to show this relation, we have analyzed the number of static and dynamic instructions of the benchmarks included in the Spec2006 suite [91]. The results are shown in Table 3.1. This table contains four columns: (i) the benchmark name, (ii) the input set identifier, (iii) the number of static instructions of the benchmark, and (iv) the number of dynamic instructions when the benchmark is executed using the input set specified in the second column. It is important to note that benchmarks with different input sets have different number of static instructions because we are counting static instructions that have been executed at least once. Instructions that are never executed are not considered and they are referred to as unused instructions. As shown in Table 3.1, the number of static instructions is in the order of thousands whereas the number of dynamic instructions is in the order of hundred billions for all benchmarks. These trends are similar for other well-known benchmarks such as the ones included in the Spec2000 suite, but in this case dynamic instructions are in the order of ten billions [92].

Since static instructions are executed multiples times, it is possible to dynamically analyze their behavior first, and then to optimize them in order to improve program execution. However, such a runtime analysis and optimization introduces an overhead that must be amortized by the benefits achieved by executing the optimized version of the code. In other words, if program execution is improved by X%, but the execution time devoted to analyze and optimize it is Y%, being Y greater than X, then the dynamic optimization is useless.

Benchmark	Input Set	Static Insts	Dynamic Insts
astar1	Ref - 1	14,251	4.39E+11
astar2	Ref - 2	14,241	8.84E+11
bwaves	Ref - 1	17,813	1.96E+12
bzip2	Ref - 1	14,546	4.28E+11
bzip2	Ref - 2	14,385	1.85E+11
bzip2	Ref - 3	14,392	3.12E+11
bzip2	Ref - 4	13,917	5.66E+11
bzip2	Ref - 5	13,849	6.00E+11
bzip2	Ref - 6	14,574	3.44E+11
gamess	Ref - 1	127,589	1.17E+12
gamess	Ref - 2	112,161	8.40E+11
gcc	Ref - 1	275,038	8.50E+10
gcc	Ref - 2	264,941	1.67E+11
gcc	Ref - 3	257,406	1.41E+11
gcc	Ref - 4	254,473	1.05E+11
gcc	Ref - 5	267,338	1.62E+11
gcc	Ref - 6	261,931	1.99E+11
gcc	Ref - 7	240,623	1.81E+11
gcc	Ref - 8	265,571	6.39E+10
gobmk	Ref - 1	114,872	2.60E+11
gobmk	Ref - 2	120,434	6.96E+11
gobmk	Ref - 3	110,028	3.73E+11
gobmk	Ref - 4	116,736	2.61E+11
gobmk	Ref - 5	115,636	3.76E+11
h264ref	Ref - 1	44,005	5.59E+11
h264ref	Ref - 2	53,595	3.88E+11
h264ref	Ref - 3	53,715	3.53E+12
hmmer	Ref - 1	19,069	9.75E+11
hmmer	Ref - 2	15,806	2.08E+12
lbm	Ref - 1	7,705	1.68E+12
leslie3d	Ref - 1	39,542	1.79E+12
libquantum	Ref - 1	9,093	4.01E+12
mcf	Ref - 1	8,969	3.76E+11
milc	Ref - 1	20,239	1.22E+12
perl	Ref - 1	101,718	1.26E+12
perl	Ref - 2	79,347	4.11E+11
perl	Ref - 3	88,378	7.73E+11
povray	Ref - 1	65,902	1.30E+12
soplex	Ref - 1	36,624	8.59E+11
specrand	Ref - 1	4,694	6.54E+08
sphinx3	Ref - 1	31,292	3.02E+12
tonto	Ref - 1	202,295	3.28E+12
wrf	Ref - 1	283,124	3.14E+12
zeusmp	Ref - 1	56,277	2.24E+12

Table 3.1: Static and dynamic instructions of some selected Spec2006 applications using ref input set.

Figure 3.2 shows the trade-offs between overhead costs and benefits that could be obtained by optimizing the code for the Spec2000Int benchmarks. In particular, the figure shows the relation between optimization overheads and code coverage. Coverage is defined as the ratio of the dynamic execution weights of a subset of static code instructions compared to the dynamic weight of the full program static instructions, being dynamic weight the number of times one static instruction is executed during the whole program run. Note that coverage is a ratio that is always lower or equal to 100% because the dynamic weight of any subset of static program instructions is lower or equal than the full program dynamic weight. The X axis shows the optimization threshold or minimum number of times a static instruction should execute before it is considered for optimization. As an example, if the threshold is 0, the subset corresponds to all the instructions that are executed more than 0 times, which, by definition, represents the whole static instructions (not considering unused). Such a subset always represents 100%

coverage. In the figure, the coverage starts to decrease when considering subsets of instructions with thresholds higher than 10,000 execution times. Moreover, it is important to remark that still considering a threshold of 1,000,000 execution times the coverage remains around 89%.

On the other hand, in the left Y-axis of Figure 3.2, we have the overhead costs of optimizing such subsets of considered instructions. The cost of the optimizer is computed as the increment of dynamic instructions required to optimize the static instructions in the subset. In other words, the overhead is the increment of dynamic instructions introduced by the optimizer in the system. Note that in the Figure 3.2 we consider 3 different costs for the optimizer. Each one represents a different type of optimizations, starting from very simple ones and ending with very aggressive ones. For example, we assume in the figure that a simple optimizer spends 10k dynamic instructions to optimize a single static instruction, while a very aggressive optimizer incurs in 300k dynamic instructions to optimize one single static instruction (these numbers are based on previous studies from the literature [36] [93] [94] [95] [96]). The overhead for simple optimizations is lower than for aggressive ones but the performance improvements expected by using them are also lower. From the figure, we can see that if we optimize all code in a very aggressive manner (extreme case, threshold of zero and overheads of 300k instructions per static instruction) we incur in an overhead of 5.6%. This means that the optimizations must provide more than 5.6% performance benefits to start improving the execution compared to executing the code non-optimized. If we move to instruction subsets with threshold higher than 10,000 execution times, then the overheads decrease to less than 1.5% still having a good coverage of 99.8%. The major inflexion occurs at threshold 100,000 where the coverage is at 99.0% and the overheads are less than 1.1%. After this point the coverage degrades very rapidly.

Figure 3.3 shows the same information but for Spec2006 benchmarks. In this case, the overheads are significantly lower because in these benchmarks the number of dynamic instructions is significantly higher than the number of static instructions, and therefore, overheads are more easily amortized. For this reason, when it is possible, in the rest of this document we will consider also the Spec2000 benchmarks suite where overheads are more difficult to amortize.

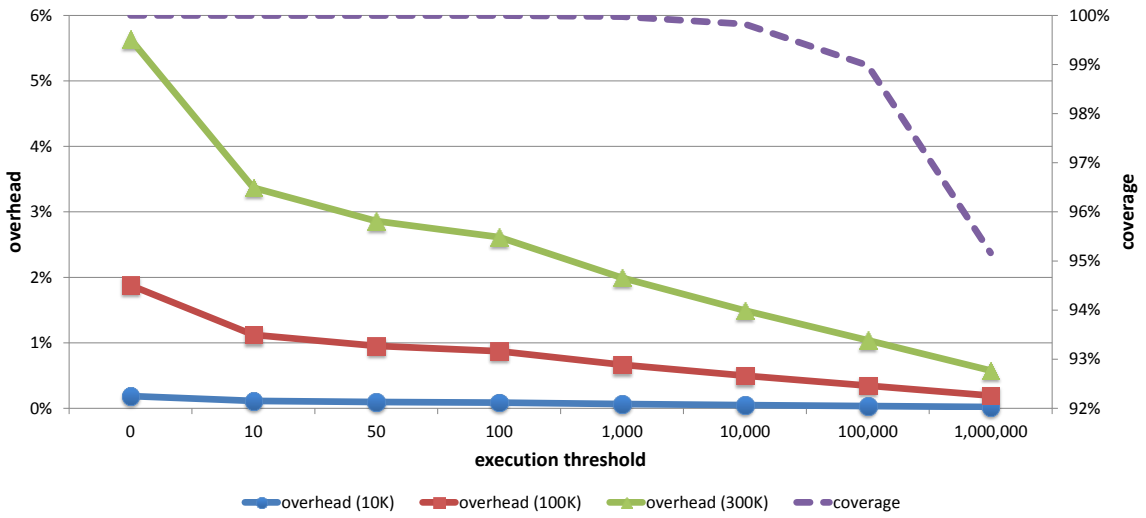


Figure 3.2: Coverage & overheads of optimizing different subsets of static instructions considering Spec2K Integer benchmarks.

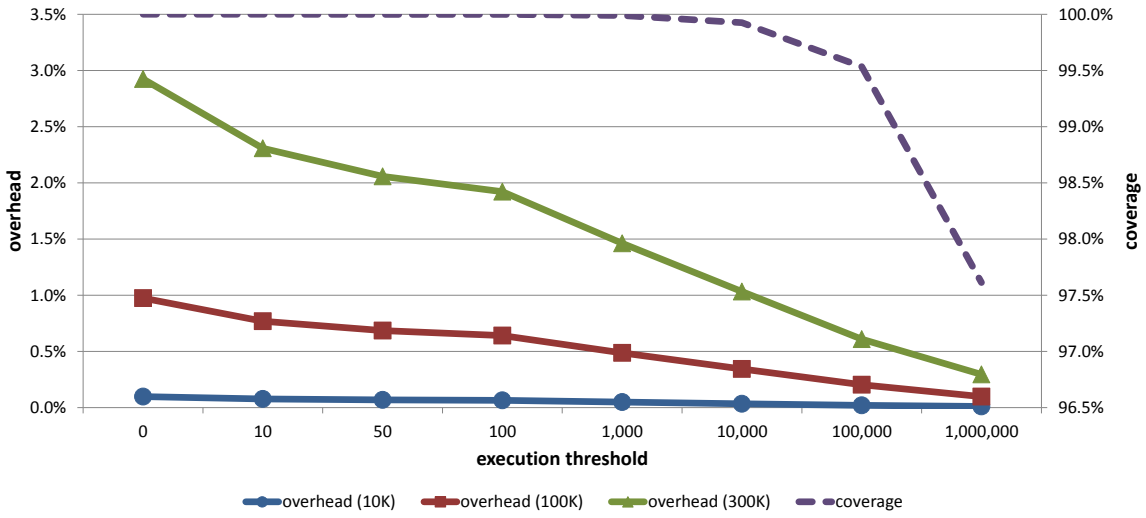


Figure 3.3: Coverage & overheads of optimizing different subsets of static instructions considering Spec2K6 benchmarks.

Note that Figure 3.2 and Figure 3.3 do not consider other overheads such as the program context switches costs (including the necessity of sharing resources through different applications) or the costs related to ensure correct execution of the optimized codes. The study of these overheads is out of the scope of this thesis since we are only focused on those related to detecting hot code.

Optimizing all the application code is not a good approach because of the overheads. Actually, only a small portion of the static code is responsible of the major part of the dynamic execution of the program as reflected in the coverage lines in Figure 3.2 and

Figure 3.3. By optimizing only these small static code regions, it is possible to reduce the cost of the optimization without compromising the possible performance improvements when executing them [18] [42]. Note also that detecting these regions is not trivial in a dynamic optimizer since they have to be found on time to take full advantage from their optimization. If a region is detected too late, then the optimization may be useless. Moreover, the time required for detecting regions suitable to be optimized compromises also the time required for making the optimization and the possible performance benefits that could be obtained in consequence.

3.2.3.2 Counting instructions

As we have seen, counting instructions for determining which code regions are of interest for optimization is important in a co-design profiler. Although in the coverage and overhead results presented before, we counted directly the number of dynamic instructions, this is not really required because we can use the number of executions of basic blocks as a proxy. This is so because all instructions within a basic block are always executed the same amount of times. Thus, counting basic blocks is equal to counting the executions of the instructions belonging to it. As described in Figure 3.4, a basic block starts on the instruction that is the target (or destination) of a previously executed branch instruction. In the figure, the "jump" instruction in BB1 has 2 destinations, one starts BB2 and the other starts BB5. In fact, the address of the instruction that is the destination of the branch identifies the basic block it belongs to. Thus, counting the number of executions of branch destinations is sufficient to count the executions of dynamic basic blocks. This instruction counting method is commonly known as basic block profiling [97][3].

The volume of information to be tracked by the profiler when counting branch targets (e.g. basic block starting addresses) is described in Figure 3.5 and Figure 3.6. The figures show the number of basic blocks in Spec2000 and Spec2006 benchmarks respectively. Comparing Figure 3.6 with the results from Table 3.1, we can observe that the basic block counting proxy reduces significantly the size of the information to be profiled for Spec2006 (similar trends apply for Spec2000). As a simple example, *bwaves* has 17,813 static instructions (from Table 3.1) spread in 2,456 basic blocks (7.25 times less information),

[3] It is also possible to count the transitions from one basic block to another, which is commonly called edge profiling [97]. This technique is more accurate but requires storing more information. We have selected basic block profiling because it is simpler for a hardware implementation.

and *gcc* has around 260,000 static instructions (depending on the input) in 75,210 basic blocks (2.46 times less information to be tracked).

Furthermore, there is a second bar in these figures that indicates the number of basic blocks that are executed more than 100,000 times. Note that this threshold is selected as the best candidate for reducing the optimizations overheads while keeping good application coverage, as described in Figure 3.2 and Figure 3.3. Thus, this bar shows the amount of information to be profiled when detecting the regions of code suitable for optimization. Taking a look at particular benchmarks, *lbm* has 7,705 instructions within 1,345 basic blocks, but just 47 of these basic blocks contribute to the majority of the dynamic execution of the benchmark. In case of *tonto*, the 202,295 static instructions are distributed in 20,583 basic blocks and the code that is suitable to be optimized is contained in 3,890 basic blocks.

In fact, benchmarks like gcc (in both suites), perl (in both suites), crafty, parser, gap, vortex, gobmk, h264ref, tonto, wrf and zeusmp contain more than 2,000 basic blocks even when filtering by using the threshold of 100,000. These numbers are still very big for a direct implementation of the profiler in hardware.

However, this analysis assumes that all instructions are executed evenly distributed during the time the application is running. In other words, variable "time" is not considered. In the next section, we describe how time may affect the way to design an efficient basic block profiler.

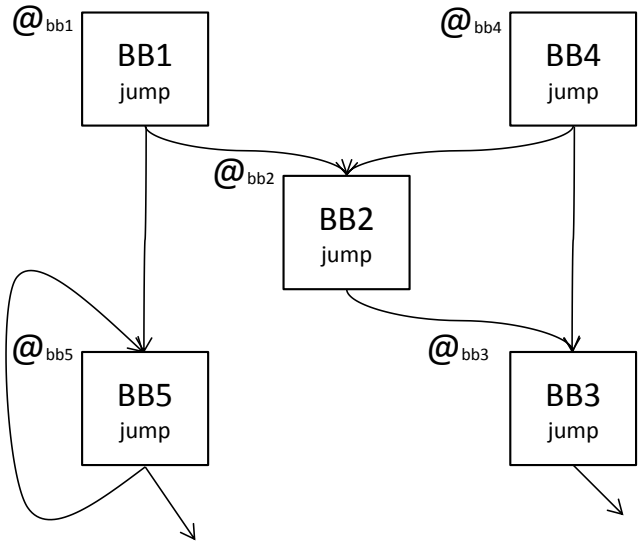


Figure 3.4: Basic Blocks execution example.

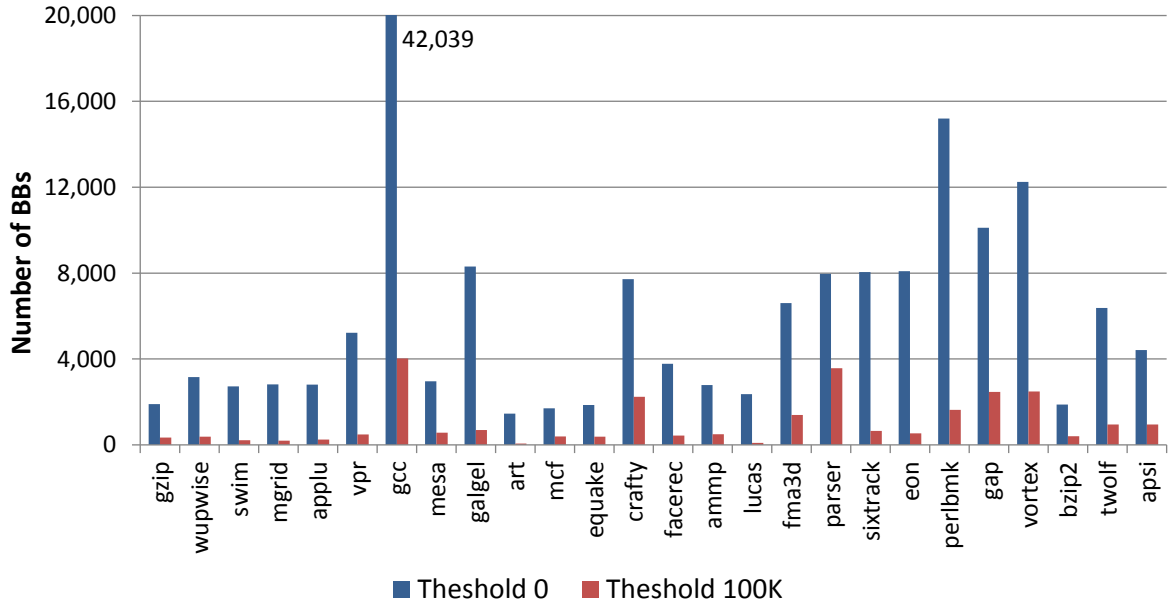


Figure 3.5: Number of basic blocks in Spec2000 benchmarks that are executed above a specific threshold.

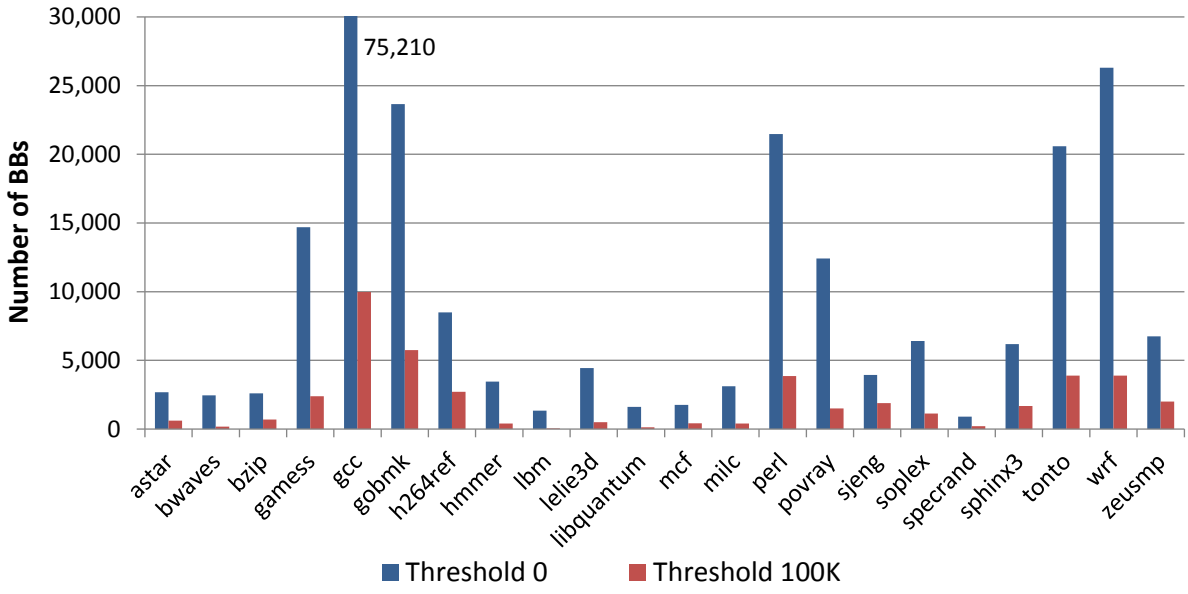


Figure 3.6: Number of basic blocks in Spec2006 benchmarksthat are executed above a specific threshold value.

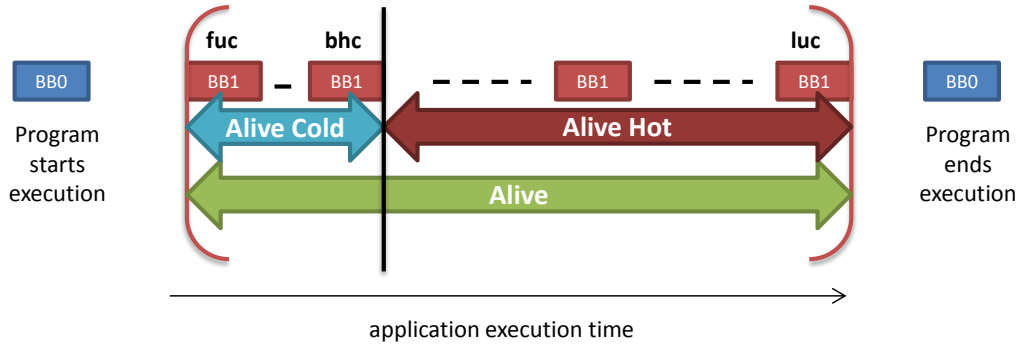


Figure 3.7: Basic Block Life Cycle during application execution.

3.2.4 Basic Block Cycle of Life

As we have seen, the big amount of basic blocks still imposes restrictions on the design of a hardware basic block profiler. However, the execution of basic blocks is not evenly distributed among time. Therefore, it is required to analyze the amount of information to be considered simultaneously during execution in order to delimit the information to be profiled. Actually, the amount of basic blocks executing simultaneously represents the size of the information the profiler requires to work over. In this section, we use an analysis model that describes how and when the basic blocks are executed. This model considers that the execution of the basic blocks passes through different stages defined as follows:

- First Used Cycle (fuc): processor clock cycle when the basic block is executed for the first time during program execution.
- Last Used Cycle (luc): processor clock cycle when the basic block is executed for the last time during program execution.
- Becomes Hot Cycle (bhc): processor clock cycle when the basic block is executed more times than a fixed threshold. In other words, the cycle when the basic block can be considered as part of hot code given a threshold.

These stages in the execution of a basic block conforms different periods of time as described in Figure 3.7. The period of time from *fuc* to *luc* is the total basic block execution time. We call this period the basic block *alive* time interval. The period of time from *fuc* to *bhc*, also called the instruction *alive cold* interval, determines the time the basic block requires to become hot. Note that if a basic block is not executed more times than the selected threshold value then this period completely matches the *alive* interval. Finally, the time from *bhc* to *luc* represents the time that the basic block is executed as hot code. This is the time that can be exploited by binary optimizers. We also refer to the process described in Figure 3.7 as the "Basic Block Life Cycle" for hot code detection.

During program execution, basic block life cycles for different basic blocks may start and end in different clock time cycles. Hence, the overlap of alive basic blocks determines the upper bound amount of information required by the profiler in order to detect the hot code regions. To be more precise, the *alive cold* interval is the one that delimits the amount of information required for the profiler. This concept is described in detail in the example in Figure 3.8 where 4 basic blocks with their corresponding life cycle time periods are shown. A dummy brute force profiler would require counting all basic blocks. In this example, there are four basic blocks. However, basic blocks 1, 2 and 3 overlap their life cycles, whereas basic block 4 is executed once the others have finalized. Thus, the maximum alive overlap is 3 basic blocks. A more sophisticated profiler would require counting only three instead of four basic blocks. In addition, if we consider only the *alive cold* intervals, then only basic blocks 2 and 3 overlap their *alive cold* interval (time to become hot). In this case the space problem is reduced to just 2 basic blocks to be profiled simultaneously.

Following the same methodology described in the example of Figure 3.8, we have executed all Spec2000Int and Spec2006 benchmarks in order to analyze the overlap of the different "Basic Block Life Cycles" in these applications. However, we are not interested exclusively in the maximum overlap value. Actually, we are interested on the evolution of the basic block execution because we want to analyze the more frequent values of overlapped life periods, since maximum values may not be the common case. As an example, it is possible to have the maximum overlap during a very short period of time whereas such an overlap is much lower in the rest of the execution. Thus, we have split the execution of all benchmark in regular intervals of time. The overlap is then computed within each one of these intervals. In this sense, a basic block that is *alive* during all program execution it will appear in all intervals. In case a basic block is only *alive* during a fixed interval, then it is only counted within this interval.

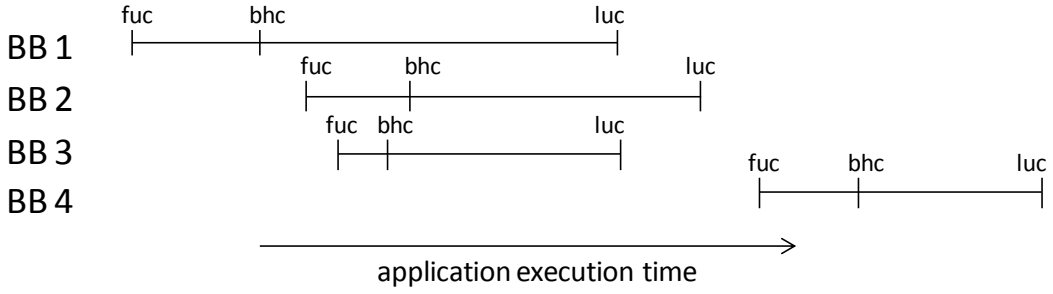
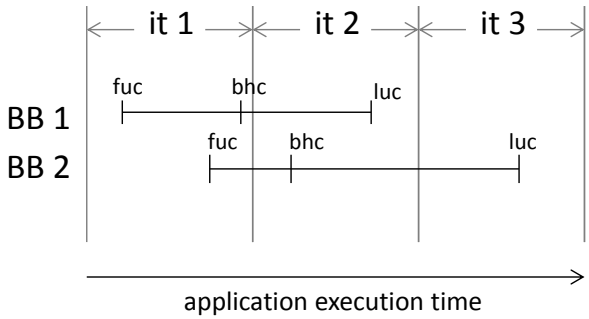


Figure 3.8: Basic block life cycles during a program execution example.



Overlap computation

	it1	it2	it3
Alive BBs	BB1 BB2	BB1 BB2	BB2
COUNT	2	2	1

Figure 3.9: Life Cycle overlap computation example.

Figure 3.9 shows a description on how to make the life cycle overlap computation. In this example there are two basic block with different life cycles represented in the left part of the figure. The program execution time has been divided into 3 fixed intervals labeled as it1, it2, and it3. Both basic blocks are *alive* during the first two intervals, so that the number of overlapped *alive* basic blocks for intervals 1 and 2 is 2. Basic block 1 is executed for the last time during interval 2, leaving basic block 2 as the only *alive* one during interval 3. The table in the right part of Figure 3.9 summarizes the overlap computation, showing that the more common case is to have 2 *alive* basic blocks simultaneously during program execution.

Since Spec2000Int and Spec2006 are very large in terms of dynamic instructions (see Table 3.1), the execution has been divided into fixed intervals of 100 million dynamic instructions. This number has been selected because it is more or less in the same order of magnitude of an Operating System (OS) context switch period as described in Section 3.2.5 of this same chapter. All benchmarks have been executed from beginning to end, and we have counted the number of basic blocks in their *alive* and *alive cold* periods for each one of the intervals. Moreover, we have also differentiated among basic blocks that at the end become hot and those that do not. In order to show the results, we have built a histogram that describes how the different basic block life cycle periods overlap during Spec2000Int and Spec2006 benchmarks execution. We have used the number of basic blocks whose life cycle periods overlap as discrete values in order to generate these histograms. For the histogram frequencies (x-axis), we have chosen the number of intervals on which basic blocks life cycle overlap is counted. Thus, these histograms show the number of intervals where a fixed number of basic blocks overlap their life cycle periods. Since the number of basic blocks is too large for plotting such a histogram, they have been discretized considering the following values: 0, from 0 to 512, from 512 to 1024, from 1024 to 2048, from 2048 to 4096, from 4096 to 8192, from 8192 to 16384, and finally

more than 16384 basic blocks. Note also that the number of intervals for each application differs significantly because of the differences in the number of total dynamic instructions of each one. Therefore, the histograms have been built considering that all applications are executed one after the other as if they were part of a bigger program. The basic block life cycle overlap study has been done for each one of the intervals and finally, we have harmonized the number of intervals considering the total number of intervals in all applications. As showed in Figure 3.10 and Figure 3.11, this allows us to compute the number of intervals as the percentage of intervals where a fixed number of basic block overlap their life cycle periods for Spec2006 and Spec2000 respectively. In particular, we show four different lines identifying the different stages basic blocks pass through:

- The line with square marks represents the number of application intervals where there are *alive* basic blocks (from first used cycle to last used cycle) simultaneously.
- The line with rhombus marks represents the number of intervals where there are basic blocks in *alive cold* stage (from first used cycle to become hot cycle) simultaneously.
- The line with circle marks represents the amount of intervals where there are basic blocks in *alive hot* stage simultaneously.
- The line with triangle marks represents the amount of intervals where there are basic blocks in *alive cold* that end up become hot.

For example, from Figure 3.11, around 80% of the time intervals have at most 16k or less *alive cold* basic blocks simultaneously (line with rhombus marks). In fact, from the same figure, there are no intervals with less than 1k *alive cold* basic blocks simultaneously (line with rhombus marks). However, if we concentrate on basic blocks that end up becoming hot, the situation is much better. For instance, 80% of the intervals have 512 *alive cold* basic blocks simultaneously that become hot at most (line with triangle marks). The same trends are observed in Figure 3.10, where 100% of the intervals have 512 *alive cold* basic blocks simultaneously that become hot at most (line with triangle marks).

Two important conclusions are extracted from the histograms in Figure 3.10 and Figure 3.11. On one hand, the number of intervals where basic blocks are *alive* is significantly high, even when considering only basic blocks that become hot. This number is interesting because it shows that hot basic blocks persist during long periods of time, so that it is possible to improve the performance of the application by optimizing them. Secondly, the number of basic blocks with overlapped *alive cold* periods is small and ranges between

4096 to 8192 basic blocks. Note that *alive cold* period is the one that is of interest for detecting hot code. With an appropriate mechanism able to filter those basic blocks that never become hot, the amount of information to be considered may be reduced to a number around 512 basic blocks, which is an appealing size for efficiently implementing the hardware component of a profiler for hot code detection. Although we have demonstrated that the amount of information to be profiled is limited and affordable, we still require the mechanism to correctly handle this information.

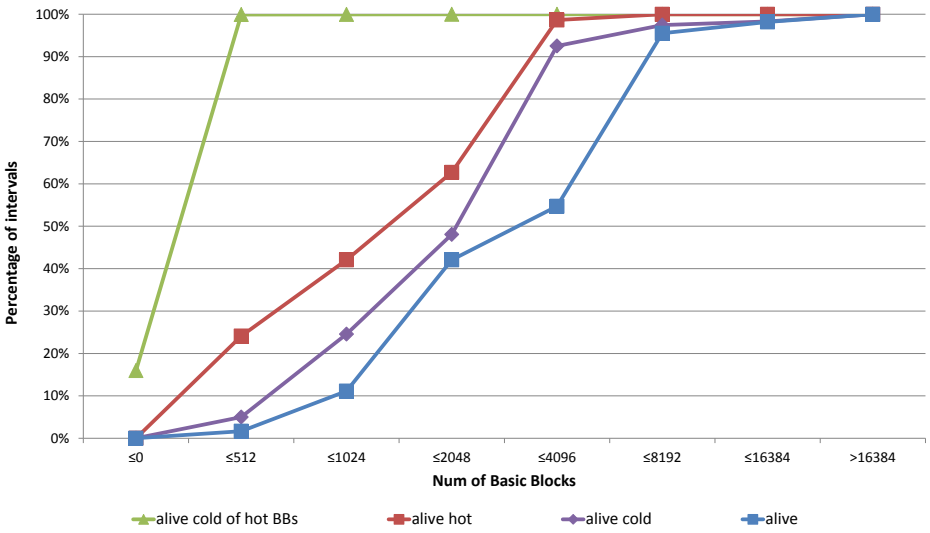


Figure 3.10: Percentage of intervals with the same number of basic block life cycle periods overlapping for Spec2000Int benchmarks.

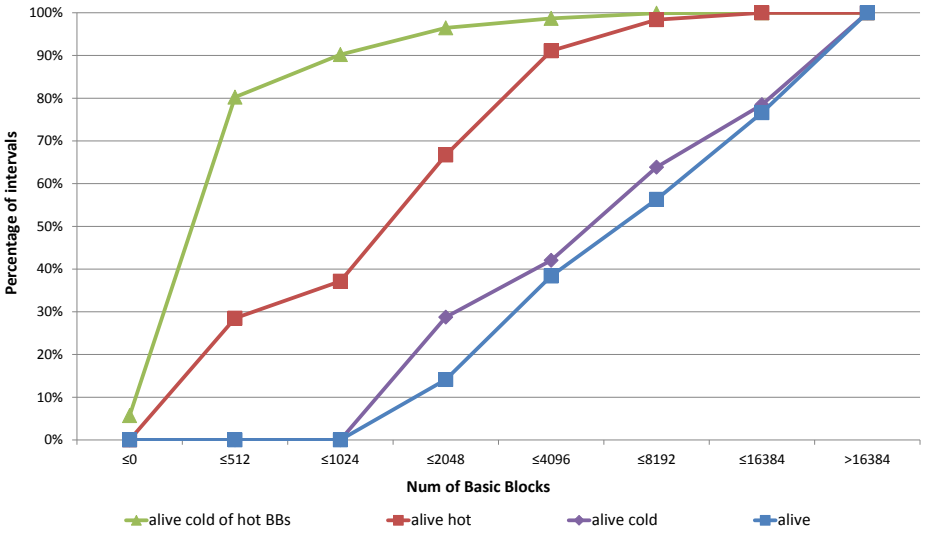


Figure 3.11: Percentage of intervals with the same number of basic block life cycle periods overlapping for Spec2006 benchmarks.

3.2.5 Context Switch

In all previous studies, we have considered that applications execute completely from the beginning to the end without external interferences. However, in a real multitasking environment the Operating System (OS) changes periodically the execution from one application to another. This is called time sharing and the process to save and restore the state (context) of the CPU is known as a program context switch.

The OS does a context switch at fixed intervals of time or when the application blocks it execution (e.g. it executes an input/output operation, it waits for a semaphore, etc.). The fixed time between context switches is called quantum and it is normally set to 100ms [98]. Thus, assuming an IPC of 1 instruction per cycle, a processor frequency of 1 Ghz and a OS quantum time of 100ms, then a context switch occurs every 100 million instructions. For this reason, we have used intervals comprising 100 million instructions for all our evaluations.

Context switches impact the way the hot code profiler handles the information about the execution of basic blocks, since the hardware component is shared among all processes. When a context switch occurs, we have identified three different design alternatives to be considered:

- To reset the counters on every context switch, losing the accumulated information.
- To store all information in a dedicated buffer that later will be used for recovering the previous state of the information. This means that the contents of the hardware profiler become part of the CPU state to save/restore on context switches.
- To add an identifier to each counter in order to differentiate among counters belonging to different applications. This option keeps information about all basic blocks but at the cost of increasing the data space for accommodating entries of other different applications.

Because we do not want to increase context switch overheads and we do not want to increase hardware resources by profiling several (may be tenths) applications simultaneously, we have chosen to reset the counters on context switches. In next section we demonstrate that resetting affects coverage, and hence this decision may imply lowering the detection threshold to maintain coverage. Fortunately, our proposal adapts better to this scenario than other already existing solutions since it detects basic block of higher quality in a faster manner, as we demonstrate in Section 3.4.2.5.

3.2.6 Reset Counters Coverage Cost

Resetting counters on every program context switch may delay the detection of hot code because the counters may never reach the threshold before they are set back to zero. Taking this problem to the extreme, this could imply not detecting hot code whatsoever. This means that coverage may be probably lower than when resetting is not considered. In addition, this coverage loss may translate directly to a reduction of the benefits that could be obtained by optimizing the detected hot code.

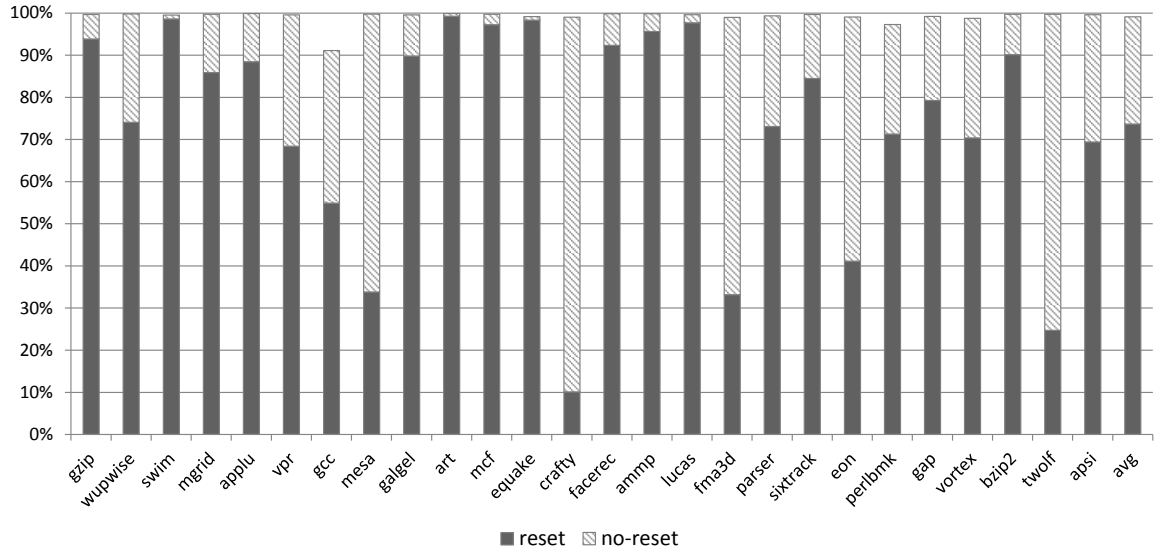


Figure 3.12: Basic block coverage for SPEC2000 benchmarks when applying counters reset.

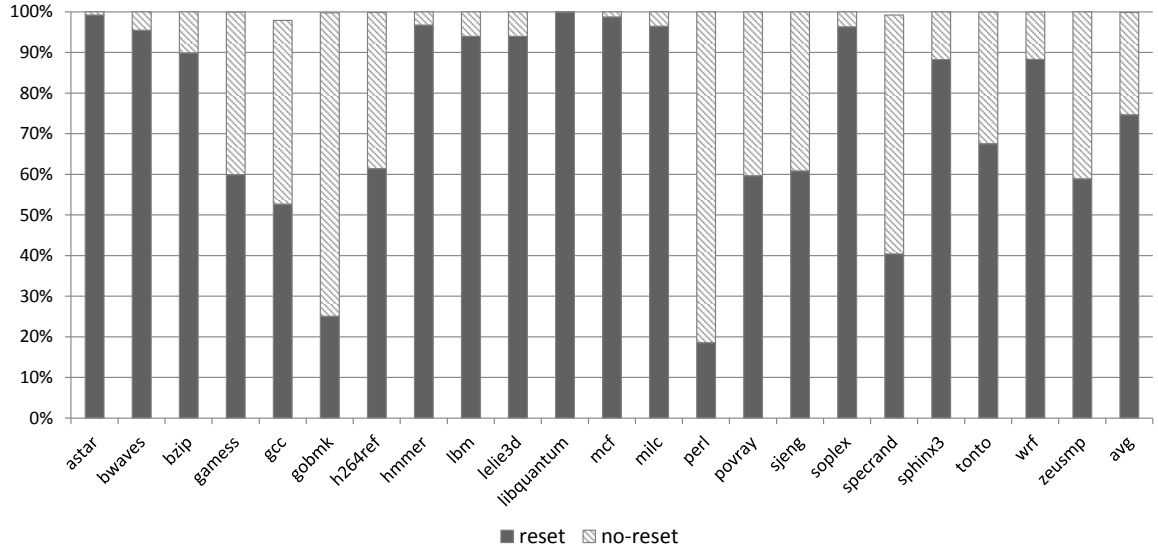


Figure 3.13: Basic block coverage for SPEC2006 benchmarks when applying counters reset.

We have compared the coverage of the hot basic blocks when enabling and disabling the profiler counter reset technique on every context switch. The numbers are shown in Figure 3.12 and Figure 3.13 for Spec2000 and Spec2006 benchmarks respectively. In this experiment, coverage has been computed as the number of dynamically executed hot basic blocks compared to the total number of dynamically executed basic blocks in the application. Note that the number of executions of a hot basic block is counted from the moment it becomes hot, discarding the required executions for its detection. As previously presented in the Section 3.2.3, counting basic block executions is a proxy to counting instruction executions, so coverage without resetting showed in Figure 3.12 and Figure 3.13 practically matches the previously reported coverage of 98% for Spec2000 and 99% for Spec2006 showed in Figure 3.2 and Figure 3.3 respectively.

As it can be observed, the coverage degrades significantly for some applications when reset is applied. Actually, only 8 benchmarks for Spec2000 (gzip, swim, art, mcf, equake, facerec, ammp and lucas) still present coverage around 90%, whereas 5 benchmarks (mesa, crafty, fma3d, eon and twolf) are impacted significantly by the reset since the coverage decrease under 50%. On average, the coverage decreases from 99% to 74% for Spec2000 when reset is applied. There is a similar scenario for Spec2006 benchmarks, where 10 benchmarks are severely impacted by the reset (games, gcc, gobmk, h264ref, perl, povray, sjeng, specrand, tonto and zeusmp) whereas 9 applications (astar, bwaves, hmmer, lbm, leslie3d, libquantum, mcf, milc, sjeng) still present coverage above 90%. On average, coverage degrades from almost 100% when no reset is applied to 75% when it is applied for Spec2006.

The coverage loss is due to the aforementioned resetting counters problem: hot blocks are detected late or are never detected. In order to analyze this in more detail, we have studied the impact of resetting on the time required for detecting hot basic blocks, and on the time the basic block is executed as hot code. Actually, these two concepts can be summarized by using the following formula:

$$useful\_time\_ratio = \frac{luc\text{-}bhc}{luc\text{-}fuc}$$

where *fuc* is the basic block first used cycle, *luc* is the basic block last used cycle and *bhc* is the cycle the basic block becomes hot, all three belonging to the basic block life cycle definition presented in Section 3.2.4. This formula quantifies the relation between the time the basic block is executed as hot code (the numerator) and the time it is alive (the denominator). In an ideal hot code detector, all basic blocks would have a

useful_time_ratio of 100%, since *luc-bhc* time period would match up with *luc-fuc* period. By contrast, a *useful_time_ratio* of 0% implies that the hot basic block is never detected as hot.

We use the Spec2006 Specrand benchmark to analyze the impact of the reset technique on the basic blocks *useful_time_ratios*. Note that Specrand presents a degradation of 60% coverage as it is shown in Figure 3.13. Figure 3.14 shows the number of executions and the *useful_time_ratios* of all the basic blocks detected as hot in the Specrand benchmark. Basic blocks are sorted from left to right by number of occurrences. The square line is associated to the left y-axis and it shows the number of executions of the basic blocks in times they exceed the threshold. The triangle and circle lines are associated to the right y-axis and show the *useful_time_ratios* of the basic blocks. Circle line considers that basic block execution counters are reset on every interval of 100 million instructions (equivalent to a context switch), whereas triangle line does not. On the other hand, Figure 3.15 shows the same information as Figure 3.14 but for the basic blocks that are not detected as hot when resetting the counters. Note that in this case, the *useful_time_ratios* are only for the approach of no resetting the counters.

These two figures help us to better understand how the basic blocks are executed during the time the application is running. In Figure 3.14, we identify in the left part basic blocks that are fast detected (high *useful_time_ratios*) and that contribute widely to the coverage. By contrast, in the middle part of the figure, we identify basic blocks that are not fast detected and suffer from coverage degradation when a counter reset is applied. These basic blocks contribute significantly to the coverage but its detection is difficult because they are not intensively executed to exceed the threshold during an interval. In the right part of the figure and in Figure 3.15, we observed the basic blocks that contribute less to the coverage. The firsts do not present problems on detection since *useful_time_ratios* are equal no matter if reset is applied or not, but the ones from Figure 3.15 are not detected although they exceed the threshold.

Basic blocks can be classified attending to these findings, as it is presented in Section 3.2.7. Moreover, this classification allows us to properly define the characteristics of the basic blocks that the profiler should keep track of. Note that the same basic block behavior found in Specrand also appears in the other Spec benchmarks.

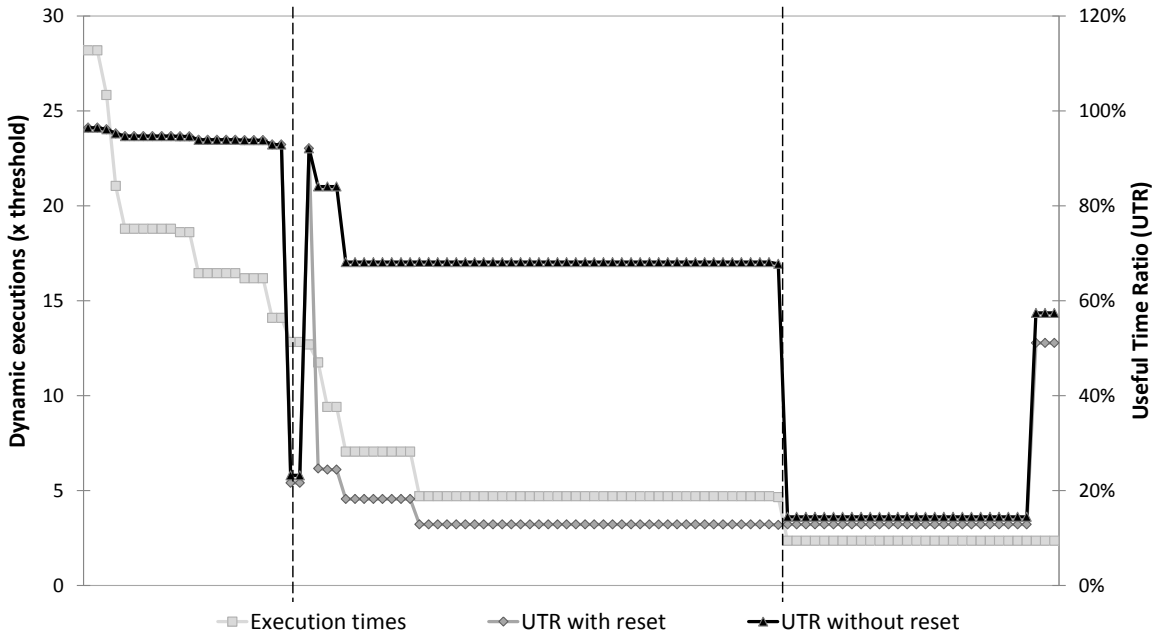


Figure 3.14: Coverage and useful time ratio for detected hot basic blocks in Spec2006 Specrand benchmark when profiler reset technique is applied.

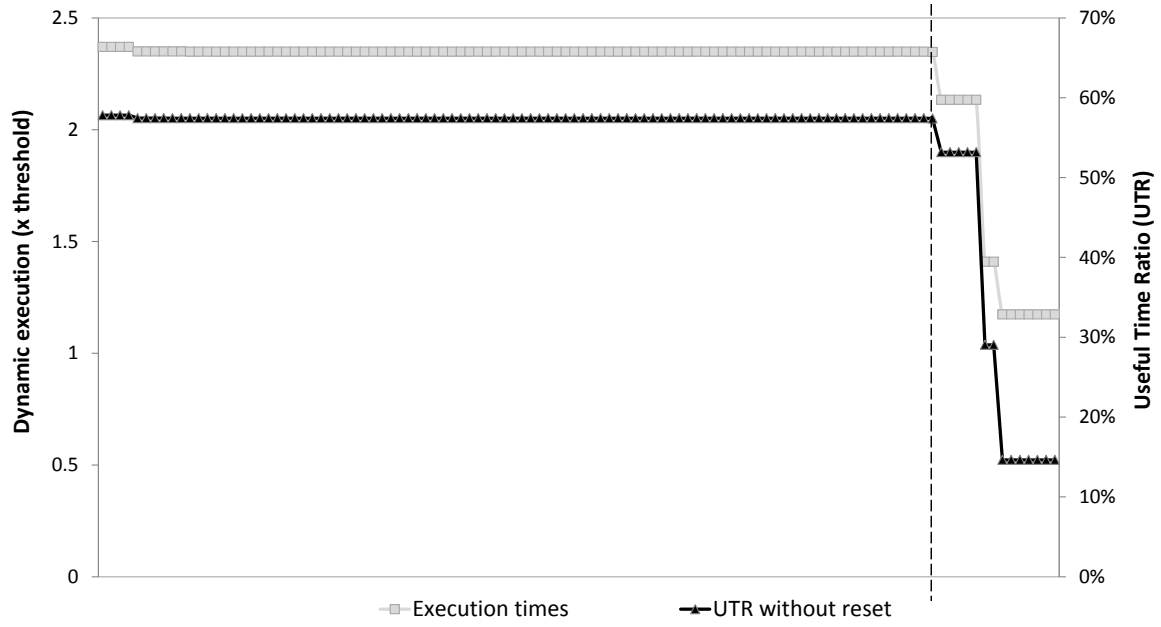


Figure 3.15: Coverage and useful_time_ratio for non-detected hot basic blocks in Spec2006 Specrand benchmark when profiling reset technique is applied.

3.2.7 Basic Block Classification

Basic blocks with a high number of executions and also with high *useful_time_ratios* are the ones that contribute more to the final application coverage. However, both concepts are directly dependent of the threshold value and the size of the profiler reset intervals. In fact, the threshold indicates the amount of executions to be detected as hot, and the reset interval indicates the time the basic blocks have to accomplish this. Actually, based on how and when basic blocks exceed the threshold during the application execution, we have classified them as described below. For illustrate purposes, each type of basic block has been defined with an animal name in order to identify its execution frequency with the animal speed. The classification is as follows:

Thoroughbred Horse type: basic blocks that are highly executed and have large *useful_time_ratios* even when reset is applied. These basic blocks are intensively executed during large periods of time. They are easily detected because they exceed the threshold during the first intervals they are executed. In the example of Figure 3.14, they are located at the left part and can be identified because their *useful_time_ratios* are close to 100% even when reset is applied. As the representative animal, we have selected the thoroughbred horse because of its high speed and its tremendous resistance.

Athletic Man type: this type includes the basic blocks that are highly executed but with high reductions in their *useful_time_ratios* when the reset is applied. This type of basic blocks is less intensively executed than the previous one and it exceeds the threshold but requiring more time. An example of these basic blocks can be seen in the Figure 3.14, located in the middle of the figure and showing high degradation in their *useful_time_ratios* when reset is applied. They are like an athletic man running a marathon, since their speed and resistance are very high but are not enough to compete with a thoroughbred horse.

Cheetah type: this type includes the basic blocks that are infrequently executed and have small *useful_time_ratios.* These instructions do not contribute much to the total coverage because they are executed in very intensive bursts but they are rarely executed after such bursts. In fact, these basic blocks are detected as hot because the number of executions exceeds the threshold during their first bursts. These basic blocks are located at the right part of Figure 3.14. We call them as Cheetah basic blocks because these animals are very fast but only in small distances when required for hunting. After they have successfully hunted a prey, they don't require running again.

Mule type: this type includes the basic blocks that are infrequently executed and have large *useful_time_ratios* when reset is not applied but that are not detected when reset is applied. These basic blocks are constantly executed during the application. They never exceed the threshold but at the end of the application they account for a significant part of the coverage. They are like a Mule, walking at medium speed during long distances. As an example, they can be seen in the Figure 3.15 located at the left part.

Turtle type: this type includes the basic blocks that are infrequently executed, have small *useful_time_ratios* when reset is not applied and are not detected when reset is applied. These basic blocks contribute poorly to the application coverage and it is difficult to get benefits by optimizing them because of its low execution and small *useful_time_ratio.* They are like turtles moving very slowly. In Figure 3.15, they are located at the right part.

This classification is graphically summarized in Figure 3.16. The figure presents a fictitious application with 5 different basic blocks, each one being one line in the figure and belonging to a different type. The x-axis shows the application execution time divided in reset intervals while the y-axis shows the number of executions of the different basic blocks. A threshold of 10 has been used in this example. As it is shown, the thoroughbred horse type is always above the threshold, whereas the athletic man type presents difficulties to exceed it. The cheetah type appears two times during the execution of the application and it exceeds the threshold easily in both periods; later such a basic block is not executed anymore. The mule type is very constant but never exceeds the threshold, whereas the turtle type appears and disappears but with very low execution frequency.

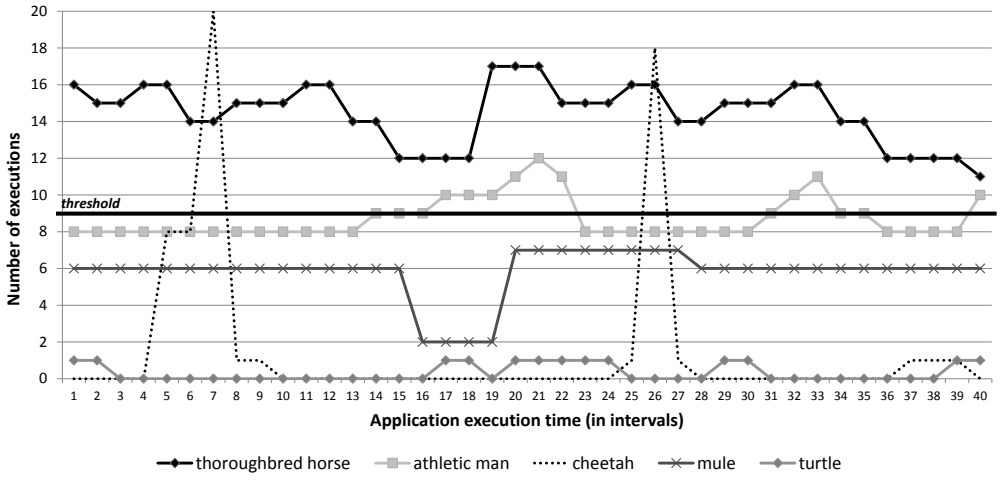


Figure 3.16: Basic Block execution Classification by using animal speeds.

The hardware profiler can take advantage from this classification in order to prioritize the detection of those basic blocks that at the end contribute positively to the coverage. Furthermore, it can give less priority to those basic blocks that are not contributing significantly to the coverage and also interfere negatively in the detection of the others. Therefore, thoroughbred horse basic blocks are the most appealing ones because of its high coverage contribution and easy detection. After them, athletic man and mule basic blocks are the next important ones. In these cases, however, their execution frequency imposes difficulties for their detection. Cheetah basic blocks may also contribute significantly to the coverage if they exceed comfortably the threshold or if they appear multiple times. Finally, turtle basic blocks contribute poorly to the coverage, so this type is the less interesting for detection. As a summary, hot basic blocks that contribute significantly to coverage always satisfy at least one of the three following conditions:

- High and constant execution frequency (thoroughbred horse and athletic man).
- High execution frequency on short intervals (cheetah).
- Low but constant execution frequency (mule).

All the studies done so far in this chapter have helped us to characterize the data information space that the hot code detector should work with and how this information evolves during program execution in order to detect it properly. In the next section, we propose a solution that takes advantages from all these find-outs and that it is able to identify the basic blocks that are the most frequently executed efficiently during application execution.

3.3 Proposed Solution (LIU)

Hybrid hardware and software hot code detectors combine the flexibility of software profilers with the low execution overheads of the hardware ones. However, hardware resources required in these detectors are still complex and big [18]. In this section, we propose a hybrid software/hardware hot code detector that requires significantly less hardware resources than other hybrid solutions. It has been designed by following the ideas described in this chapter about basic block execution behavior.

3.3.1 Hardware Structure

The proposed technique combines hardware and software solutions for detecting the hot code. A hardware table tracks the number of basic block executions during the time the application is running on the processor. Once one basic block is executed more times than the fixed detection threshold, an exception is raised and the control is transferred to the

software layer. From this point on, the software is responsible of building and optimizing the code regions that will be executed later in the processor.

The profiler hardware table is typically implemented like a small set-associative cache. Previous designs tend to overestimate the size of this structure in order to accommodate as many entries as possible. In our previous results, we have demonstrated that the size of the table could be equal or lower than 512 entries for tracking correctly hot basic blocks execution (Figure 3.10 and Figure 3.11). However, the risk of losing information increases when using a finite structure because of conflicts derived from mapping branch targets to table entries. These conflicts appear when a new branch destination address is added into the profiler and it requires evicting an existing entry (possibly with useful information). These conflicts may affect the detection of hot code. In fact, it is not possible to know exactly what basic blocks will become hot and are the more promising ones to be kept in the structure in case of conflicts. One way to efficiently use the limited number of entries of the cache structure (the table) is to use an appropriate replacement policy designed to maximize the presence of the most promising basic blocks in terms of coverage, not wasting resources for the less promising ones. In other words, it should efficiently use the limited hardware storage resources. Therefore, the replacement policy is the heart of the proposed hot code detector. We will describe the proposed replacement policy in more detail in Section 3.3.4.

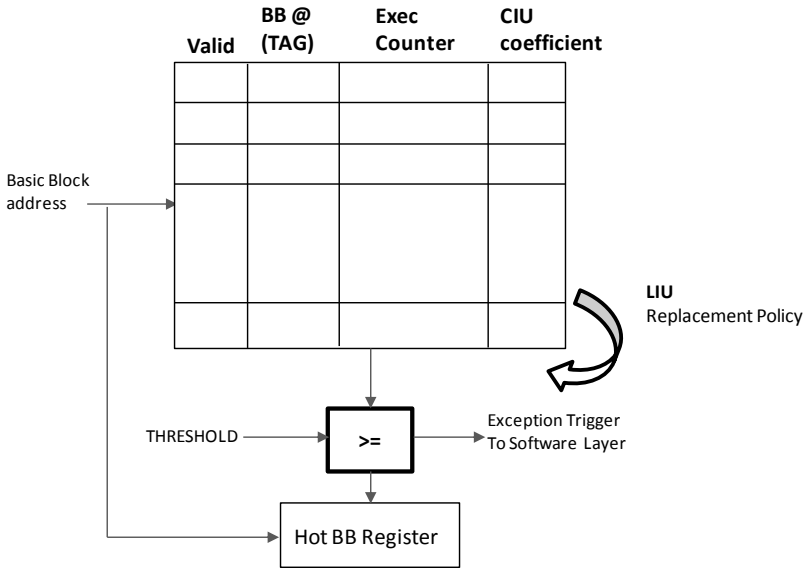


Figure 3.17: LIU Profiler hardware design.

The hardware implementation of the profiling table is shown in Figure 3.17. Each table entry contains some bits of the address as a tag, an associated execution counter, a valid bit, and a CIU coefficient field for the replacement policy (more on this in Section 3.3.4). In order to index it, we use the logical address of the instruction that is executed after executing a control instruction (a branch, a routine call, or a routine return) either if the instruction changes the control flow or not (e.g. regardless of whether they are taken or not taken). This profiled information corresponds to the starting address of a basic block. As in any other conventional cache, some bits from the address have to be selected in order to index the set of the structure we want to access. In Section 3.4.2.1, we analyze different indexing proposals with the objective of selecting the one that maximizes the coverage. When the hardware table is accessed (a basic block starting point is executed) and there is no entry in the structure associated to it, one entry in the table is allocated, evicting the previous entry if necessary and setting the initial number of executions to 1. By contrast, if the basic block has been previously allocated in the table, then its execution counter value is just incremented by 1. The execution counter is compared with the constant threshold detection value on every access. If the number of executions is greater or equal to the threshold, then the exception that communicates to the software layer that the profiler has found a new hot code region is raised. The starting address of the detected basic block is stored in a special register (hot BB register) for the software to access. Note that the threshold value for detection also affects the coverage of the detected basic blocks. A high threshold implies lower number of basic blocks that exceed it and lower coverage in consequence. In the proposed implementation, we consider a threshold of 100K executions, since the coverage is greater than 90% for most of the Spec applications and the cost of the optimizations is affordable, as previously commented in Section 3.2.3. Finally, the proposed hardware table structure is placed at the retirement stage of the processor. This removes the profiler from any processor critical path.

3.3.2 Software Support

Once an entry of the hardware table exceeds the threshold it is considered as hot. An exception is raised, the detected hot address is selected in the spatial hot basic block region, and the dynamic optimizer takes control of the processor execution. The dynamic optimizer then implements proper heuristics to build code regions and apply optimizations based on this profiling information. For instance, the optimizer can build a super-block [41] or a tree-region [99] starting at the specified address and by including reachable basic blocks where executions counters are big enough. Once it has created, optimized, and stored the region in the code cache [13], it may decide to invalidate the entry associated

with the starting address in order to provide space for later basic blocks (and it may also invalidate those of the rest of basic blocks included in the region). Note that an entry is considered as invalid when the valid bit is set to 0.

In this thesis, we are focused on the detection of the code regions that are suitable to be optimized by a dynamic optimizer. The formation and the optimization of the detected regions are not evaluated since this is out of the scope of this study.

3.3.3 Replacement Policy Motivation

The replacement policy should always try to keep the basic blocks that will contribute the most to coverage in the hardware table (see coverage definition in Section 3.2.3). These important basic blocks follow a predictable execution behavior as it has been highlighted in the classification of Section 3.2.7. Actually, this classification is based on the number of basic block executions and how recent they were executed. However, as we will show in the evaluation section, traditional replacement policies that use the number and recency of uses in a separated manner are not appropriate for detecting hot code. As described in Table 3.2 and confirmed in the evaluation section, not all the basic blocks that contribute significantly to the coverage will be detected if the hot code detector uses one of these traditional replacement policies. On one hand, replacement policies based on recency are not able to retain the basic blocks that execute slowly but constantly, since they are fast evicted from the table by younger and more recent basic blocks. In fact, basic blocks with mule behavior are not protected by such a replacement policy. On the other hand, replacement policies based on number of uses do not retain the basic blocks that execute fast on short time intervals because older basic blocks with higher number of executions are prioritized. This replacement policy do not protect basic block with cheetah behavior. Fortunately, both types of replacement policies can be combined in order to ensure the detection of the three types of basic blocks that contribute most to coverage.

Type of Hot Basic Blocks	RP based on the number of uses	RP based on the recency of uses
High and **constant** execution frequency *(horse and athletic man)*	Detected	Detected
High execution frequency on **short intervals** *(cheetah)*	X	Detected
Low and **constant** execution frequency *(mule)*	Detected	X

Table 3.2: Traditional replacement policies used for hot code detection (RP stands for Replacement Policy).

Table 3.2 summarizes how the replacement policies based on number and recency of uses adapt to detect hot basic block. The first column shows the type of basic blocks that contribute more to the coverage of the application. The second and the third columns indicate if the replacement policies are able to detect the different types of basic blocks. We can observe that when the replacement policy based on number of uses fails, the one based on recency succeeds, and vice versa.

There are some examples of replacement policies in the literature combining number and recency of accesses. The most relevant proposals are the IRP [100] and the LRFU [101]. However, they have not been designed for tracking basic block execution behavior. In fact, they have been developed to be used in scenarios like software web caching and page management in operating systems and they have been specially designed to be purely implemented in software. For this reason, they employ complex algorithms to compute the best victim candidate that are not adequate for a direct hardware implementation.

IRP has two important drawbacks to be directly incorporated into the aforementioned hardware table. First, it requires a computational costly software algorithm for selecting the best victim candidate. Second, it needs to refresh periodically the cache on every processor clock in order to compute the age (that determines its recency) of each entry. On the other hand, LRFU also uses a complex mathematical expression for selecting the victim that is hard to be implemented in hardware. In particular, it computes a coefficient for every way in the cache and it selects the one with the minimum value as eviction candidate. The main complexity of this coefficient comes from two factors: (1) the expression is based on a variable that keeps the history of all previous accesses (which means that every time a way is accessed this history factor must be recomputed), and (2) the expression requires floating point division and exponential operations.

We propose a new replacement policy that combines frequency and recency aimed to detect hot basic blocks, avoiding the complexity of previous proposals. In next section, we describe in deep detail how this replacement policy, called LIU, works.

3.3.4 LIU Definition

The proposed LIU scheme combines the philosophies behind two well-known replacement policies: the Least Recently Used (LRU) and the Least Frequently Used (LFU). The former is based on the recency of the accesses, and the second is based on the number of executions. In fact, the LIU replacement policy works by observing the recent uses of basic blocks and by taking into account also their number of executions. LIU name comes from

Least Intensively Used, where intensity refers to the relation between the number and the recency of the uses[4].

LIU selects a victim entry from the hardware profiler cache structure based on its elapsed time since it was last accessed and the number of times it has been used. This relation between frequency and recency of the accesses is numerically described by using a mathematical coefficient stored in each entry of the table. Since the coefficient combines the age and the number of accesses, we call it as Coefficient of Intensively Used (CIU). In particular, for a given cache entry *i,* it is computed by using the following expression:

$$\mathrm{CIU}_i = \frac{\mathrm{T} - \mathrm{t}_i}{\mathrm{C}_i}$$

In the above expression, T is the current absolute time, $\mathrm{t_i}$ is the last absolute time entry *i* was accessed and $\mathrm{C_i}$ is the number of times entry *i* has been accessed since it was added to the table. The entry with the highest CIU value in the affected cache set is selected as the replacement policy victim. When a basic block is evicted from the table, the information in the entry is lost. The corresponding entry counter is set to 1 when a basic block is reallocated again in the table. Note that we could also use the total number of accesses instead of time, but we will use time for the rest of this document for the sake of simplicity.

The LIU proposal gives priority to basic blocks that are constantly executed during long periods of time and to those that are very frequently executed in short periods of time. Note that the formers also include basic blocks that are highly and constantly executed during all the time the application is running because they derive in several short but frequent execution periods. The selection of which basic blocks are kept in the table is controlled by the CIU coefficient. As described in Table 3.3, a low CIU value for an entry indicates a basic block that is frequently executed or that it has been recently accessed. By contrast, a high CIU value indicates that the associated basic block is infrequently accessed or it has not been used for a long period of time.

Since the CIU value does not require historical information, it is only computed when eviction is required (a given basic block is not found in the accessed set and there are no invalid entries in the set). However, the time of the last access ($\mathrm{t_i}$) and the number of executions ($\mathrm{C_i}$) for a given entry have to be updated on every access to that entry.

[4] The intensity measures the force with which an event occurs.

CIU_i Value	*It means*	*It implies*
Low	Frequently or very recently used entry	**Candidate to be kept**
High	Infrequently or not used for a long time entry	**Candidate to be evicted**

Table 3.3: CIU meaning.

3.3.5 pLIU: A Realistic LIU Implementation

LIU requires a relatively complex expression that can be simplified in order to achieve reasonable implementation complexity. We call pLIU (pseudo-LIU) to this simplified version of the original replacement policy. pLIU assumes a set associative table with its corresponding valid bit, tag, and execution counters fields per entry and the following additional information.

- Global time counter (T), incremented on every processor cycle.
- t_i (last access time) field in each entry of the cache. It is set with the current global counter value every time its corresponding entry is accessed.
- Logic to compute the CIU value when an eviction is required.

Since the division operation within the CIU expression requires complex hardware, we propose a simplification of the expression consisting on a division by a power of two. This can be implemented by a less costly shift operation. In fact, since the division denominator is the value of the counter (C_i), the simplification basically consists on truncating this value. This operation can be done in a very efficient manner by using a priority encoder and a decoder [102] [103]. The priority encoder returns the position of the highest input signal set to 1 and the decoder transforms this result into a binary number with only the indicated position set to 1. For instance, if the value of the counter is 00101101, then the priority encoder returns 101=5 as the position of the most significant bit set to 1, from this result the decoder generates 00100000=2^5 as the final value. The encoder and the decoder can be seen as a unique hardware component in our proposal, so that for the rest of the document we will refer to this logic as the encoder. Therefore, the proposed simplification transforms the CIU expression formula as follows:

$$\mathrm{pCIU}_i = \frac{\mathrm{T} - \mathrm{t}_i}{2^{\lfloor \log_2 C_i \rfloor}}$$

In the formula above, T is the global time counter, t_i is the time counter stored in each entry, and C_i is the number of executions of the entry. We call to this coefficient as the pseudo-Coefficient of Intensively Used (pCIU).

Figure 3.18 describes the pLIU implementation for a particular profiler cache with 4 ways per set and 16-bit counters. The extension to other configurations can be done directly by using this as a baseline. The pLIU is computed for each way/entry in the set. The T value is sent to 4 adders, one per way, in order to compute the division numerator *(T-t_i)* part of the formula. Moreover, the C_i value is truncated by using the encoder. The result from (T-t_i) is then shifted by the number of times indicated by the encoder. This encode and truncate logic is graphically described in the figure by the DIV dotted grey box. Finally, the resulting coefficient of each way is sent to a MAX component that computes which is the maximum value in order to select the victim. From a timing point of view, the coefficient computation affects only the hardware profiler eviction time since coefficients do not need to be computed in case of hit accesses to the structure.

The profiler table entries are 32 bits wide, with 16 bits for representing t_i and 16 bits for C_i. The detection of a hot entry is done by just checking when a C_i counter overflows. Therefore, no comparators involving the threshold value are required. However, this solution requires scaling down the detection threshold from 100k to 65k executions (16 bits for the threshold counter) because a power of 2 value is required[5]. We have implemented the pLIU proposal in an FPGA on top of a processor similar to the Intel® P54C [104]. The P54C is a 2-way superscalar in-order core with separated instruction and data caches of 8KB each one. A high-level block diagram of the processor with the pLIU hardware additions is shown in Figure 3.19. The pLIU gets the address of a basic block from the Target Address wires once the branch instruction is completely executed. Therefore, the proposed hardware structure can be placed at the retirement stage of the processor, not affecting any critical path of the processor and avoiding the profiling of speculative code instructions.

When using a 128 entry profiler table (32 sets and 4 ways), the total processor area is only incremented by 1%. Note that the P54C is a small in-order core. Thus, adding this profiling hardware to more complex cores would have a negligible impact on area and leakage.

[5] Using 17 bits for the threshold (131k executions) is less appealing than using 16 bits because the ratio coverage/optimization-overheads is clearly in favor for the second (see section 3.4.2.5). The reason is that higher threshold values than 100k executions degrade very rapidly the coverage.

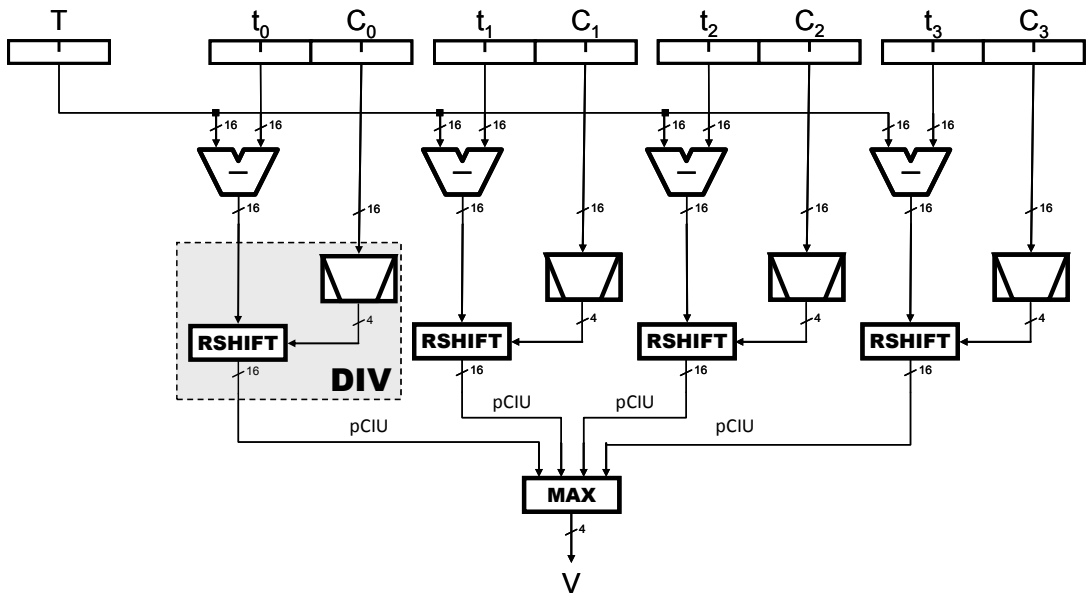


Figure 3.18: Hardware implementation of the LIU. V is the victim.

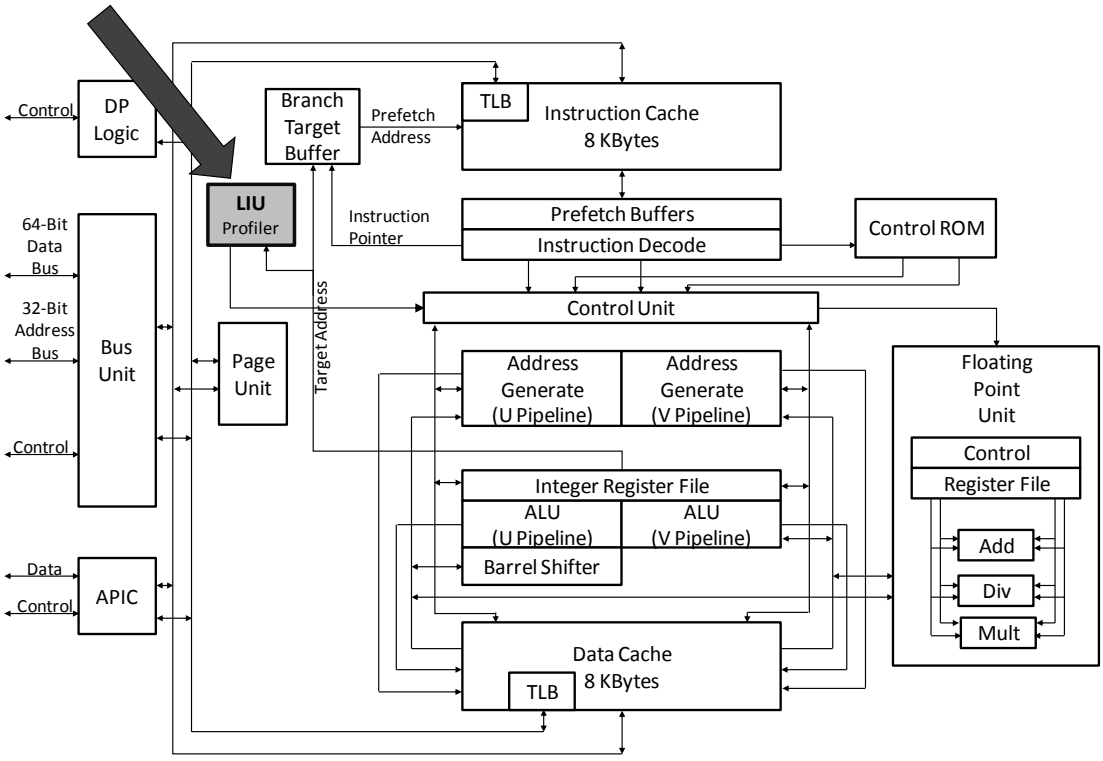


Figure 3.19: High-level block diagram of the P54C core with the pLIU profiler (highlighted with the arrow). Figure from P54C Datasheet [105].

3.4 Evaluation

3.4.1 Simulation Framework

In order to evaluate the quality of the proposed hot code detector, we use the execution coverage metric presented in previous Section 3.2.3.1. This metric indicates the ratio between hot dynamic basic blocks and the total dynamic basic blocks of the application. A high coverage value indicates that most of the application execution could be improved by applying code optimizations. By contrast, a low coverage indicates that fewer basic blocks can be optimized and the benefits that could be obtained are lower in consequence.

The simulation framework has been implemented by using PIN [76]. PIN is a software tool designed to dynamically instrument the execution of applications. It provides special functions to track the execution of instructions, allowing access to the register identifiers and memory addresses among other instruction properties. We have developed a fully customizable PIN tool that simulates the behavior of a hot code profiler based on a set-associative cache for storing the information. This tool allows multiple configurations of thresholds, cache sizes (including number of sets and ways), and replacement policies.

We have employed the Spec2000Int benchmarks for the sensitivity studies required to determine the best hardware profiler configuration. The Spec2006 benchmarks have very large execution times under our simulation infrastructure, making unaffordable running all sensitivity experiments among them. However, we demonstrated in our previous characterizations in Section 3.2 that Spec2006 should behave similarly to Spec2000 benchmarks. Therefore, we use Spec2006 benchmarks for the final evaluation, once we have selected the most appropriate configuration in our sensitivity studies.

In all the studies, we have used the reference input sets for both Spec2000Int and Spec2006 benchmark suites. For Spec2000Int we have 12 benchmarks that result in 33 executions when counting for the different input sets, and for Spec2006 we have 9 benchmarks that result in 30 executions.

Finally, we have estimated the power consumption of the LIU profiler by a model based on the CACTI access and time model for on-chip caches [106].

3.4.2 Results

In this section, we present the evaluation results of the hot code detector. First, we evaluate the performance of different cache indexing methods. Then, we analyze the performance of the LIU replacement policy compared to well-known replacement policies.

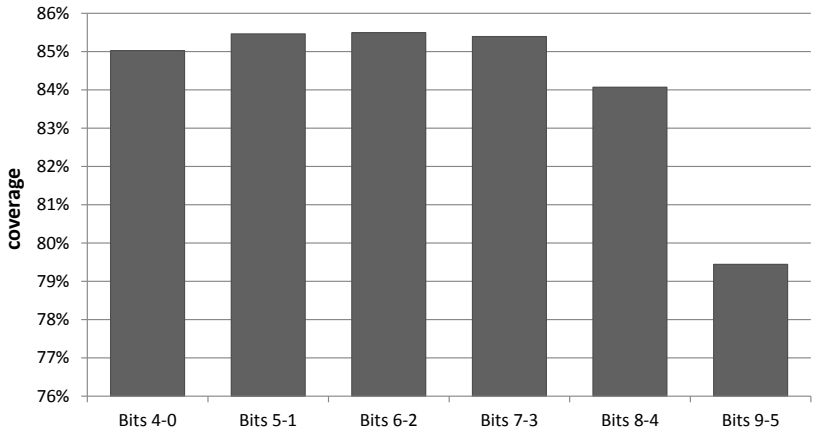


Figure 3.20: Hot code coverage evaluated in a 32sets-4way cache profiler by using different cache indexing bits from instruction address.

Later, we analyze how the different types of basic blocks contribute to the coverage when employing recency and frequency based replacement policies. We continue by comparing the LIU replacement policy against its simplified version, and we analyze the overheads and the performance improvements that could be obtained by implementing a co-designed hardware/software system with a hot code detector based on the LIU proposal. Finally, we conclude this section by analyzing the power requirements of the LIU profiler.

3.4.2.1 Indexing Bits

In order to index the profiler cache, we use the address of the destination of a branch (the instruction that is executed after a branch, either if it is taken or not taken). In other words, we use the first address of the basic block. As in any other conventional set-associative cache, we need to select some bits from the address or a combination of them to index the set we want to access. It is necessary to select the best subset of address bits in order to avoid unnecessary conflicts that may end in wrong basic block execution tracking (note that two basic blocks sharing the same entry may have totally different executions life cycles). Therefore, we have evaluated different simple indexing alternatives and we have selected the one that maximizes the coverage. As previously described in the Section 3.2.3, the coverage is computed as the accumulated number of executions of detected hot basic blocks divided by the accumulated frequency of all basic blocks in the application. In order to compute this metric, we have considered that once a basic block is executed more times than the given threshold, it does not make use of the counters again because it has already been detected as hot. Hence, this basic block produces no interference with other basic blocks that are profiled from that point on. Moreover, if the basic block is optimized it will be placed at a different memory address. The profiler

configuration employed for this study consists on a 32 sets and 4 way associative cache structure with the LIU replacement policy.

The simple evaluated schemes use the necessary n bits after ignoring the m lowest bits of the address. Since the selected cache structure configuration contains 32 sets, we need 5 bits from the address in order to index it. We have evaluated 5 alternatives: $bits_{4\text{-}0}$, $bits_{5\text{-}1}$, $bits_{6\text{-}2}$, $bits_{7\text{-}3}$, $bits_{8\text{-}4}$, and $bits_{9\text{-}5}$. Results are shown in Figure 3.20.

The alternative with more benefit is the $bits_{6\text{-}2}$ with 85.49% coverage, whereas the alternative with lower coverage is the $bits_{9\text{-}5}$ with 79.44% coverage. The remaining alternatives present similar coverage numbers than the $bits_{6\text{-}2}$ one. This study highlights the importance of using the correct address bits to index the profiler in order to avoid conflicts. In fact, the probability of having different basic blocks attached to the same profiler entry increases if the less significant bits from the address are not considered for indexing the cache.

3.4.2.2 Replacement Policies Evaluation

In this section, we evaluate the performance of well-known replacement policies when employed for hot code detection by using the coverage metric. In Section 3.2.4, we have demonstrated that a profiler with less than 512 entries should be enough for detecting hot code if an efficient mechanism for handling the execution of the basic blocks is employed. Thus, the cache configurations we have selected are the following ones: 16sets-2way (32 entries), 16sets-4way (64 entries), 32sets-4way (128 entries), 64sets-4way (256 entries), 128sets-4way (512 entries), and 128sets-8way (1024 entries).

The LIU proposal has been compared with five different replacement policies. All of them have been implemented into the profiling scheme presented in Section 3.3.1. Since LIU combines number and recency of the accesses, we have evaluated how these two factors work in a separate manner by taking into account the LRU and LFU traditional replacement policies. We have included a random technique that selects victims blindly and a perfect technique, called oracle, that it is able to select the best victim candidate at any time. This oracle replacement policy is an unrealistic technique that has the information of the application in advance prior to its execution and we use it as an upper-bound. Finally, we have considered the LRFU proposal since it has been also designed taking into account the recency and the frequency of the accesses [101].

The coverage evaluation for the SPEC2000Int programs is shown in Figure 3.21. All of the replacement policies achieve more than 80% coverage when employed in the 128sets-8way

cache configuration. However, when reducing the size of the cache, the coverage achieved by the LRU, the LFU, the LFRU, and the random policies degrades rapidly. This does not happen for the LIU proposal. In fact, the LIU replacement policy achieves good coverage even when the table has a small 16sets-4way configuration. In the 32sets-4way cache configuration, the LIU outperforms by 2x the LRU and the random policies, and by ~1.5x the LFU and the LFRU policies. At this same cache configuration, the LIU achieves only 10% less coverage than the ideal oracle. In fact, the LIU proposal is pretty close to the oracle from this point on, guaranteeing 90% coverage when more than 128 profiler entries are considered, with minimum differences between using 512-entry and 1024–entry configurations. The figure also shows that policies based on frequency perform better than those based on time. As it is shown, the LFU presents better results than the LRU replacement policy in all cache configurations. The main reason is that mule type basic blocks, not detected by time based replacement policies, contribute more to the coverage than cheetah type basic blocks, not detected by frequency based replacement policies.

The low results, with respect to the proposed solution, of the traditional replacement policies (LRU, LFU, random) when limiting the number of profiler entries highlight the importance of selecting an appropriate replacement policy based on basic block life-time execution. On the other hand, the LRFU policy, that combines time and frequency, is also far from the LIU proposal. In fact, LFRU performs similar to the traditional LFU. Note that the parameter configuration proposed in [101] and used in this study, configures LFRU to behave more like LFU than LRU[6]. However, this proposal has not been designed to track program code activity and, moreover, its hardware implementation would be more complex than the LIU one.

The best design point for the hardware profiler table is the 32-sets and 4-way configuration because it maximizes the coverage while keeping a reduced number of entries. In Figure 3.22, we show the evaluation of LIU, LRU, and LFU replacement policies by using the Spec2006 benchmarks for this configuration. In this case, LIU gets 83.3% coverage whereas LRU gets 48% and LFU 68.7%. These results are similar to the ones reported for the Spec2000Int benchmarks and highlight again the importance of combining properly recency and frequency of the accesses.

[6] We have used a LFRU value of 0.00005. This value is suggested by the LFRU authors to maximize the cache hit-ratio.

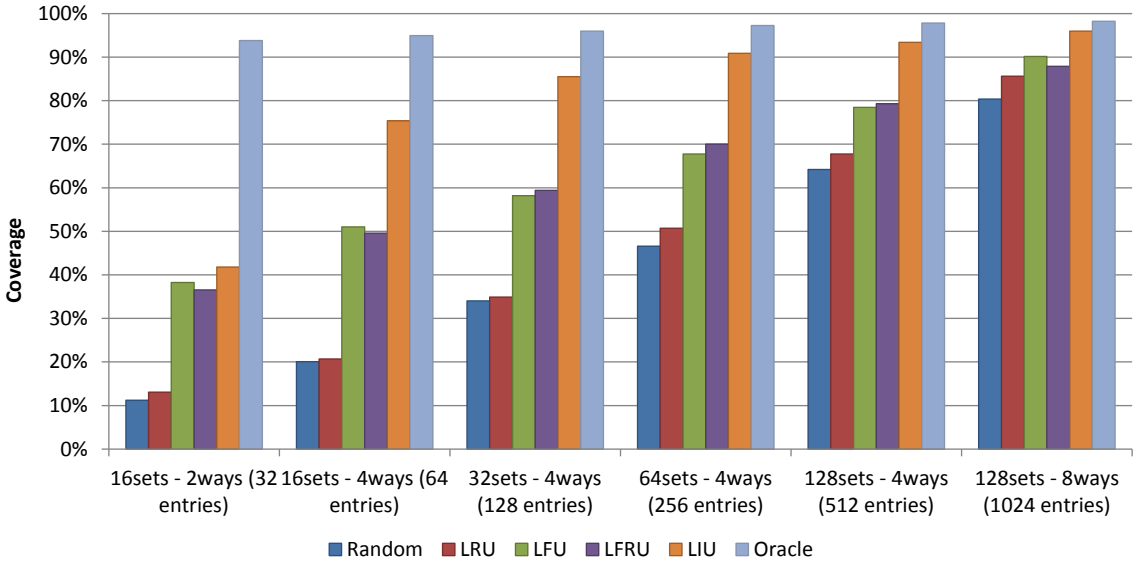


Figure 3.21: Hot code coverage evaluated by using different cache configurations and replacement policies.

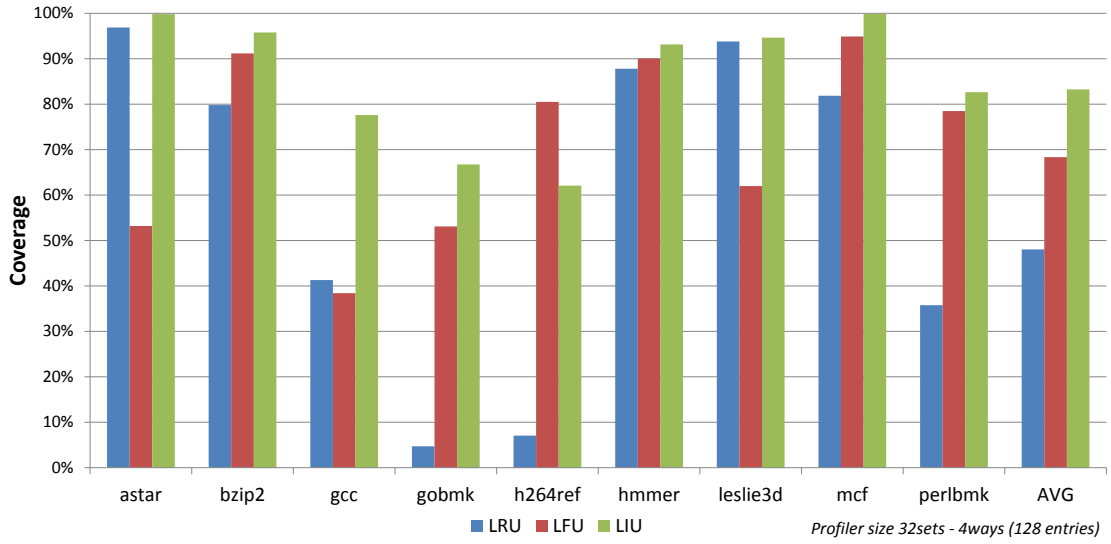


Figure 3.22: Hot code coverage evaluated by using LIU, LRU and LFU replacement policies and Spec2006 benchmarks.

3.4.2.3 Basic Block Characterization Evaluation

We have analyzed the contribution to the coverage of the different types of basic blocks for the LRU, LFU, and LIU replacement policies. For the sake of simplicity, we only show the results for mcf, perlbmk and vpr benchmarks from the Spec2000 suite. However, we have observed similar trends for other benchmarks. Mcf is an example of a benchmark that performs very well in all three replacement policies scenarios, perlbmk is an example

where LRU performs better than LFU, and vpr with the place_log input set is an example in which LFU performs better than LRU.

Figure 3.23 shows the dynamic coverage of the different types of basic blocks that are detected when employing LRU, LFU, and LIU replacement policies. Note that we only show cheetah, mule, athletic man (from now on just man), and thoroughbred horse (from now on just horse) types because turtle basic blocks do not contribute significantly to coverage. As it can be observed, man and horse types of basic blocks are the ones that contribute more to the coverage. In the case of mcf, the different replacement policies are able to detect them reaching coverage closer to 99%. In perlbmk, LRU is able to detect more man and horse basic blocks than LFU but less than LIU. By contrast, LFU and LIU detect more mules. In this case, man and horse accesses occur in burst which makes LRU and LIU more efficient than LFU. Finally, vpr is partially dominated by mule type basic blocks as shown in the LIU coverage results. In this case, LFU does better than LRU, although it cannot reach the numbers obtained by LIU. In this case, LFU policy is able to track them but conflicts with man and horse basic blocks delay their detection with respect to LIU. On the other hand, LRU makes it impossible to track this type of basic blocks correctly since it prioritizes cheetah-like behaviors.

Although man and horse types are the most important ones for coverage, mule type also contributes to it non-negligibly.

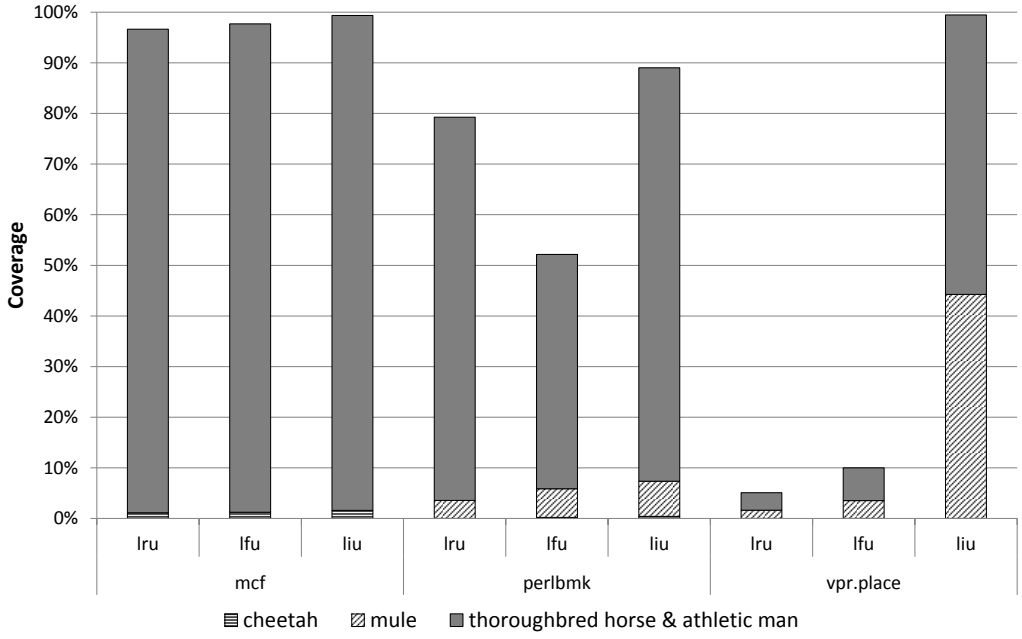


Figure 3.23: Coverage of the different types of basic blocks in mcf, perlbmk and vpr with place input set benchmarks.

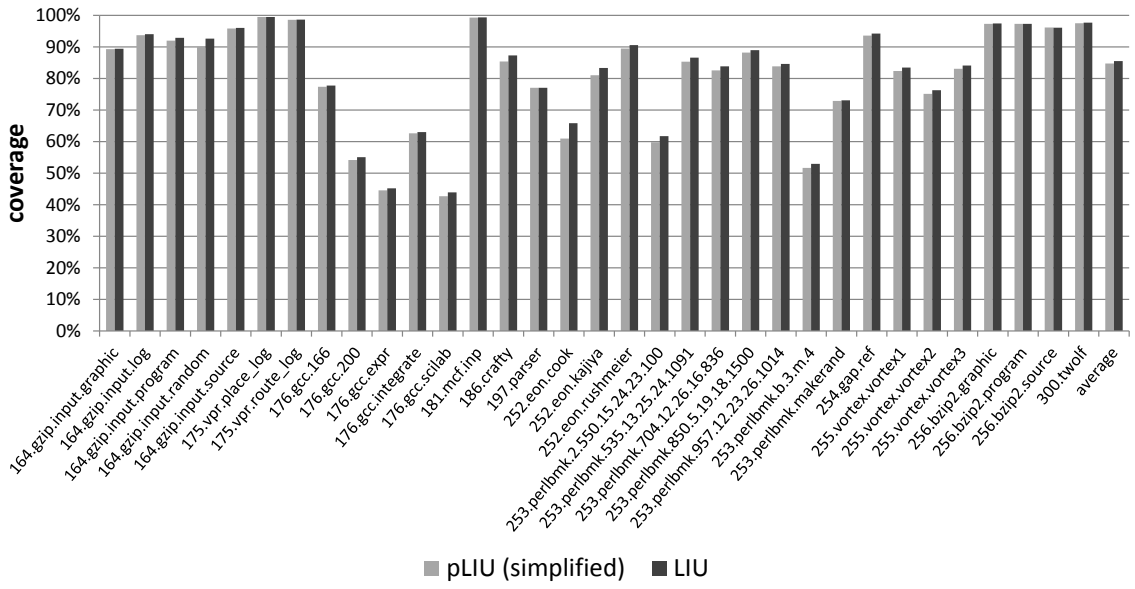


Figure 3.24: Dynamic code coverage evaluation for SPEC2000Int benchmarks.

3.4.2.4 pLIU Evaluation

We have also evaluated the performance of the pLIU proposal (the hardware simplified version of the LIU proposal). As in previous sections, this evaluation has been done by analyzing the coverage that could be obtained when the replacement policy is implemented on top of the profiler cache. In this case, we have compared the pLIU proposal with the original LIU by using all Spec2000Int benchmarks.

Figure 3.24 shows the coverage for the SPEC2000Int benchmarks and all the ref input sets when using a profiler table of 32 sets and 4 ways (chosen configuration). As it is shown, both original and simplified LIU proposals perform similarly. The complete LIU proposal gets on average 85.49% coverage whereas the simplified proposal gets 86.46%. The higher coverage of the pLIU is due to the lower threshold of 65k executions. In this case, pLIU detects 13,6k static basic blocks as hot whereas LIU detects 12,2k. This extra 10% detected basic blocks only contribute with an extra 1% coverage. The higher differences in favor of the pLIU proposal are found in the gcc, parser, and gzip with the "random" input set, obtaining 2.47%, 4.31%, and 6.33% better coverage respectively. By contrast, the higher differences in favor of LIU are found in eon with the "cook" input set where LIU gets 3.78% more coverage than pLIU. In the rest of the benchmarks, the differences are always below 2%.

Therefore, we can say that pLIU proposal presents similar coverage numbers than the original LIU and, although it may detect some extra basic blocks with low coverage

contribution, it can be implemented requiring less hardware resources because it does not require the complex division computation for the CIU coefficient.

3.4.2.5 Performance & Overheads

In this section, we evaluate the benefits in terms of performance that could be obtained by implementing a dynamic binary optimizer using different profiling schemes. The purpose of this experiment is to show the importance of using correct profiling schemes in order to balance optimizer overheads and performance. For this reason, we use a simple analytical model that considers coverage, optimizer overheads, and performance benefits and we compare the results against a baseline system that does not use dynamic binary optimization.

For the analysis, we have categorized the optimizations to be applied to hot code as simple, medium, aggressive, and very-aggressive. The cost (a.k.a. overhead) and the benefit for each one of the optimizations is totally different. The simpler the optimizations, the lower the cost of applying them and the lower the benefits obtained. Table 3.4 shows the considered group of optimizations (first column), the overhead that they introduce (second column), and the benefits that could be obtained by executing the resultant optimized code (third column). Note that the overhead is measured as the number of additional instructions to be executed per instruction to be optimized. These parameters are consistent with previous work [36] [93] [94] [95] [96]. Examples of simple optimizations include dead code removal, constant propagation, simple loop unrolling, etc., whereas example of more complex optimizations may include control and data speculation optimizations, complex loop transformations, module scheduling, instruction fusion, etc. (see Section 2.5.1 for more details).

For the sake of simplicity, the model does not assume a staged compilation optimizer in which hot code that is detected for the first time is optimized with less aggressive optimizations than code that is detected as hot for the second or third time (super-hot). In addition, we do not account for replication in the analytical model, where a particular guest instruction may be included in more than one optimized region.

Optimizations	Overhead (insts per inst to optimize)	Perf. Improvement
Simple	1,000 instructions	10%
Medium	10,000 instructions	20%
Aggressive	100,000 instructions	40%
Very-aggressive	300,000 instructions	60%

Table 3.4: Optimizations overheads and performance improvements.

The total overhead introduced in the system when optimizing hot code regions is computed by using the following formula:

$$overhead = NSHI \times Cost_{opt}$$

where NSHI is the number of static instruction of the application that have been detected as hot code (Number of Static Hot Instructions) and $Cost_{opt}$ is the overhead of optimizing one instruction.

Thus, the execution time of the application can be computed as follows:

$$execution\ time = ((NDI - NDHI) \times CPI_{nat}) + (NDHI \times CPI_{opt}) + (overhead \times CPI_{emu})$$

where, NDI is the total number of dynamic instructions of the application (Number of Dynamic Instructions), NDHI is the total number of dynamic instructions detected as hot code (Number of Dynamic Hot Instructions, which is related to coverage), CPI_{nat} is the number of processor clocks required to execute one instruction in native mode, CPI_{opt} is the number of processor clocks required to execute one instruction in optimized mode (which depends on the value in the Performance Improvement column) and CPI_{emu} is the number of processor clocks required to execute one instruction from the software layer.

In this experiment, we assume that code that is not detected as hot is executed with a CPI of 1. This means that CPI_{nat} is set to 1 in the previous formula, describing the fact that cold code is executed natively in the processor, as in dynamic binary optimization systems in which the guest and the host ISA is the same (for example, Dynamo [36]). Note that hot code coverage could be of much greater importance if we had assumed a lower CPI for cold code in the model (such as in systems in which interpretation is used instead). We also assume that the optimizer has a similar CPI than native execution. For this reason we have also set CPI_{emu} to 1 in the formula.

The potential optimizer overheads using the analytical model for the Spec2000Int benchmarks are shown in Figure 3.25. The x-axis of the figure shows the previously presented 4 groups of optimizations, and the y-axis shows the percentage of additional instructions executed in the system that are required to optimized the hot code. The LIU, pLIU (considering two versions of 16 bits and 17 bits for the threshold) and LFU replacement policies incur in the higher overheads, ranging from 0.01% overhead for the simpler optimizations to a maximum of 5% overhead in the case of pLIU (16 bits). The LFRU goes from 0.01% for the simpler optimizations to 2.12% overhead for the very-aggressive ones. Finally, the LRU replacement policy, that is the one that has also the

lowest coverage, incurs in a 0.0028% overhead for the simpler optimizations and in a 0.84% overhead for the very-aggressive ones. As an interesting reference point, a dynamic binary optimizer that considered all static code to be hot would incur in overheads of 0.53%, 5.26%, 52.58% and 157.73% for simple, medium, aggressive and super-aggressive optimizations respectively.

However, overhead numbers need to be correlated with performance numbers to understand the potential benefits of the dynamic binary optimizer. The performance numbers for Spec2000Int benchmarks using the analytical model are shown in Figure 3.26. The performance is computed as the ratio between the execution time of a design that does not use dynamic binary optimization and the execution time of a design that uses dynamic binary optimization and different profiling strategies. In particular, we show the results for profilers that use the LRU, LFU, LRFU, and the proposed LIU and pLIU (16 and 17 bits versions) replacement policies. As it can be observed, the LIU and pLIU proposals would perform in a similar way. For the simple optimizations they get an 8.4% execution improvement, whereas for the very-aggressive optimizations they get a performance improvement of 47.2% in the case of LIU and 48% in the case of pLIU (16 bits). These benefits cover widely the overheads introduced for optimizing the code. Based on these results, we can say that the expected benefits would be much higher in the LIU and the pLIU proposals than in the other proposals, and especially against LRU. This is mainly because coverage is increased significantly. Moreover, LFU overheads are similar than the ones from LIU and pLIU but the performance improvements are lower. LFU detects approximately the same number of static instructions than the pLIU proposal does, but the static instructions from LFU contribute less to the final coverage. Finally, from Figure 3.25 and Figure 3.26, we can observe that although LFU and LFRU present similar performance improvements, LFRU introduces lower overhead to the system. The main reason of this can be found in the number of static instructions detected by both alternatives. The LFRU detects less static instructions than the LFU but this less instructions contribute more to the final coverage of the application than the ones detected by the LFU.

We have fed the analytical model with other overhead and performance values and we have observed similar trends: the proposed LIU and pLIU proposals increase the overheads minimally compared to other proposals but performance is increased significantly due to: (i) an increase in hot code coverage and (ii) the ability to detect more relevant static instructions as hot code. These results highlight the importance of combining recency and frequency of the accesses for detecting hot code.

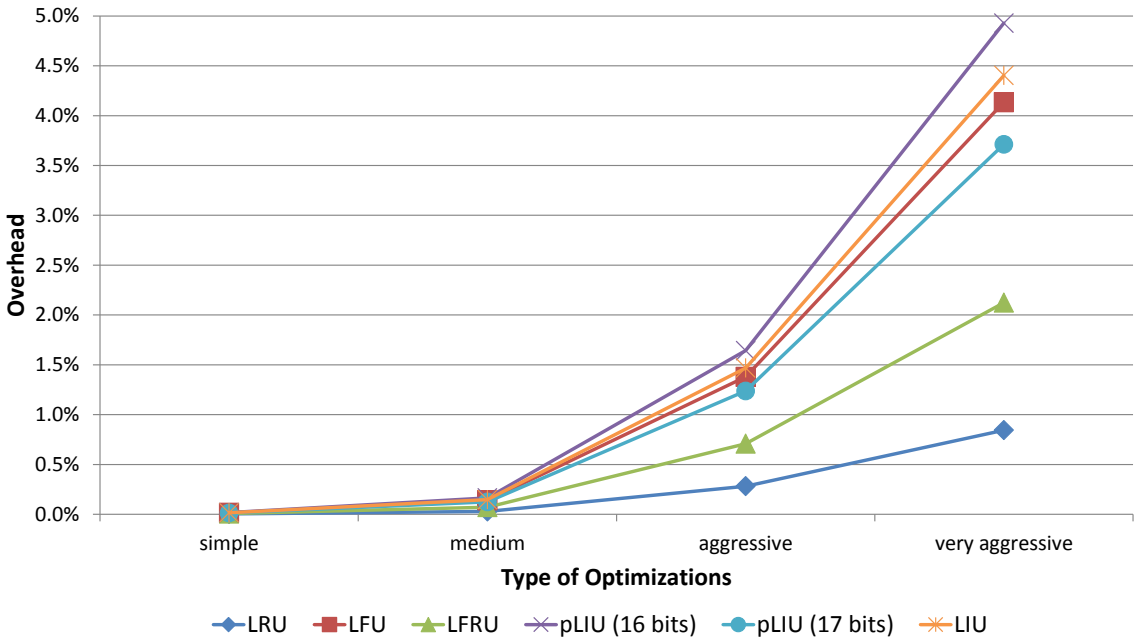


Figure 3.25: Potential optimization overheads in Spec2000Int for different replacement policies using a simple analytical model.

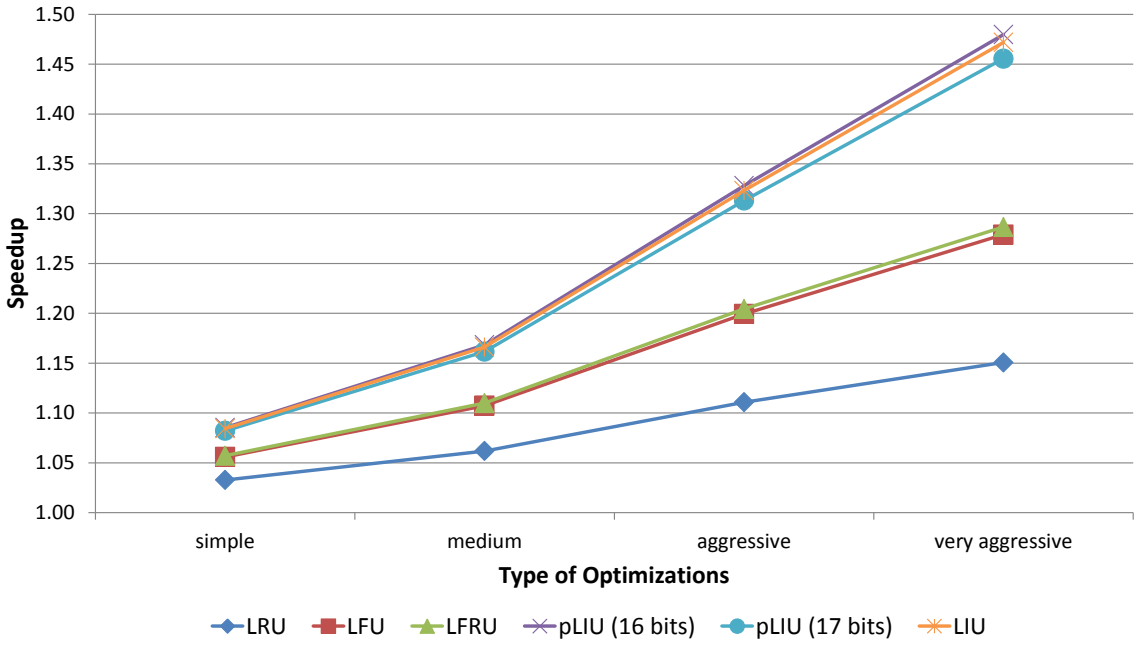


Figure 3.26: Potential speedup for Spec2000Int benchmarks of a dynamic binary optimizer system using different profiling techniques compared to a system that does not implement a binary optimizer using a simple analytical model.

3.4.2.6 Power Requirements

As it has been commented in Section 3.3.5, the LIU profiler has been implemented in a FPGA on top of a processor similar to the Intel® P54C [104]. However, power estimations cannot be made by employing the FPGA model, and moreover, the P54C power numbers are not representative of current processor power budgets because it was developed by employing an old semiconductor manufacturing process[7]. Instead of employing the P54C approach for estimating the power requirements of the LIU profiler, we have selected the contemporary Intel® Atom® [107] [108]. This processor has been built on a 45nm which is more representative of current semiconductor manufacturing processes. Apart from this, we have selected this processor because it has in common with the P54C the in-order execution and the low power design constraints. The Intel® Atom® processor presents a maximum Thermal Design Power (TDP) of 2.2W when it works at 1.6 Ghz.

We have employed CACTI [106] for estimating the power requirements of the LIU Profiler. Unfortunately, CACTI does not allow modeling a small cache structure of 32 sets and 4 ways like the one used by the LIU profiler. For this reason, we have analyzed the power of the LIU profiler by employing a four times bigger cache structure of 128 sets and 4 ways.

Even when considering the extreme case where this structure is accessed in every clock cycle (not only accessed by branches), the LIU profiler only consumes 8.8mW when the processor is clocked at 1.6Ghz. The total read dynamic energy per access to the cache structure is 0.12pJ. Therefore, considering the configuration used for the CACTI estimations, the LIU proposal consumes less than 0.87% of the total power of the Intel® Atom® processor.

3.5 Conclusions and Future Work

Hardware/software co-designed systems require fast hot code detection in order to improve application execution performance by applying different code optimizations/transformations. In this chapter, we have presented a profiling technique that is able to detect hot code regions fast and in a very efficient manner by using a simple hardware table. This hardware table can be complemented by software algorithms and heuristics to build code regions for optimization based on this information. This co-designed approach is able to detect important basic blocks with minimal overheads while maintaining the flexibility by implementing heuristics in software. The pillar of the

[7] The P54C processor was developed in semiconductor fabrication processes ranging from 0.8 µm to 0.25 µm.

hardware component is a novel replacement policy, called LIU, which outperforms the results that can be achieved when considering other traditional replacement policies. The proposed LIU Profiler achieves 85.49% hot code coverage when used in a 128 entries hardware table. Furthermore, it outperforms by 2x the hot code coverage of similar profilers that makes use of random and LRU replacement policies and by 1.5x to profilers that employ LFU and LFRU replacement policies. These other profilers require at least 1024 entries to achieve similar coverage numbers than the proposed LIU Profiler. Moreover, the proposed technique can be implemented requiring few hardware resources, incrementing the total processor area by only 1% and the total processor power by less than 0.87%.

However, some tasks have not been covered in this work. For instance, while we have considered the effects of context switches, we have done so in an analytical way. Hence, it would be extremely interesting to understand the effects of multi-tasking in a real execution environment. Finally, we have just presented the part of the technique designed for detecting hot code but not the software required to build the code regions by using the profiled information. We emplace to future work these drawbacks since we consider that they are out of the scope of this thesis.

Chapter 4

HW/SW Register Checkpointing

Because code transformations in a hardware/software co-designed processor often imply instruction reordering, instruction removal, and other speculative optimizations, the execution of a dynamically optimized region must be observed as atomic. This is because the architectural state may not be fully correct during the execution of an optimized region as long as it matches the original architectural state at the end of the region. In addition, atomic execution is also a requirement if precise exceptions are assumed, as it is the case for most modern ISAs.

After executing a dynamically optimized region in a hardware/software co-designed system, the architectural state of the processor must be exactly equal than the one that would be generated by the execution of the same code region in its original form. However, the way the architectural values are produced in the optimized region may be different when compared to the original execution. If speculative optimizations are considered, even wrong values maybe produced during execution. For this reason, if an exception or a mispeculation occurs before the end of the region completes, then the architectural state of the system may differ significantly when compared to the original. In addition, supporting precise exceptions implies recovering the state of the machine as if the original code would have been executed until the precise point where the exception occurs [109]. In order to achieve this, a hardware/software co-designed system implements a mechanism that allows the processor to restore a previous architectural state, including memory and registers. In case an exception or a mispeculation occurs, the architectural state is recovered and the region is executed again in its original form in order to reproduce the exception as in the original code (see Section 2.5.2 for more details) or to avoid the mispeculation failure.

In this section, we focus on the architectural registers. For this particular case, the most common approach in the literature is to save (checkpoint) the value of the architectural registers in a secondary register file, called Shadow Register File (SRF) [13] [39] [110]. The movement of the data through both register files is made with a flash-copy operation that can normally be achieved within a single processor clock cycle. If a mispeculation or an exception occur, the contents of the shadow register file is copied back to the regular register file and the region is rolled back. The implementation of this pure hardware checkpointing mechanism increases the complexity of the processor design. In addition, in simple cores, such as the Intel® Atom® selected for the studies of this thesis, incrementing the size of the register file may affect significantly the power and the area of the design.

In this section, we propose a hardware/software co-designed scheme, called HRC (Hybrid Register Checkpointing), where the software is in charge of checkpointing and recovering the register state of the processor on demand and the hardware offers mechanisms to speed up the process. The key feature of this scheme is the low additional hardware support compared to pure hardware solutions [111]. In particular, no additional registers are required because already existing ones are used instead. Moreover, only those registers modified within the code region need to be checkpointed. The proposed scheme incurs in less overheads than previously software checkpointing and recovery mechanisms [29] [112] [109], and it is intended not only to be used for exception recovery but also to help on performing aggressive dynamic code optimizations (e.g. code regions optimized speculatively).

Results show that the HRC achieves almost the same performance as a zero-cycle register file flash-copy solution (copying the register file into a shadow register file) without the additional hardware and additional complexity this latter solution requires. In fact, the proposal only degrades performance by 1% when compared to the SRF proposal. In addition, the proposal is still very competitive when checkpointing resources are scarce, guaranteeing coverage of 95% of optimized code. Moreover, the HRC scheme reduces by 11% the area and by 19.4% (Spec2000FP) and 30.1% (Spec2000Int) the power of the SRF solution.

The rest of this chapter is organized as follows. First, we introduce the checkpointing main concept. In Section 4.2, we describe previous work done in the literature. In Section 4.4, we present the HRC implementation and in Section 4.5, we discuss the evaluation results. Finally, in Section 4.6, we conclude the chapter by presenting the conclusions and future work.

4.1 HRC Overview

In order to correctly update the architectural state of the machine, we need a mechanism to keep track of the values produced during the execution of an optimized region and a recovery mechanism to undo the work done when an exception or a miss-speculation event occurs. Common solutions generate a checkpoint at the beginning of each optimized code region with the state of the architectural registers at that point. In particular, the content of the register file is flashed into a shadow register file. When an exception occurs, the state is recovered from this shadow register file copy.

Architectural registers are those specified by the instruction set architecture (ISA) of the processor. In the particular case of the Intel® Atom® processor, this includes general purpose, x87, MMX, XMM, and EFLAGS registers [113] and these are the ones that are checkpointed, as they define the architectural register state. Apart from these registers, there are other registers not visible to the applications (programmers) but accessible by the hardware resources [113]. These registers are commonly called non-architectural or temporal registers and they are not checkpointed because they are not part of the ISA.

The proposed register checkpointing mechanism uses software to create and restore register checkpoints. In particular, the software needs to copy the original value of an architectural register before this register is speculatively overwritten. Since the software layer is in charge of generating the optimized regions, it knows which registers are read and which are written within the region and it also knows when they are consumed. Therefore, the software layer can easily keep track of these registers to implement the checkpointing and the recovery mechanisms.

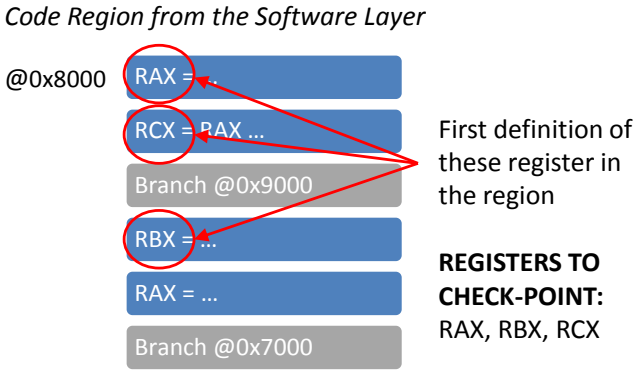


Figure 4.1: Example of first usage of registers in a code region.

We use Figure 4.1 to illustrate how the software layer identifies which registers need to be checkpointed. The figure shows an example of a code region generated by the software layer. As it is shown, three architectural registers, RAX, RCX, and RBX, are written by instructions of the region. We also highlight when these registers are written for the first time. In this case, the software layer needs to checkpoint the value of the RAX, RCX, and RBX registers before their first definition. The idea of the proposal is to introduce instructions to copy explicitly the values to temporal registers. In the example of the figure, the software layer would introduce extra move instructions to copy the values of RAX, RBX, and RCX into available temporal registers. Note that these new move instructions would be handled by the software layer as any other instruction when doing optimizations such as code reordering. In particular, the only restrictions to these instructions is that they cannot be eliminated (for instance, when applying dead code removal) and they need to be scheduled before the corresponding register is defined.

If the execution of the region is successful, the temporal data stored for the checkpointed values can be discarded and the values produced by the region become the new architectural state. In contrast, in the case of region rollback, the recovery mechanism takes the checkpointed information and recovers the state of the registers before the region was executed. This can be implemented by employing a software routine that is formed by a sequence of move instructions in charge of copying the checkpointed values from the temporal registers into the corresponding architectural registers. This software routine is called region recovery code.

The proposed software register checkpointing mechanism presents the following characteristics:

- Minimal hardware support required (see Section 4.4 for more details). In particular, there is no need for an additional shadow register file as required in hardware approaches. The proposal makes use of the currently available temporal registers in the processor.
- Small changes required in the software layer implementation (see Sections 4.4.1 and 4.4.2 for more details).
- Low impact on performance since only registers that are modified by the region need to be checkpointed. Hence, few extra instructions are introduced. In particular, it performs only 1% worse than with pure hardware solutions (see Section 4.5.2 for more details).

- Low overhead for the recovery mechanisms when compared to other software checkpointing solutions, which enables aggressive speculative optimizations.

4.2 State of the Art

Several techniques have been proposed in industry and academia on how to handle the speculative state of dynamic optimizers in hardware/software co-designed processors that use speculative optimizations.

RePlay [114] and Parrot [22] [115] build traces of instructions after the commit pipeline stage and optimize them in hardware. They target superscalar out-of-order machines, and therefore rely on already existing hardware checkpointing mechanisms. Although adding or extending these mechanisms to an in-order pipeline is possible, it would significantly impact the complexity of the design, making it not very different from a traditional out-of-order design. Since our proposal targets simple in-order processors like the Intel® Atom®, we need a different checkpointing mechanism.

Speculative optimizations are also used by Transmeta's Crusoe® [16] and Efficeon® [14] [110] processors when optimizing x86 code on top of a VLIW in-order processor. Transmeta's processors rely on a hardware checkpointing mechanism. Actually, all registers are shadowed, that is, there are two copies of each register, a working copy and a shadow copy. Instructions in an optimized region only update the working copy. When the optimized region completes successfully, the contents of the working copy are copied to the shadow one. On the other hand, if an exception or a mispeculation occurs, the shadow copy containing a previous valid state is copied back to the working copy, restoring the previous valid architectural state. This requires doubling the number of registers and some additional logic to perform the copies among the registers files. Moreover, the registers used for checkpointing cannot be used for other purposes.

IBM® DAISY [30] and BOA [29] projects also use the hardware/software co-designed virtual machine concept to execute PowerPC code speculatively on top of a VLIW processor. These systems rely on a software-controlled checkpointing and recovery mechanisms to provide precise exceptions. They propose two different alternatives. The basic approach [30] relies on using a set of non-architectural registers. The registers accessed while executing an optimized region are renamed to non-architectural registers, keeping the architectural registers untouched. Therefore, the speculative state generated is not promoted to the architectural state. The moves from speculative to architectural state are done by software at the end of each code region, when the state can be safely

committed. Hence, the execution of these move instructions cannot be overlapped with the execution of the other instructions belonging to the optimized region, conversely to the instructions introduced by our proposal. Moreover, such explicit move instructions have to be executed always introducing an additional overhead to the system that penalizes the most common scenario, which is committing correctly the region.

The second alternative proposed for the DAISY and BOA is based on keeping the results always, even after the commit of the region, in non-architectural registers and only move them to the architectural registers when required [9]. Once an exception arises, a repair mechanism is called, which is part of the software layer. The repair mechanism produces the appropriate recovery code based on information previously generated when the code was optimized. Generating recovery code at run-time is possible because the virtual machine translator has always the control of the system and knows at each point where the non-speculative values reside. This scheme and our proposal differ in the following: (1) additional information (annotations) must be stored for each optimized region to generate the recovery code [109] whereas our scheme does not, and (2) the former scheme incurs in important overheads compared to our proposal when an exception arises, because it requires the virtual machine to generate recovery code on demand.

The important overheads associated to the second alternative used in DAISY and BOA may be affordable for handling exceptions and interrupts, which rarely occur. However, the mechanism proposed in this thesis can also be applied as a fast recovery mechanism to enable aggressive speculative optimizations. In order to exploit this feature, the incurred overheads must be negligible, which is the case of the proposed scheme but may not be the case for the second solution proposed for DAISY and BOA.

4.3 Baseline Core Characteristics and Pipeline

The core employed for implementing the software checkpointing scheme is based on the Intel® Atom® processor [107]. This processor follows the bases commented in the introduction of this chapter since it is a small in-order core with very low power consumption. Both characteristics make it ideal for its integration in a multi-core environment.

The Intel® Atom® supports the x86 ISA, but internally the instructions are split into simpler RISC instructions called μops. The core allows the execution of two μops per clock cycle, and it can be used in a traditional single thread execution mode or in a two simultaneous multithreading execution mode. In this second mode, the two threads

compete for the core resources. In particular, the caches, the decoders, the ports, the branch predictor, the branch target buffer (a.k.a. BTB), and the execution units are totally shared between the threads. By contrast, the prefetch buffers, the instruction queues (IQs), and the registers files are replicated for each thread. Moreover, both threads are treated equally for accessing to the shared resources, having availability to a particular resource every two clock cycles in case of conflict. The maximum throughput of the core is always two instructions per cycle independently of the number of executing threads.

The core has three caches. A first level instruction cache of 32KB and 8-way set associative, a first level data cache of 24 KB, and 6-way set associative and a second level cache of 512KB (or 1MB) and 8-way set associative. All caches use a cache line size of 64 bytes. The memory unit is connected only to the integer cluster, and so, floating point/SIMD instruction require more time for execution. The processor also supports store forwarding in a very efficient manner allowing reading back to back a value written by a previous operation.

The pipeline of the core has sixteen stages. The front-end pipeline has three stages for instruction fetch, three for instruction decoding, and two for instruction dispatch. The backend pipeline has one stage for reading the register operands, one for calculating the address of a memory operand, two for accessing the data cache, one for execution, two more for exception and multithreading handling, and one for committing the results. Both front-end and backend pipelines are graphically described in Figure 4.2 and Figure 4.3 respectively. This pipeline allows the execution of read-modify and read-modify-write x86 type instructions in one cycle. Other more complex instructions require to be split into serial µops. To guarantee the execution of two instructions per cycle the execution unit has two ports, each one connected to the integer and the floating point execution units.

The front-end and the back-end pipelines are decoupled. Instructions (in this case µops) after the decoding stages are placed into a queue of 32 entries. In multithread execution mode this queue is split into 16 entries per thread.

The simple integer instructions have an execution latency of one clock cycle. The multiplications, divisions and floating point instructions require longer latencies (from 5 cycles in the case of *imul* instruction and up to 31 cycles for floating point divisions). Moreover, moving data between a SIMD vector register and a general purpose register or a flag register require 4-5 clock cycles due to the communication required between separated register files for SIMD and integer data.

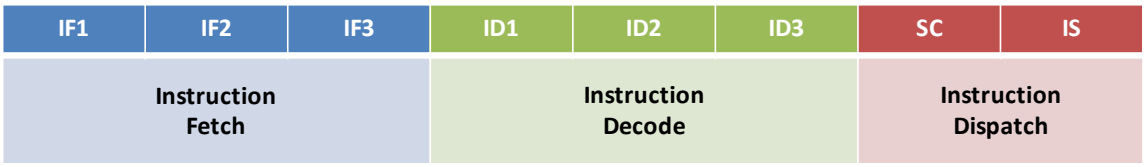


Figure 4.2: Processor front-end pipeline.

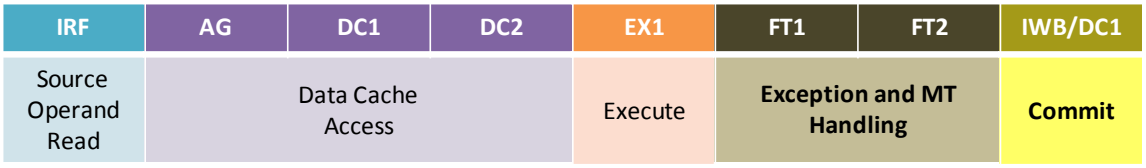


Figure 4.3: Processor back-end pipeline.

4.4 Detailed HRC Implementation

The proposed hybrid hardware/software register checkpointing scheme requires minimal hardware support to be implemented on top of a low-power processor such as the Intel® Atom® [107]. In particular, the checkpointing and recovery mechanisms only require a new type of *move* instruction and a dedicated register.

This new register is called RCIP (Recovery Code Instruction Pointer) and it keeps track of the IP address where the recovery code resides. The RCIP register is updated at the beginning of every optimized region executed with a regular move instruction, as each region has a different recovery code routine.

The new type of instruction is a conventional *move* that additionally updates the RCIP register. We refer to this new type of *move* instruction as *CKP_MOV* and it is distinguished from regular *move* operations by using a new opcode. *CKP_MOV* instructions are used to save the content of the architectural registers before they are overwritten for the first time within the region. In such a way, the saved values correspond to the architectural register state before the execution of the region. Moreover, every time a *CKP_MOV* instruction is executed, the RCIP register is updated (subtracting a fixed amount from the value already stored) in order to prepare the recovery mechanism to restore the previous values of the checkpointed architectural registers in case there is a mispeculation or an exception.

Temporal/non-architectural registers are already implemented in most processors nowadays [113] and it is not required to add them for the HRC scheme. However, if there are not enough temporal registers in the processor to keep track of the architectural state

of a region, we have assumed that the software layer will not be able to optimize that region. As we will show in Section 4.5.2.2, this is not the case for more than 95% of the regions.

4.4.1 Checkpointing Mechanism

A register checkpoint is performed by inserting a set of *CKP_MOV* instructions at the beginning of each optimized code region. *CKP_MOV* instructions are included within the original code region to preserve the value of the registers. Each architectural register written within the original region has its corresponding *CKP_MOV* instruction, which copies the architectural register before it is overwritten by any other instruction. These instructions are included before the software layer generates the optimized version of the region, so that the instructions are also considered for optimization with the rest of the instructions of the region. This means that these *CKP_MOV* instructions can be reordered and optimized with the rest, except that they cannot be eliminated by optimizations such as dead code removal and they cannot be reordered below the instruction that defines the register being checkpointed.

A first regular *move* operation included in the region initializes the RCIP register that indicates the address where the recovery code for this particular region resides. This instruction must be executed prior to all other instructions belonging to the same region and it cannot be reordered as the CKP_MOV instructions. Note that a regular *move* instruction can be employed in this case because the software layer has full access to all registers in the processor, including the RCIP and all temporal registers.

The architectural state is then distributed between architectural registers (those architectural registers that have not been overwritten in the region yet) and temporal registers (those architectural registers that have already been overwritten in the region). By using temporal registers, the movement of data between the speculative and the architectural state and vice versa can be done in a fast and cheap manner. However, we are constraining the usage of temporal registers for other purposes since processors normally have a small number of dedicated temporal registers. As shown later in Section 4.5.2, we decided to use the temporal registers because the pressure over these registers is not increased significantly.

```
writtenRegistersQueue.clear();
for each instruction i in region r do
        if !(writeRegister(i) ϵ writtenRegistersQueue) &&
           (writeRegister(i) ϵ architecturalRegisters) then
                writtenRegistersQueue.push_back(writeRegister(i));
        endif
end for
```

Figure 4.4: Software layer algorithm to indentify the registers that require checkpointing within a region.

In Figure 4.4, we show the algorithm that the software layer employs to identify which registers are required to be checkpointed. As it has previously been described, all architectural registers written during the execution of the region need to be checkpointed. Thus, the algorithm traverses in program order all instructions within the region and for each one it checks (i) if the instruction has a destination register, (ii) if such a register is an architectural register (the algorithm uses a list named *architecturalRegisters* that stores the identifiers of all architectural registers used in the processor), and (iii) if such a register is defined for the first time in the superblock. The algorithm uses a list named *writtenRegisterQueue* where it enqueues the registers that satisfy the three conditions as it traverses the instructions. Note that in such a way, the registers are stored in their definition order. The *writtenRegisterQueue* is later used by the software layer to insert the checkpointing *CKP_MOV* instructions.

In Figure 4.5, we show an example on how an original code region is modified with the additions of the checkpointing instructions and the recovery code. On the left of the figure, we show the original region. It starts at address 0x8000 and it contains two branch instructions and five instructions writing into a register. From these instructions, only architectural registers RAX, RBX, and RCX are detected by the algorithm depicted in Figure 4.4, and so, only these three registers need to be checkpointed (Tmp10 is not an architectural register). On the right part of the figure, we show the original region with the checkpointing and the recovery codes. Note that the address of the optimized region is different from the address of the original one because the software layer maps optimized code regions into the code cache memory. The checkpointing instructions are placed at the beginning of the region whereas the recovery code is placed at the bottom.

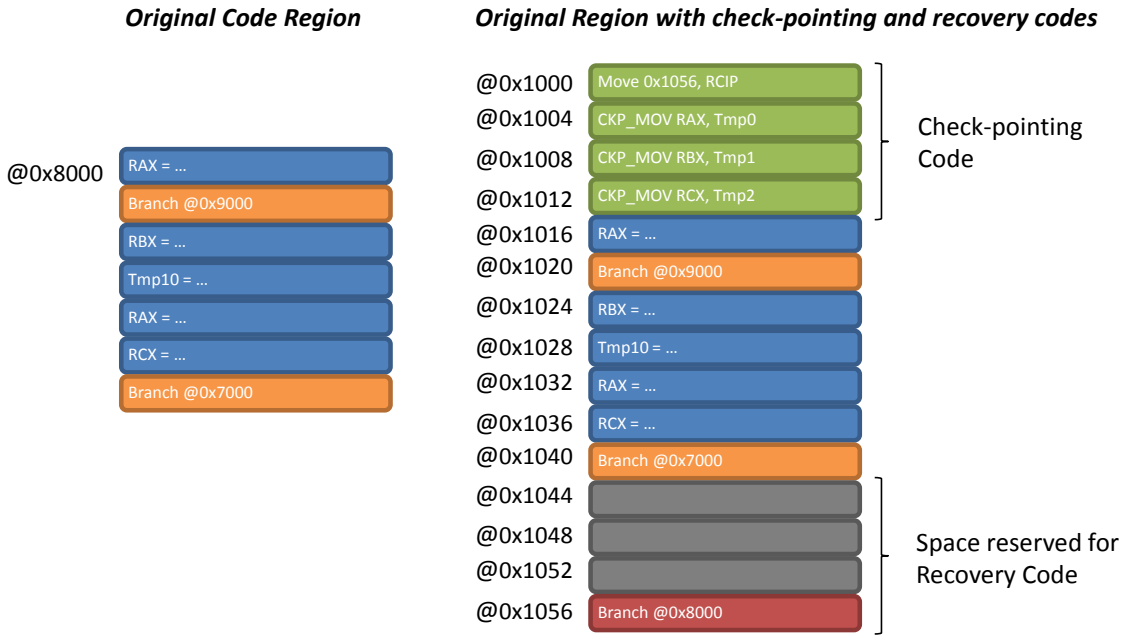


Figure 4.5: Example of code generated when using the checkpointing and recovery mechanism.

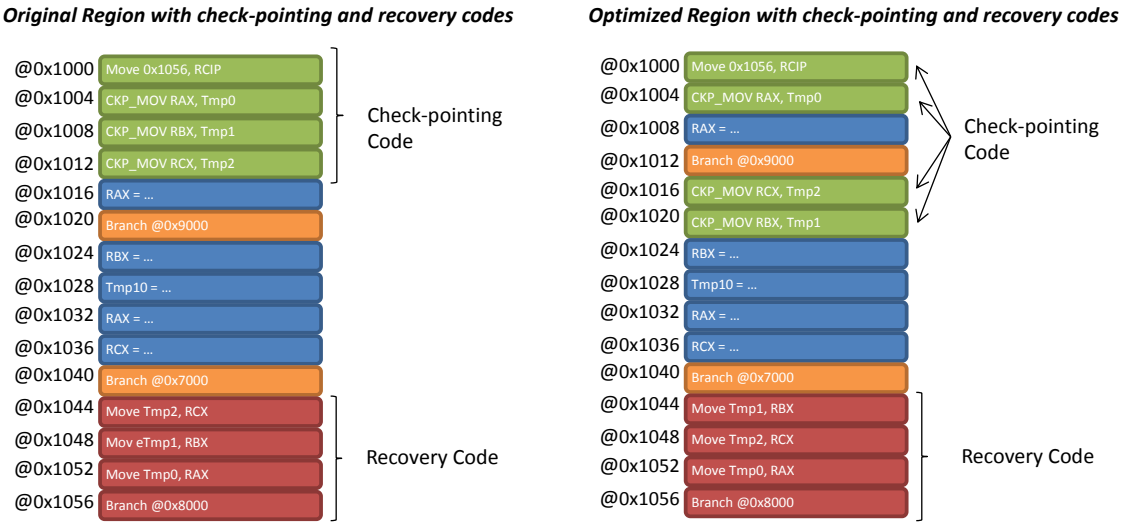


Figure 4.6: Example of code reordering applied on top of a region that already includes the checkpointing and recovery codes.

The first instruction of the checkpointing code initializes the RCIP register to point to the last instruction of the recovery code that is placed at address 0x1056 (see Section 4.4.2 for more details). The rest of the checkpointing instructions are the *CKP_MOV* for storing the initial values of RAX, RBX, and RCX into Tmp0, Tmp1, and Tmp2 respectively. The recovery code is depicted as empty spaces because recovery code is generated once the

region is optimized, as we explain later in Section 4.4.2. As for now, we just need to know that there is space for the same amount of CKP_MOV instructions attached to the region plus one.

Finally, once the checkpointing code is created, the code region is passed to the software layer optimizer which performs aggressive speculative optimizations such as code reordering (instruction scheduling), control, and data speculation, dead code removal, common subexpression elimination, among others. For example, in Figure 4.6, we show how the original region presented in Figure 4.5 may be optimized once the checkpointing and recovery instructions are included. As it can be observed, the "CKP_MOV RBX, Tmp1" instruction at address 0x1008 is reordered and placed at address 0x1020 and the instruction "CKP_MOV RCX, Tmp2" is also reordered and placed at address 0x1016. Therefore, the checkpointing code is intermixed within the instructions that belong to the region and they are handled as regular instructions. The only restriction when reordering instructions is that an instruction overwriting an architectural register for the first time cannot be executed before its associated CKP_MOV instruction. The software layer ensures this by adding an anti-dependence relation between the instruction overwriting the architectural register and the associated CKP_MOV instruction in the Data Dependence Graph (DDG) of the region while doing optimizations. In Figure 4.4, we can see that all CKP_MOV instructions precede their associated register overwritten instructions. Note also that, although the "CKP_MOV RBX, Tmp1" is first in the original order, the "CKP_MOV RCX, Tmp2" precedes it in the reordered version of the region. Moreover, this latter instruction is separated from the first RCX overwriting register instruction by several instructions, highlighting that the software layer may decide to separate the CKP_MOV instruction from its associated register overwriting instruction if there is an opportunity to have a better instruction scheduling.

Since *CKP_MOV* instructions can be reordered, the recovery code is generated once the final disposition of the *CKP_MOV* instructions is known. This guarantees that the *move* instructions in the recovery code are executed in reverse order than the register checkpointing definition order, which is necessary for the recovery mechanism as described in next Section 4.4.2.

4.4.2 Recovery Mechanism

When an exception or mispeculation occurs during the execution of the optimized region, control is given back to the software layer, with the objective of executing the region in its original version in order to reproduce the exception with the correct architectural state if

required (real exception) or to guarantee forward progress (mispeculation). In any case, prior to transferring the control to the re-execution of the region, the software layer executes the recovery code to re-establish the architectural register state.

The recovery code is in charge of moving back the state stored in the temporal registers to the architectural ones. After that, the recovery code transfers the control to the original code to reproduce the exception. In order to accomplish this, a branch instruction that jumps to the initial IP of the original code is added as the last instruction of the recovery code. The recovery code is placed at the end of the optimized region, after the last exit instruction and it is not subject to optimizations. Note that this code is only executed in case an exception is raised.

An exception or mispeculation may arise at any point in time when running the optimized code region. Since the *CKP_MOVs* instructions are mixed and reordered with the rest of the instructions, there is no guarantee that all of them have been executed at the moment the exception occurs. Therefore, specific recovery code must be executed to recover the state at each possible execution point of the optimized code region. However, this may significantly increase the amount of instructions stored in the code cache. Instead, the proposed mechanism generates the recovery code assuming that all the checkpointing moves have been executed, and relies on the contents of the RCIP register in order to only execute the appropriate recovery move instructions.

As commented before, the software layer generates the recovery move instructions in reverse order than that of the *CKP_MOVs* inserted within the region code. The RCIP register is initialized to point to the last instruction of the recovery code. This last instruction is the branch that jumps to the beginning of the original code region. At this point, since no *CKP_MOVs* have been executed yet, the recovery code is only formed by this branch instruction. Once the first *CKP_MOV* instruction is executed, it decreases the RCIP pointer by an amount equal to the size of a move instruction, including in such a way its associated recovery move counterpart operation in the recovery code (the move instruction that recovers the checkpointed value for the corresponding register). As more *CKP_MOV* instructions are executed, the RCIP register is updated to include their recovery instruction counterparts. Thus, an exception can occur at any point in time in the optimized code region because the RCIP register always points to the appropriate set of recovery move instructions. Note that since regions are stored internally as superblocks (single entry multiple exists block regions), the order and the number of *CKP_MOV* instructions is always the same within the region because there is no multiple control flow

within the region. This guarantees the correct order between the CKP_MOV instructions and the counterpart recovery move instructions.

To better illustrate how the RCIP register is updated during the execution of a region, we use the example of Figure 4.7. The code shown in the figure includes the instructions for performing the checkpoint (*CKP_MOV* instructions) and their respective recovery instructions (highlighted at the bottom of the region). The first instruction in the region initializes the RCIP register to point to address 0x1056. At this point, in the recovery code there are no *move* instructions since any architectural register has been checkpointed. After the first *CKP_MOV* instruction is executed (move from RAX to Tmp0), the RCIP is updated to point to IP 0x1052, where the counterpart move resides (move from Tmp0 to RAX). After the execution of the *CKP_MOV* for RCX, the RCIP points to 0x1048 where the counterpart move instruction that restores RCX from the Tmp2 register resides. Finally, once the *CKP_MOV* instruction for the RBX register is executed, the RCIP points to 0x1044 which allows the execution of the whole recovery code if needed to guarantee precise exceptions or to recover from a miss-speculation.

4.4.3 Hardware Implementation Details

The update of the RCIP register occurs in the write-back stage of the Intel® Atom® processor pipeline. The architectural registers are also updated at this same stage. Therefore, the update of the RCIP register must occur in a synchronized manner with the update generated by the instruction overwriting the architectural register. In Figure 4.8, we show two examples of the execution of the *CKP_MOV* instruction and the associated instruction that overwrites the register to be checkpointed. In the case a), the *CKP_MOV* instruction is executed at the same time that the instruction that overwrites the register R. If an exception occurs during the execution of this latter instruction, which is detected in FT1 and FT2 stages, the *CKP_MOV* operation will not retire and the RIP register and the content of register R will not be updated. In the case b), both instructions are executed back-to-back (the instruction that overwrites register R is executed one cycle later than the *CKP_MOV* instruction). In this case, if an exception arises when executing the second instruction, the content of register R is not updated (the instruction does not retire). However, the *CKP_MOV* will retire and therefore it will update the RIP register. Fortunately, this is not a problem because the value of register R and the checkpointed value are equal at that point. Therefore, the recovery mechanism will overwrite register R with the same original value, and although RCIP and R will not be updated in a synchronized manner, the architectural state will be correctly recovered despite the

execution of the unnecessary move operation. In case that the instruction that overwrites register R is executed more than 1 cycle later than the corresponding *CKP_MOV*, the same rationale described for case b) applies. Note that there are no other possible scenarios for updating the RIP and the architectural registers because the overwritten operation for an architectural register cannot occur before its corresponding *CKP_MOV* operation (the software layer prevents this scenario to occur as it is commented in Section 4.4.1).

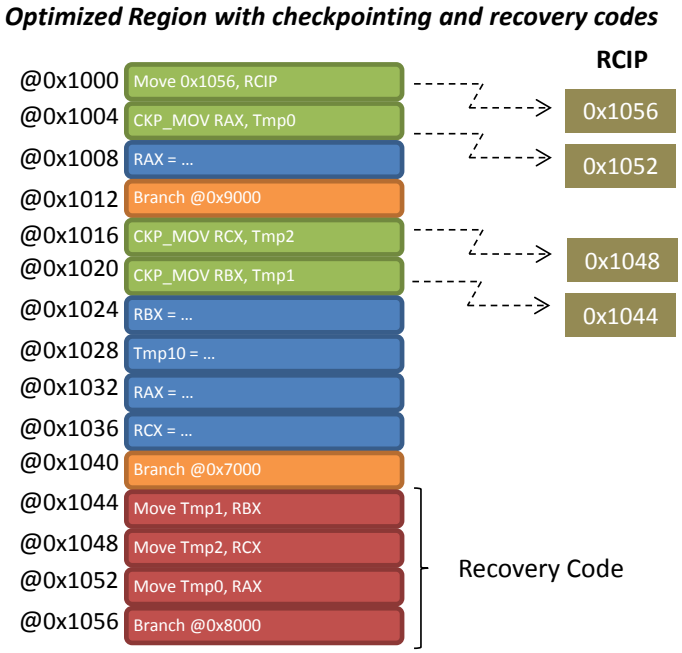


Figure 4.7: Recovery mechanism functionality example.

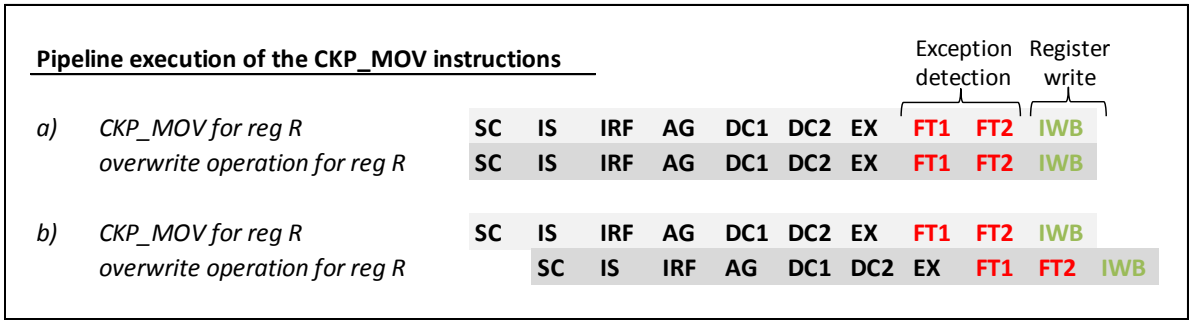


Figure 4.8: Processor pipeline execution examples of the CKP_MOV instruction. In case a), the instruction is executed with its corresponding instruction that overwrites the checkpointed register. In case b), the two instructions are executed back to back.

4.5 Evaluation

4.5.1 Simulation Framework

The HRC scheme has been implemented on top of an in-house research simulator that models a hardware/software co-designed processor. The hardware component of the simulator models a processor similar to the Intel® Atom® and it is based on the architecture described in Section 4.3. This cycle accurate simulator allows up to 4 instructions (µops) to be issued and retired simultaneously and it models the Intel® x86 ISA [113] in very high detail, including interrupts, exceptions, APIC support, etc.

The software layer implemented in the simulator uses profiling counters in order to detect pieces of code that are frequently executed. When a hot code is detected, instructions belonging to that code are grouped into a code region. Currently, the software layer groups µops into superblocks which are single-entry multiple-exits block regions. Once a superblock has been constructed, instructions for the proposed mechanisms are included into it. Code reordering and register reallocation are applied over the whole superblock (including the checkpointing instructions but not the initial move and the recovery code).

We have compared the software register checkpointing technique presented in this thesis against a hardware checkpointing mechanism based on a shadow register file (SRF). We have included the new type of *CKP_MOV* instructions and the RCIP register in the hardware component of the simulator, and we have implemented the software checkpointing and recovery mechanisms in the software layer of the simulator. Moreover, the SRF technique has also been implemented in the simulator.

In this section, we evaluate the performance impact of the proposal and also the pressure it exerts over the registers. The proposed scheme makes use of the non-architectural registers of the processor, which reduces the number of available registers for the instruction scheduler. Both performance and register pressure metrics are also determined by the register allocator and the instruction scheduler employed. The software layer implements a SSA-based register allocator [116], and two different instruction schedulers. These schedulers analyze and reorder the instructions of the detected superblocks in order to improve instruction-level parallelism. One of these schedulers is based on a Top-Down (TD) approach and the other is based on a Bottom-Up (BU) approach. The Top-Down (TD) scheduler analyzes the code from the entries of the superblock to the exits, whereas the Bottom-Up (BU) scheduler proceeds from the exits to the entries. TD pulls control-dependent instructions up across the control-flow branch points (last instruction of a basic

block in the superblock) and BU pushes them down across the control-flow joint point. Pulling up introduces speculative execution and pushing down requires predicated execution. Therefore, TD scheduling tends to increase register pressure but it is better than BU to exploit speculation and improve ILP performance [117].

We have performed simulations by using 22 programs from the Spec2000 benchmark suite. The selected programs are *bzip2, crafty, eon, gap, gcc, gzip, mcf, parser, perlbmk, twolf, vortex,* and *vpr* from the Spec2000Int and *art, ammp, applu, apsi, facerec, fma3d, galgel, mesa, sixtrack, and wupwise* from the Spec2000FP. Each benchmark is simulated by using a set of representative traces of 20 million instructions. These traces have been obtained by gathering the program information directly from real processor activity. For each trace there are two simulation phases. During the first phase, the software layer identifies the regions and generates the optimized code with the checkpointing additions. In the second phase, the optimized regions are executed in the hardware simulator and relevant statistics are collected. In such a way we do not assume overheads due to the use of the dynamic optimizer. However, the overhead of the dynamic optimizer should not increase significantly when moving from the hardware checkpoint to the software one.

Finally, we have estimated the area and the power consumption of the HRC scheme by employing a model based on CACTI for on-chip caches [106]. We have configured CACTI to model a regular register file and a shadow register file. Moreover, for the shadow register file we have considered an optimized implementation that minimizes the dynamic power overheads when the registers need to be checkpointed. More details in Section 4.5.2.3.

4.5.2 Results

We have evaluated the proposed checkpointing mechanism by analyzing the performance impact due to the additional instructions introduced in the optimized code regions, as well as the impact on register pressure due to the usage of the non-architectural/temporal registers.

4.5.2.1 Performance Impact

The HRC scheme has been compared against a hardware checkpointing mechanism based on a SRF scheme. We have assumed that the hardware checkpointing mechanism incurs in a zero-cycle penalty and that the HRC scheme uses unlimited number of temporal registers.

In Figure 4.9 and Figure 4.10, we show the performance impact of the proposal for Spec2000FP and Spec2000Int benchmarks respectively. The performance results are shown by varying the issue width of the processor and assuming a Top-Down (TD) instruction scheduler approach. The baseline is the Intel® Atom® processor without dynamic binary optimization support. As it can be observed, the additional extra instructions required for the checkpointing do not introduce an important penalty for issue width 4. On average, in this case the slowdown is 1.75% and 1% for Spec2000FP and Spec2000Int respectively. The slowdown slightly increases when the issue width is decremented. On average, it is 7.9% and 5% for 2-issue and 3-issue for Spec2000FP benchmarks, and 7.2% and 2.7% for 2-issue and 3-issue for Spec2000Int benchmarks.

When Bottom-Down (BU) instruction scheduling is employed the performance numbers are slightly lower. In Figure 4.11, we show the speedup results of the HRC proposal for the Spec2000FP and Spec2000Int benchmarks and different processor issue widths. As it can be observed, the slowdown for BU is always less than 2% with respect to TD. In particular, for 4-issue width, the different between the two instruction schedulers is 1.2% and 0.6% for Spec2000FP and Spec2000Int respectively. BU allows less speculation and it is not able to exploit as ILP as the TD approach.

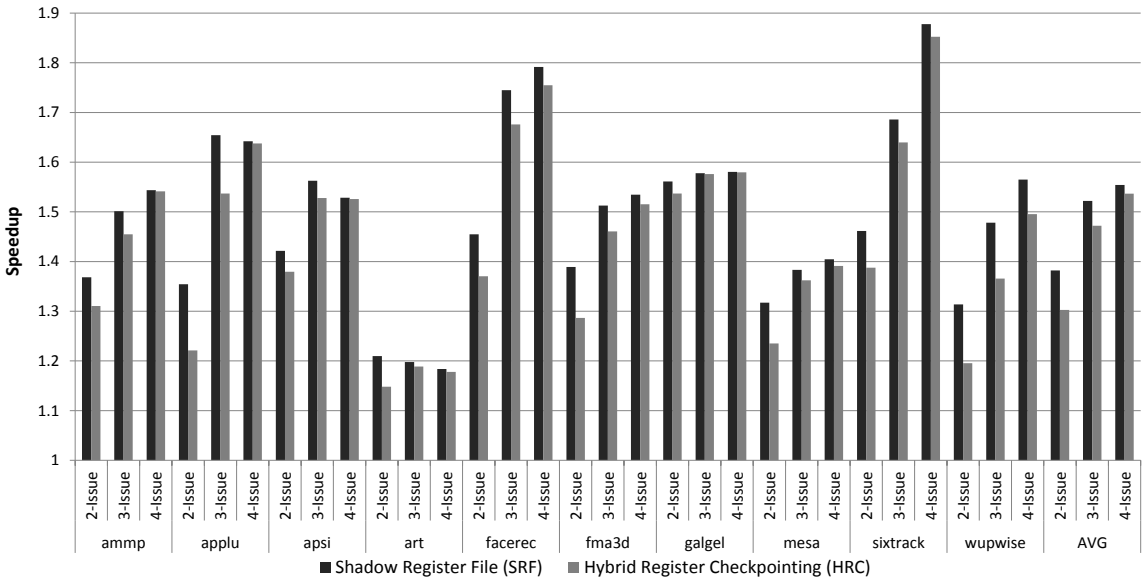


Figure 4.9: Performance impact of the HRC scheme when compared to a traditional SRF scheme. Results for Spec2000FP benchmark suite and top-down instruction list scheduling.

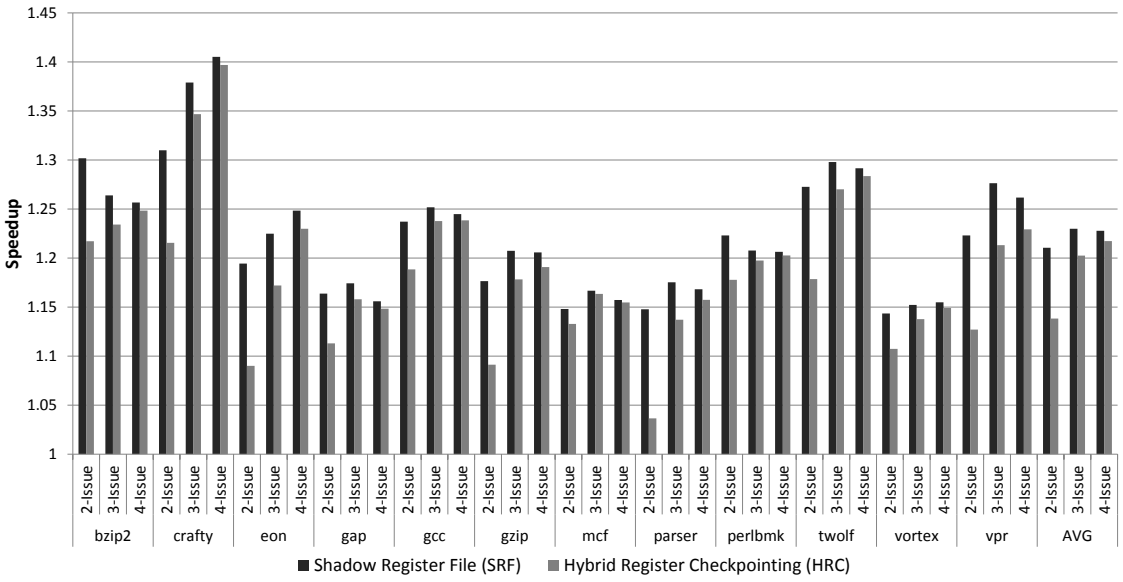


Figure 4.10: Performance impact of the HRC scheme when compared to a traditional SRF scheme. Results for Spec2000Int benchmark suite and top-down instruction list scheduling.

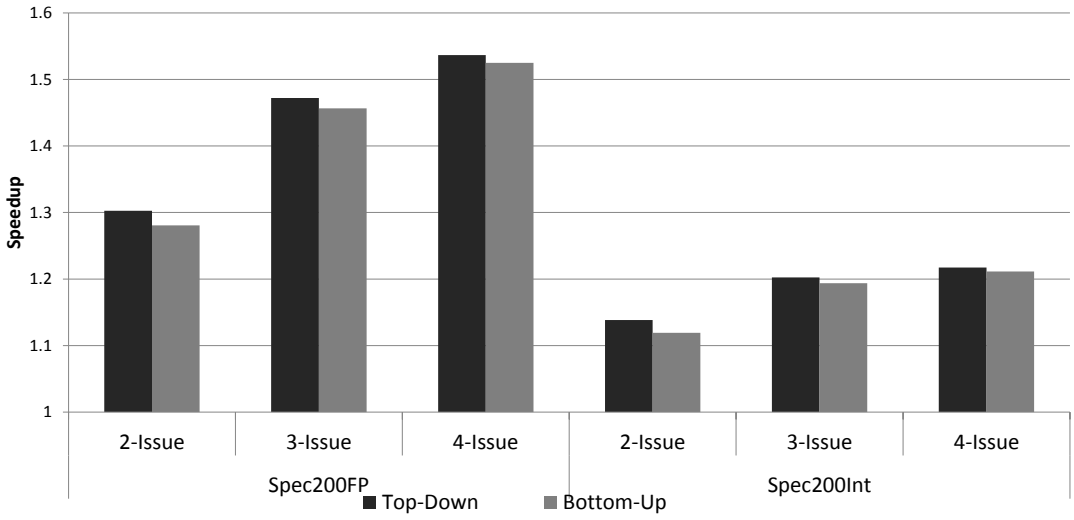


Figure 4.11: Performance of the software checkpointing with TD and BU list-scheduling.

4.5.2.2 Register Pressure

We have also evaluated the impact on performance when restricting the amount of temporal registers in order to understand how register pressure is affected when introducing the extra move instructions. As we explained before, our dynamic binary optimizer does not generate an optimized version of a code region if the register allocator fails at assigning registers to instructions. Thus, we show how register pressure is affected by employing the coverage metric, that is computed as the ratio of dynamic instructions

executed in optimized mode in comparison to the total number of dynamic instructions of the program.

We have considered different number of temporal registers assigned to save the architectural non-committed state of the processor, including integer and floating point registers. Figure 4.12 shows the coverage obtained when employing the HRC scheme limiting the number of integer temporal registers. In the y-axis, we show the region coverage and in the x-axis we show the number of registers available for the checkpointing scheme. As it can be observed, the proposal is able to optimize 98% of the code using only 16 temporal integer registers. There are no remarkable differences between the different tested issue widths and the two different instruction schedulers employed (TD and BU). However, as expected, BU exerts slightly less register pressure when the number of registers is scarce. Figure 4.13 shows the region coverage when limiting the number of floating-point temporal registers. In this case, the proposal is able to optimize 95% of the code using only 8 temporal floating point registers. As in the case of the integer registers, there are no remarkable differences between the different tested issue widths and neither between the two employed schedulers.

Therefore, with 16 integer temporal registers and 8 temporal floating-point registers, the proposal is able to optimize 95% of the code. Moreover, although the BU scheduler presents slightly less register pressure when the number of temporal registers is scarce, TD scheduler is the preferred option because it presents better performance numbers with similar register pressure.

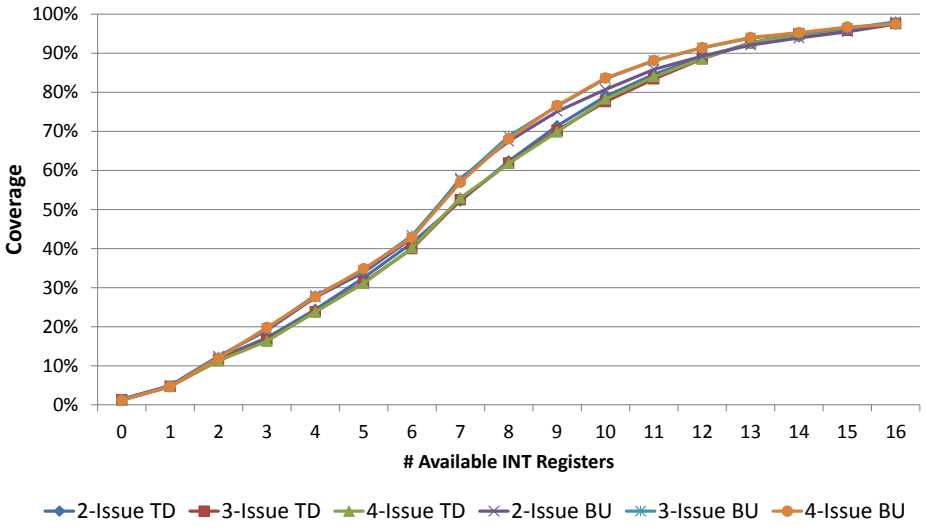


Figure 4.12: Integer Register pressure impact of the software checkpointing proposal.

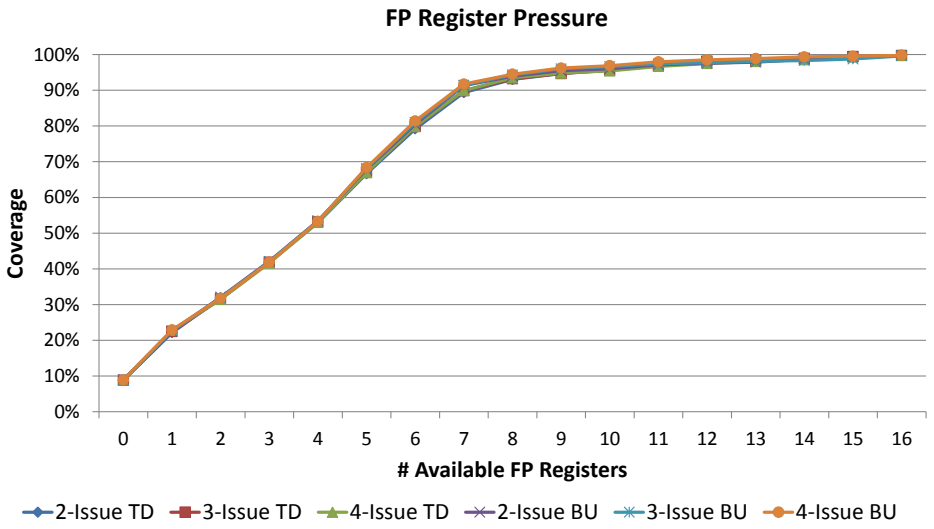


Figure 4.13: Floating-Point Register pressure impact of the software checkpointing proposal.

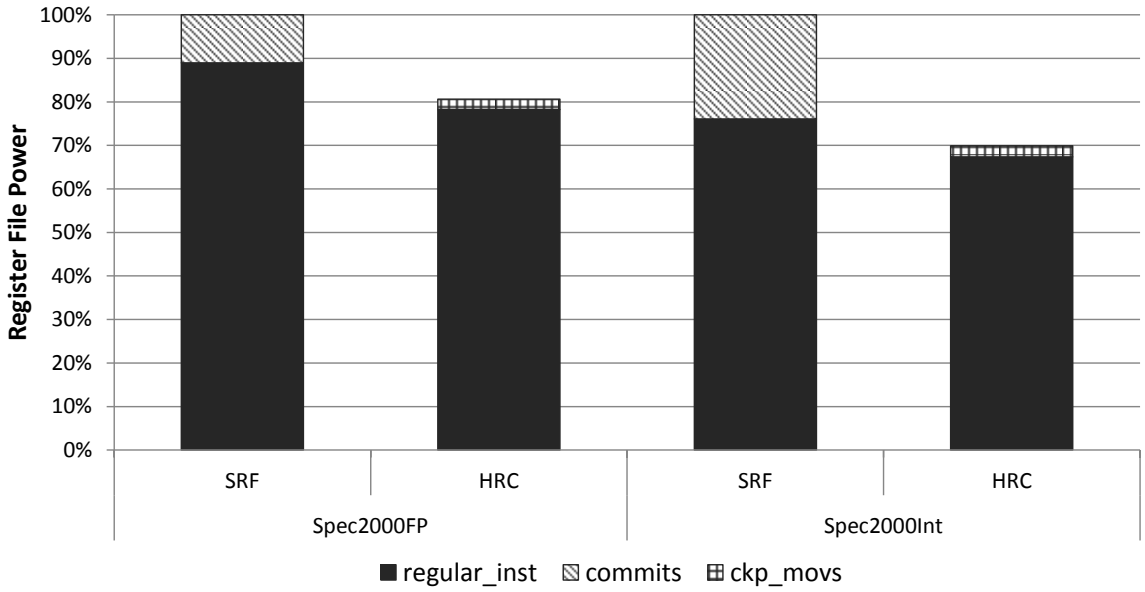


Figure 4.14: Register file dynamic power consumption evaluated with CACTI.

4.5.2.3 Area and Power Estimations

We have employed CACTI [106] to estimate the area and the power consumption of the register file for the HRC and SRF schemes. For these analyses we have considered a register file with 16 integer temporal, 8 floating-point temporal, 16 integer general purpose (architectonic) and 26 floating-point (including x87 and SIMD registers) general purpose registers. In the case of the SRF scheme, we have also considered 16 integer and 8 floating-point shadow registers.

For the area and power estimations, we have modeled an optimized version of the SRF scheme that minimizes the dynamic power consumption of the checkpointing operation. In this design, each cell of the register file contains the original value and the shadow one. This makes possible to access both values by using the same circuitry and therefore no extra access to index the shadow cell is required. In addition, the original value can be copied into the shadow one without exiting the register file. Therefore, the logic related with moving out the values from the register file is not required (sense amplifiers, multiplexors, etc.), which significantly reduces the dynamic power of the checkpointing operation. Finally, the design of the cell allows all register values to be copied simultaneously in a zero-cycle operation (flash copy), which also reduces the possible performance overheads of the SRF scheme.

Figure 4.14 shows the dynamic power consumption of the register files for the HRC and SRF schemes for the Spec2000FP and Spec2000Int benchmarks. Power numbers have been normalized to the total power consumption of the SRF scheme. Moreover, the power consumption is split into two categories. First, the power contribution of the regular instructions accessing the register file, such as memory, arithmetic, and control instructions. Second, the power contribution of the additional extra instructions, such as the CKP_MOVs for the HRC scheme and the commits for the SRF scheme[8]. As shown in Figure 4.14, the cost of executing the regular instructions is lower in HRC. This is so because the register file for the software checkpointing is smaller and therefore, its dynamic power cost per access is lower.

Moreover, HRC also shows an advantage over the SRF scheme since the extra CKP_MOV instructions consume less power than the required flash-copy of the registers. Note that in the SRF scheme, every time a new region is going to be executed all registers need to be checkpointed, no matter if they are going to be modified during the execution of region. By contrast, the HRC proposal only copies the registers that really need to be checkpointed as commented in previous sections. In addition, the size of the regions in the Spec2000FP and the Spec2000Int benchmarks is small (in the order of 30 instructions for Spec2000Int benchmarks and 60 instructions for Spec2000FP benchmarks), which implies a high number of checkpoints. Considering the total power consumption of the register file, the SRF scheme consumes 19.4% and 30.1% more dynamic power than the software checkpointing for the Spec2000FP and Spec2000Int benchmarks respectively. Moreover,

[8] The number of commit instructions executed is equal to the number of optimized regions executed.

SRF commit operations consume more power in Spec2000Int than in Spec2000FP , this is mainly due to the smaller size of the regions found in Spec2000Int benchmarks. As shown in the figure, HRC is less sensitive to the size of the regions.

Finally, given the aforementioned configuration for the register file, the proposed software checkpointing scheme reduces by 11% the area required to implement the SRF scheme.

4.6 Conclusions and Future Work

In order to guarantee precise exceptions and to perform speculative optimizations in a hardware/software co-designed processor, it is necessary to implement a mechanism to correctly handle the architectural and speculative states and to update the former at specific safe points. The solution is to checkpoint the state of the processor prior to execute the optimized region and promote the speculative state to architectural state upon successful execution of the region. In case of an exception or mispeculation, the speculative state is discarded, the architectural state is restored, and execution continues by transferring control to the non-optimized version of the region. In this thesis, we have focused on checkpointing mechanisms for the registers.

We have presented a register checkpointing technique, called HRC, oriented to correctly update the register architectural state of the machine. The proposal does not require important hardware additions and it uses the software combined with the hardware to implement very efficient checkpointing and recovery mechanisms. The presented proposal is a good example of a technique oriented to find a balanced trade-off between high performance and low hardware complexity. In fact, the proposal makes use of already existing processor resources, such as the temporal (non-architectonic) registers that are present in most of nowadays processors. Common hardware proposals use the Shadow Register File scheme that requires doubling the number of registers and the implementation of additional logic to perform the copies among the registers. The proposed mechanism does not require these hardware structures, saving a significant amount of power and area, and it is only 1% to 1.75% slower than the Shadow Register File scheme when both are implemented on top of an in-order 4-issue width processor. Moreover, it is focused only on those registers that are overwritten during the region execution and the recovery code only executes the minimum amount of move instruction to reestablish the architectural state, which significantly reduces the software overheads when compared to other software checkpointing proposals. Finally, the proposed HRC scheme reduces by 11% the area and by 19.4% (Spec2000FP) and 30.1% (Spec2000Int) the power of the Shadow Register File scheme.

The recovery mechanism of our proposal does not penalize the execution of the code region when it is correctly executed, which is the most common scenario. Instead, the proposed recovery mechanism only penalizes the execution of the region when a rollback operation is required. Moreover, the proposed technique does not impact the performance of the non-optimized code regions.

As future work, we emplace the study of the interactions that our proposal may have with other optimization techniques that make use of the temporal registers too. This task is out of the scope of this thesis because it requires developing and analyzing such optimization techniques. It would be good to analyze the possible interactions between our technique and those optimizations because they may impact the final optimized code coverage and hence the performance that could be achieved by optimizing the regions.

Finally, it would also be very interesting to study heuristics in order to improve the register allocation mechanisms to better adapt them to the inserted CKP_MOV instructions within the optimized region.

Chapter 5

Loop Parallelization

The industry and the scientific community have adopted chip multiprocessors (CMPs) and simultaneous multithreading (SMT) as good design points for improving performance while keeping a reasonable power budget [8]. These schemes allow executing multiple processes simultaneously as well as exploiting the parallelism of multi-threaded applications, what is commonly known as thread level parallelism (TLP). However, instruction level parallelism (ILP) is still important for legacy and serial code and processors have become very complex to increase their performance. Thus, we face a design trade-off: dedicate the available area to put a lot of simple cores and exploit TLP or dedicate it to put fewer complex cores and exploit ILP. A very appealing design would be to have several simple cores that can cooperate to some extent to exploit ILP when necessary.

In this chapter, we propose Loop Parallelization (LP), a hardware/software co-designed scheme that uses multiple threads to improve the execution of single-threaded codes requiring very few hardware changes since it makes and efficient use of the already existing processor resources. By means of a software layer, this scheme dynamically detects loops and prepares them to execute multiple iterations in parallel by using Simultaneous Multi-Threading (SMT) threads. Parallel iterations are in turn software layer optimized regions and therefore they are speculative until they commit. Moreover, given that, by definition, iterations are not independent, the hardware keeps track of memory and register dependences to allow communication among them. Actually, the proposed scheme uses memory dependence speculation, and the hardware, in collaboration with the software, is in charge of rolling back the execution of the faulty iterations in case of a memory violation. On the other hand, register dependences are satisfied in a synchronized manner by employing instruction marks and new hardware extensions.

The proposed LP technique has been designed on top of a simple in-order processor based on the Intel® Atom® architecture [107]. The small area and low power consumption of this processor allow us to integrate several of them in the same envelop with the objective of intensively exploit TLP. Moreover, the loss on ILP performance due to the core simplicity is minimized to some extent by implementing the proposed LP scheme. As other hardware/software proposals, LP passes part of the complexity of some hardware components to the software layer. In such a way, it is possible to increment the performance of the processor without significantly incrementing the hardware complexity of the individual units. For instance, in our system, the software layer performs the instruction scheduling and register allocation without requiring dedicated hardware components.

The proposed LP scheme takes advantage from the fact that the full execution width of an Intel® Atom®-like processor (which is 4 instructions) is only used in 28.1% of the cycles[9]. This means that it is possible to issue additional instructions in 71.9% of the cycles, as there are available execution slots. The proposed scheme uses the empty issue slots to accommodate instructions of next loop iterations, parallelizing the execution of instructions of consecutive iterations. Therefore, although instructions within an iteration are executed in order, instructions across iterations may be executed out of order.

Parallel loop iterations are executed in the same core by employing SMT threads and, therefore, all loop execution happens locally. This allows reducing the parallelization overheads and makes possible that even small loops and loops that iterate just a few times get significant performance improvements. In fact, the execution time of the loops in Spec2000Int benchmarks is improved by more than 16% when compared to a fully optimized baseline.

The main contributions of LP when compared to other speculative SMT approaches are as follows:

- **Fast speculative thread start up:** LP makes use of a novel mechanism to start the execution of the speculative thread as soon as the loop to parallelize is detected. This solution minimizes the overheads associated to the thread spawning process and allows parallelizing small loops and loops that iterate just few times. More details about this mechanism in Section 5.3.2.

[9] This number has been obtained by using the infrastructure presented in Section 5.5.1.

- **Fine-grain register communication:** LP employs a hardware/software co-designed mechanism to synchronize the register communications across the SMT threads. This mechanism does not require executing additional instructions and it is implemented by employing minimum hardware changes. More details about this mechanism are given in Section 5.3.4.1.
- **Thread dynamic binding for registers:** LP implements a dynamic binding between registers and threads that avoids moving register values across the different register files on loop finalization. This solution minimizes the loop finalization overheads. The dynamic binding scheme is described in more detail in Section 5.3.6.
- **Dynamic code optimization**: LP has been implemented on top of a hardware/software co-designed processor, where the software layer is able to detect and optimize the regions to be parallelized. Therefore, LP can take advantage from the inherited capabilities of hardware/software co-designed virtual machines. Special optimizations oriented to improve the execution of parallel loops are presented in Section 5.4.

The rest of the chapter is organized as follows. Previous work related to LP is described in Section 5. Section 5.2 introduces the basic principles of the LP scheme. Later, in Section 5.3 the hardware and software mechanisms required to implement LP are described in detail. Additional optimizations for improving loop parallelization are presented in Section 5.4. The evaluation of the proposal is presented in Section 5.5, and we conclude the chapter and expose the future work in Section 5.6.

5.1 State of the Art

Improving the performance of loops has been deeply studied in the literature [60]. Techniques such as hoisting of loop-invariant computations, elimination of induction variables, elimination of null and array-bound checks, loop splitting, and loop tiling are currently considered as classical loop optimizations and are commonly applied by offline compilers. These techniques are focused on improving the ILP of the loops by transforming the original loop codes into more efficient versions of them. All these techniques are orthogonal to the presented LP scheme and can be combined together.

Loop Unrolling, also a classical loop optimization, presents more similarities to the LP scheme than the previous techniques based on code optimization [57] [118]. Loop unrolling is based on repeating the original static instructions of the loop body multiple times, eliminating non-necessary branch instructions, adjusting the loop termination conditions,

and reusing common computations. Besides of the positive effect of removing instructions from the original code, loop unrolling combined with other optimizations can increase the ILP of the application by overlapping the execution of instructions belonging to different iterations [57] [118] [119]. This is achieved, however, at an increase in code footprint. Another classical technique specially designed to improve the ILP of loops is Software Pipelining [120]. This technique improves the performance of loops by overlapping the execution of instructions belonging to different iterations. Actually, Software Pipelining is a type of out of order execution but the scheduling of instructions is done by the compiler. Loop Unrolling and Software Pipelining differ from the presented LP scheme in the fact that they are pure software techniques that are only applied at compilation time and they do not get feedback from the dynamic execution of the code. They can also be combined with the presented LP scheme and/or other classical loop optimizations.

LP scheme has been proposed to improve the performance of loops by exploiting TLP. A significant amount of research efforts in the literature have been devoted to exploit TLP of the applications and they have come up with appropriate design points to deal with ILP and TLP. The techniques more similar to the one presented in this chapter are the ones based on speculative multithreading. These techniques decompose the original sequential application and distribute the execution of the instructions into different threads. On one hand, there are the traditional techniques that decompose the sequential codes into large chunks of consecutive instructions [11] [121] [122] [123] [124] [125] [126] [127] [128] [129]. This coarse grain instruction decomposition may constraint the benefits of this paradigm because code that is hard to parallelize may present too many dependences among threads, limiting the parallelisms that can be achieved. On the other hand, there are techniques like Anaphase that parallelizes applications at instruction granularity [10] [130], which provides more flexibility than previous schemes. In general, these speculative multithreading schemes distribute instructions among threads belonging to different core units and the threads are generated by the software compiler. These two characteristics imply that threads need to be big in order to pay off their creation and management overheads and they need to be co-scheduled at the same time by the operating system to achieve the designed performance.

Speculative multithreading implemented in Simultaneous Multithreading (SMT) processors [131] has also been deeply studied in the literature. SMT processors are able to improve system throughput by overlapping the execution of multiple threads on a single wide-issue processor. Therefore, besides the explicit parallelism that can be exploited by the programmer, these techniques are aimed to exploit the implicit parallelism of the

applications by generating speculative threads. When compared with the aforementioned CMP proposals, threads in speculative SMT compete for the same resources but introduce lower overheads for inter-thread communication [132].

The Thread Multipath Execution (TME) [133] and the Dynamic Multithreading (DMT) [134] proposals create speculative threads that are executed in parallel with the original program instructions by employing hardware heuristics. TME creates the speculative threads on branch miss-predictions whereas DMT does it on call instructions (routines) and backward branches (it assumes all backward branches belong to loops). Both techniques speculate on register and memory communications, and they use value prediction to improve the performance of the communications among the threads. Marcuello et al. [135] proposed a similar hardware based speculative simultaneous multithreading technique specialized in loop parallelization.

These hardware based proposals offer binary compatibility in a total transparent way for the programmer. However, these techniques have a limited scoped for generating the speculative threads. They apply simple code optimizations and they rely on value prediction to improve inter-thread communications. Implicitly-Multithreaded (IMT) [136] was proposed as a solution to overcome the problems of hardware based solutions by employing the software compiler. In IMT, the compiler is in charge of selecting the best code points to generate the speculative threads (a.k.a. spawning pairs). Moreover, it reduces the inter-thread register dependences and avoids inter-thread control-flow mispredictions by enclosing both "if" and "else" paths within the thread code. However, memory communications are serialized to guarantee correct execution. Finally, IMT introduces a fast thread spawning startup technique aimed to reduce the overheads of initiating the speculative threads (mostly because of the required register renaming initializations).

Venkatesan et al. [137] proposed a similar solution to IMT but in this approach register dependences are satisfied in a synchronized manner. The technique to synchronize the register communication among threads is based on the software solution described in [138]. Moreover, this technique makes use of a cache based scheme to further speculate in memory communications, which allows incrementing the size of the speculative threads. This proposal is very similar to LP although it does not make use of the dynamic optimizer and it uses a coarser grain register communication solution.

Finally, Dual-thread Speculation [132] shows that by employing very few hardware resources it is possible to significantly improve application performance. In fact, they demonstrated that most of the parallelism exploited by an 8-way CMP processor could also be exploited with their simple dual-thread model based on a 2-way SMT processor.

The LP scheme is a type of speculative multithreading scheme that takes the principles of traditional multithreading schemes (it works with chunks of instructions) but it generates code for hardware thread contexts implemented within the same core unit (SMT threads). Executing the parallel codes in the same core allows creating light threads even for small loops. Moreover, LP scheme is a co-designed hardware/software technique where threads are generated during the dynamic execution of the application (online) and it can take advantage from analyzing the dynamic information of the programs running on the processor to better adapt the codes to the hardware underneath. Finally, the same code is used to represent the sequential version and the parallelized version. Thus, if the other SMT context is available, the second thread is spawned. Otherwise, the loop is executed serially.

5.2 Loop Parallelization

The objective of the LP scheme is to overlap the execution of two iterations in an in-order hardware/software co-designed processor by fully utilizing all its resources. The processor needs to support at least the execution of two simultaneous threads in order to distribute the loop iterations among them (one iteration executed by each thread). Moreover, the software layer is in charge of detecting the loops that are more appealing to be optimized, as well as preparing them for their correct execution in the hardware. The software layer may also decide to optimize the code (even by reordering memory instructions) if it is necessary to improve the parallelism of the iterations. Additional information is included in the loop instructions in order to guide the hardware when executing them and special hardware structures are included to guarantee the correctness of the execution. In principle, threads are not expected to communicate through memory, although hardware mechanisms designed to keep track of memory accesses are considered to detect possible memory dependence violations. Moreover, threads communicate through registers in a synchronized manner by having access to the register values produced by the other threads.

The rest of this section is organized as follows. First, we introduce the requirements that loops have to accomplish to be parallelized. Later, we describe in detail the different hardware and software mechanisms required for parallelizing the execution of the loops.

Finally, we present different optimizations to be applied on top of the original loop codes in order to improve their performance when used in the LP scheme.

5.2.1 Code Regions to Parallelize

Almost all application time is spent executing loops of instructions. In particular, loops that do not include other nested loops represent a significant part of the dynamic code execution [139]. These loops are commonly known as inner-most loops and by optimizing them it is possible to improve the overall execution time of the application. Optimizing bigger code regions, such as outer-loops, is more costly as the overheads associated with analyzing and optimizing complex control flows increase.

The software layer uses inner-most loops as code region candidates to be parallelized by the LP scheme. The hardware monitors the execution of the code (with mechanisms similar to the ones explained in Chapter 3) and invokes the software layer when hot code is detected. The software layer then builds a superblock (SB) starting at the specified hot address. If it detects an inner most loop inside the SB and the loop is suitable to be parallelized, it is optimized and stored into the code cache for later reuse. Note that, although SB regions may have multiple exit points, only one entry point is allowed. This entry point serves as the loop identifier for the hardware and the software layer.

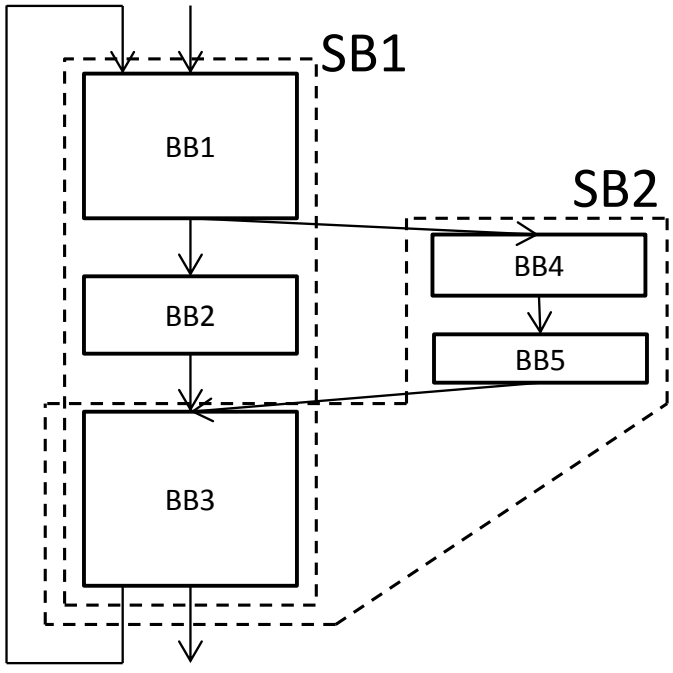


Figure 5.1: Example of a loop region respresented by using super-blocks.

In Figure 5.1, there is an example of a loop represented by using superblocks. As it can be observed, there is a complex control flow consisting of 5 basic blocks (BB). The starting point of the hot region is the address of the first instruction of BB1. The software layer then builds a SB starting at this address and it continues adding BBs following the most probable path until a finalization criteria is met. In the example, we have assumed that the path BB1-BB2-BB3 is the most common one and such a SB is created. If later BB4 becomes hot, the software layer will create a second SB consisting of BB4, BB5 and a replica of BB3. Such a replica BB is required because SBs consist of a control flow with single entry points and potentially multiple exit points as previously commented. In this particular example, SB1 is detected as an innermost loop by the software layer as it contains a loopback branch from BB3 to BB1. Hence, it will be considered for loop parallelization. Note that although SB2 forms part of a complex innermost loop, it is not detected in isolation and the software layer is not able to parallelize it in consequence.

The reasons to finalize the construction of a SB are the common criteria found in the literature, which are as follows:

1. The SB cannot contain more than a maximum number of instructions (in our case the limit is 64 instructions).
2. The SB cannot contain more than one indirect branch and if so, this must be the last instruction of the SB. Note that exceptions are allowed since they are considered as return statements with paired call instructions.
3. The SB must have a higher probability than a given threshold (in our case 80%) to be exited by the last BB. This probability is based on profiling information.
4. The SB cannot contain instructions that have execution restrictions in the processor pipeline [113], such as changes to segment registers, special move instructions, etc. We call these instructions not to be in the correct format.

Given the aforementioned constraints, not all inner-most loops of an application may end being parallelized by the proposed technique and, moreover, not all instructions of them may form part of the region to be parallelized. For instance, as previously commented, the inner-most loop of Figure 5.1 cannot be completely parallelized. In fact, only the dynamic instances of the SB1 instructions can be parallelized. This means that the LP will end on every transition to SB2 instructions and also every time that the back-edge of BB3 is not taken.

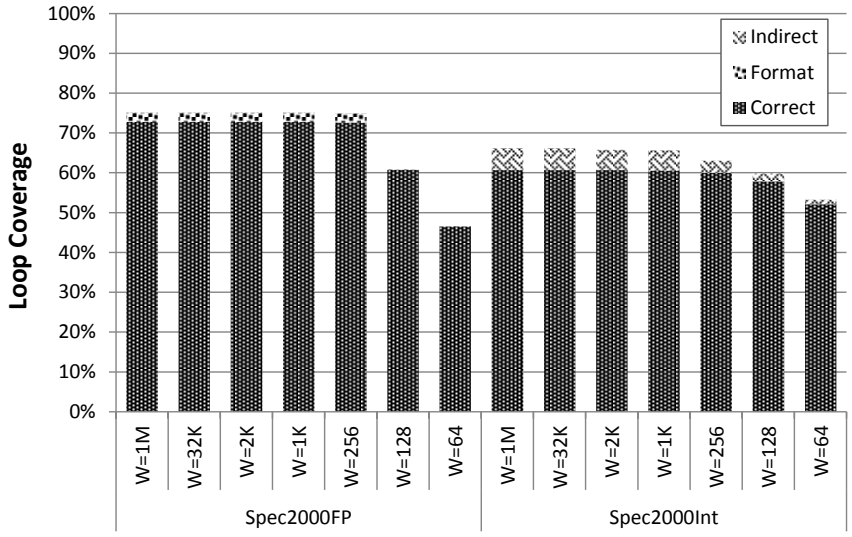


Figure 5.2: Inner-most loops dynamic code coverage.

We have analyzed the coverage of the inner-most loops in the Spec2000 benchmarks by using an off-line analysis tool. This tool analyzes the application dynamic stream execution and counts the number of instructions that belong to the inner-most loops. The loops are simply detected by tracking branch instructions whose target is an older instruction in the dynamic stream and between this older instruction and the current instruction there are no other loops. This process is done by truncating the dynamic stream into fixed windows of dynamic instructions ranging from 64 to 1024 dynamic instructions. In Figure 5.2, we show the percentage of the dynamic stream that belongs to the inner-most loops computed by using the tool (a.k.a. code coverage of the inner-most loops). In the figure, the x-axis shows the benchmarks separated as floating point (Spec2000FP) and integer (Spec2000Int). Moreover, in the same axis there are the different dynamic window sizes used in the study. The y-axis represents the loop coverage in percentage. The bar named as *Correct* is the actual percentage that can be achieved (coverage of the correct loop regions satisfying all the constraints), while the bar named as *Format* shows the coverage lost by not including special x86 instructions (the aforementioned incorrect format instructions). The bar named as *Indirect* shows the coverage lost by not including indirect branches in the loop regions (except for paired call/return instructions). As it can be observed, the coverage of the inner-most loops is 72.81% for Spec2000FP and 60.7% for Spec2000Int benchmarks when using a dynamic window of 1024 instructions. However, the coverage is reduced to 46.5% and 52.14% for Spec2000FP and Spec2000Int benchmarks respectively when using a dynamic window of 64 instructions. The main reason is that when limiting the dynamic window size some

loops are not detected and they do not contribute to the coverage. Note that some loops are discarded because of format constraint in Spec2000FP, whereas some loops are discarded because indirect jumps in Spec2000Int. In both cases the coverage loss is low and always below 5.5%.

Starting and stopping the execution of an LP loop requires additional tasks to be done by the processor (see Section 5.3.2). These tasks introduce an additional overhead to the system. Therefore, if the loops do not iterate a certain number of times the overheads will have a significant impact and overall performance may not improve (it could even degrade). We have enhanced the analysis tool to detect loops, build super-blocks, and compute the number of iterations of the inner-most loops. We have employed Spec2000 benchmarks and we show the results in Figure 5.3. In x-axis of the figure, we show the average number of iterations discretized by using the following intervals: less than 5, between 5 and 10, between 10 and 50, between 50 and 100, between 100 and 1024, between 1024 and 2048, between 2048 and 4069, and more or equal than 4096 iterations. In the y-axis, we show the number of Spec2000 simulation traces that satisfies that their inner-most loops iterate on average the number of times reflected in x-axis. A total of 59 traces have been used for this experiment. As it can be observed, 12 traces have inner-most loops that on average iterate less than 5 times. However, there are 10 traces with inner-most loops that on average iterate between 100 and 1024 times. In fact, 76% of the traces have loops that iterate less than 1024 times (45 from a total of 59 traces). Since there are a significant number of loops that iterate just a few times, the proposed LP scheme has been developed very efficiently to introduce minimal overheads when spawning the threads of a parallel loop. Otherwise, the cost of enabling loop parallelization for those loops would not be amortized by the possible benefits that could be obtained by their parallelization.

We have also evaluated the coverage of using hyper-blocks [140] for representing the loops instead of superblocks using the same tool. Hyper-blocks allow multiple control flow paths within the regions at the expense of added complexity to the software layer and/or the hardware to generate efficient parallel code. Since we did not observe a significant increase on coverage and neither on the average number of iterations, we discarded hyper-blocks and continued the work using superblocks.

5.2.2 Potential Numbers of the Regions to Parallelize

In this section, we evaluate the upper-bound performance that can be achieved by parallelizing the execution of the aforementioned inner-most loops. As in previous studies,

we have considered the Spec2000 benchmarks and the offline analysis tool. When memory dependences between consecutive iterations are not taken into account, the estimated performance results for LP scheme show speedups of 1.07x, 1.17x, 1.29x, and 1.36x in loops for the Spec2000FP benchmarks when employing a 2-way, 3-way, 4-way, and an unbounded issue width in-order processor. For Spec2000Int benchmarks, the speedups are 1.17x, 1.31x, 1.37x and 1.41x, for 2-way, 3-way, 4-way, and unbounded issue width respectively. These numbers are reflected in Figure 5.4, in which speedups are shown on the y-axis and plotted as bars. Note that better results are obtained with wider machines because there are more issue slots available for the next iteration.

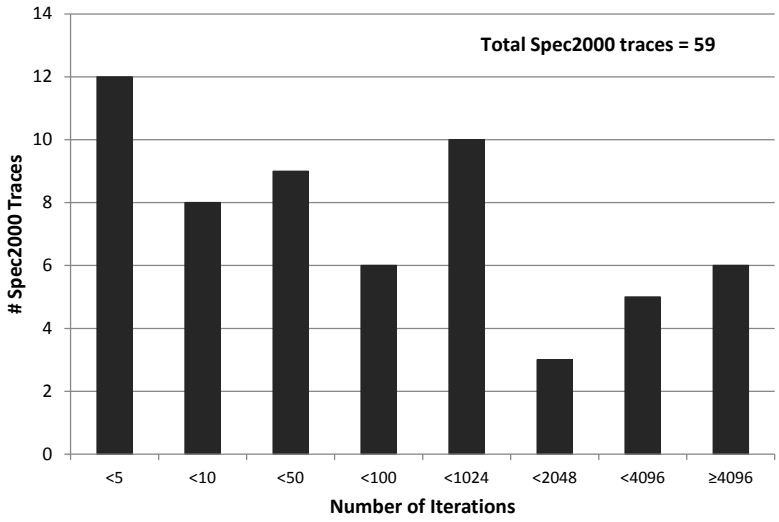


Figure 5.3: Average number of inner-most loop iterations in Spec2000 benchmarks.

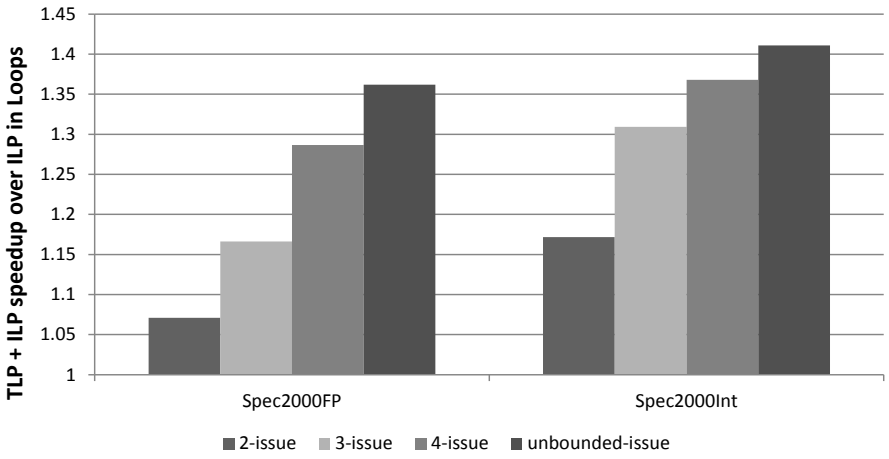


Figure 5.4: Estimated performance for loop parallelization in loops.

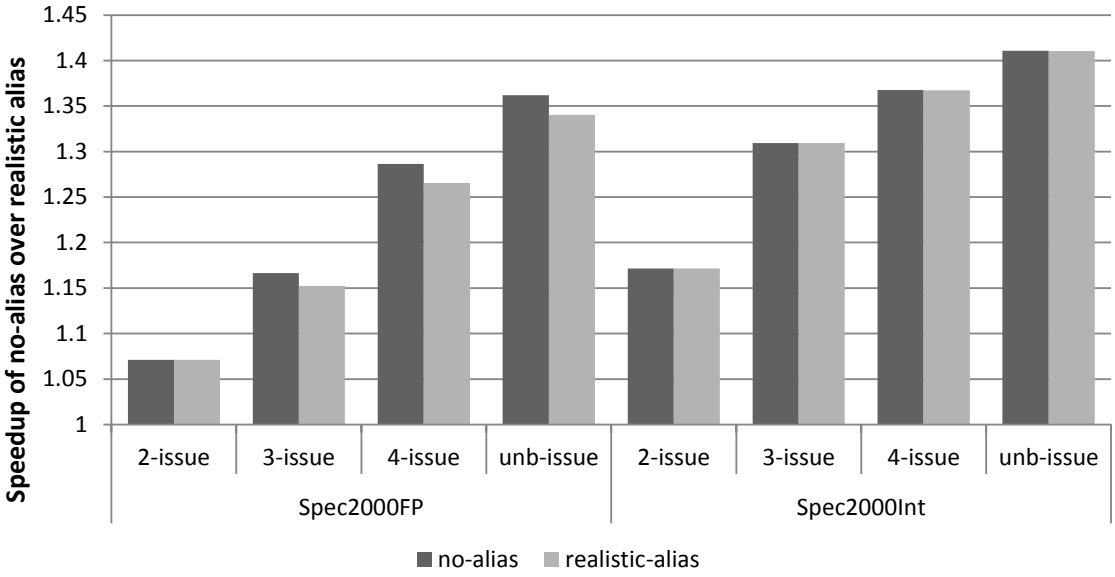


Figure 5.5: Impact of memory communications between instructions belonging to consecutive iterations.

The impact of memory communications between instructions belonging to consecutive iterations has also been evaluated. In this next study, we compare the previous estimated speedups (where memory dependences[10] are not considered) and the estimated speedups when a memory dependence between two consecutive iterations incurs in a thread squash. We call to the former as no-alias[11] and to the second as realistic alias. Figure 5.5 shows these speedups for Spec2000FP and Spec2000Int benchmarks. The slowdown of realistic-alias is lower than 2% for Spec2000FP and negligible for Spec2000Int benchmarks for all the issue width bandwidth configurations. Note that slowdowns for the realistic alias are larger on wider machines because there are more chances for reordering memory instructions. Such a small performance indicates that a simple eager squash policy is sufficient and we do not need to design fancy synchronization mechanisms in memory or other hybrid techniques.

The presented potential numbers indicate that it is possible to improve application performance significantly by employing the LP scheme, even when considering the dependences across iterations. However, the design should be very efficient to do not introduce big overheads and to do not penalize loops that iterate only a few times.

[10] Memory dependences come in three flavors: Write After Write (WAW), Write After Read (WAR) and Read after Write (RAW) [152].

[11] Alias comes from aliasing.

In the Section 5.3.4 we analyze these numbers again by using an enhanced version of the simulation tool and we provide insights on why a 2x speedup is not achieved.

5.3 Loop Parallelization Scheme Implementation

This section describes the different mechanisms required to implement the LP scheme on top of an in-order processor built in combination with a hardware/software co-designed virtual machine. The required mechanisms strongly depend on the particular architecture of the processor, so we start this section by describing the characteristics and the pipeline of the core. Once the processor architecture is presented, we continue describing in detail the implementation and the functionality of the new required support for LP. In particular, the introduced mechanisms are the following: thread spawning, register communication, memory communication, iteration finalization, loop finalization, and exception handling techniques. This section ends with a brief summary of all the required hardware and software modifications for supporting the technique and it provides a general overview of the processor once all the techniques are implemented on top of it.

5.3.1 Baseline Core Characteristics

The core employed for implementing the LP scheme is based on the Intel® Atom® processor [107]. This is the same processor employed in the Chapter 4 (Hybrid Hardware/Software Register Checkpointing) and its characteristics have already been presented in Section 4.3.

The Intel® Atom® is very attractive for implementing the LP scheme because it supports native simultaneous multi-threading (SMT) execution, where two internal threads execute independently but sharing common resources. By means of the hardware/software co-designed virtual machine, the software layer may easily adapt the code of single thread applications to take advantage of the two threads by parallelizing the execution of particular loops.

5.3.2 Thread Spawning

Thread spawning refers to the action of creating a second thread to be executed into the processor. In LP, thread spawning is used to generate new threads that in conjunction with the original one will parallelize the execution of a particular loop. The original thread is also known as the spawning thread because it is in charge of creating the other thread, whereas the new non-original threads are also known as the spawned threads. The execution of the loop is distributed among the different threads, which execute the different iterations that comprise it. In particular, in a two threads scenario, the odd

iterations of the loop are attached to one thread and the even iterations to the other. Note that the proposal described in this document is focused on two threads only, although we propose some extensions to support more threads as a future work.

The software layer is the responsible for deciding when to start a new thread, and hence it is in charge of initiating the loop parallelization process. It does this by communicating to the front-end of the processor the initial PC (Program Counter) of the thread to be created by means of executing a new instruction "*Spawn PC@*"[12]. In the meantime, the processor continues fetching instructions from the original thread. However, there is a delay between the creation of a thread and the actual dispatch and execution of its instructions since the *spawn* is processed once it is executed and there are several pipeline stages between execution and fetch. Figure 5.6 describes this problem graphically where the original thread, in the left of the figure, initiates the process at the Begin Of the Loop (BOL, that is the first dynamic instruction of the loop) instruction point. The instructions of iteration 1 in the spawned thread require some cycles to reach execution. In the meantime, the original thread continues executing instructions of iteration 0. The delay before executing instructions of the spawned thread may produce that subsequent iterations execute in an unbalanced manner, making one iteration to start too late with respect to the other. All the unproductive time waiting to execute instructions of the spawned thread is called spawn bubbles. In Section 5.5.2, we analyze in detail what is the performance impact of the spawn bubbles on the execution performance of the parallelized loops.

To understand better the impact of spawn bubbles, we show in Figure 5.7 how the first instructions of a loop parallelization process are executed in an Atom®-like processor. On the left of the figure, we show the involved instructions. They are identified by the thread *id* and by the iteration number they belong to. On the right part of the figure we show the pipeline stage each instruction occupies at a given processor cycle. Note that thread 0 starts the loop parallelization (*Spawn* instruction) and it executes the instructions belonging to iteration 0. On the other hand, thread 1 is the spawned thread and it starts executing the instructions of iteration 1. As shown in the figure for instruction 1 of iteration 0, after the decode stages the spawn signal is sent to the IF1 stage to start fetching instructions for the spawned thread. The first instruction from iteration 1 is

[12] Note that in the case of loop parallelization, both threads share the same code so there is no need to specify explicitly a starting address for the other thread. That is, the other thread starts fetching instructions at the PC address right after the SPAWN instruction. However, we decided to leave the PC explicit for future extensions to overlap the execution of code not in loops.

fetched at cycle 6 and it reaches the SC stage at cycle 12. This is 6 cycles later than the first instruction from thread 0. Moreover, thread 0 has already 7 instructions in-flight (from inst1 to inst7 of iteration 0) when thread 1 starts fetching instructions of iteration 1. In the right part of the figure, we highlight with a double-ended red arrow that the first committed instruction belonging to thread 1 occurs at cycle 21, whereas thread 0 has committed already 6 instructions and it is committing instruction 7 at this same cycle. This is what we call unbalanced work. This problem is exacerbated in loops with few instructions that iterate few times because unbalanced execution has an important impact on performance.

In order to remove the bubbles introduced by the spawning thread process, we have adapted the front-end of the processor to start fetching instructions as soon as possible. The solution consists of marking the thread spawning instructions with a special flag. A pre-decoding hardware logic implemented in the IF1 stage of the processor is able to identify the mark and, therefore, to start fetching instructions for the spawned thread just at the next cycle. By doing so, we are able to reduce the aforementioned 6 cycle bubble to just 1 cycle. Moreover, since both threads execute the same code, the spawned thread will only miss in the instruction cache when the spawning thread misses, generating a balance flow of instructions for both threads. As later shown in Section 5.5.2, a significant number of loops are beneficiated by this solution.

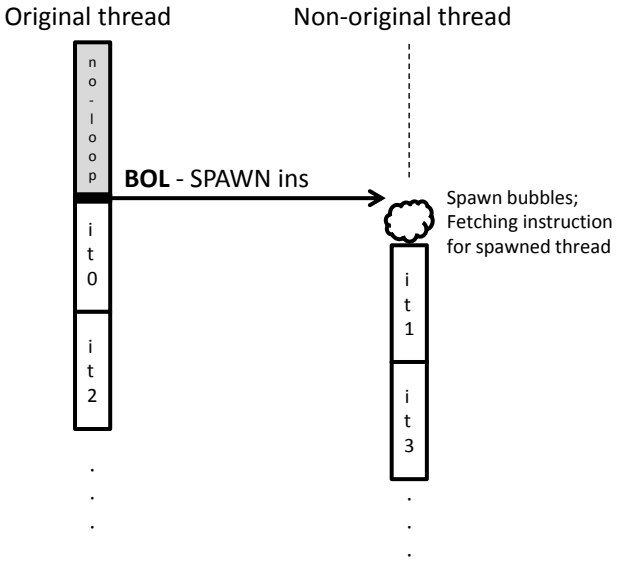


Figure 5.6: Loop parallelization thread spawning.

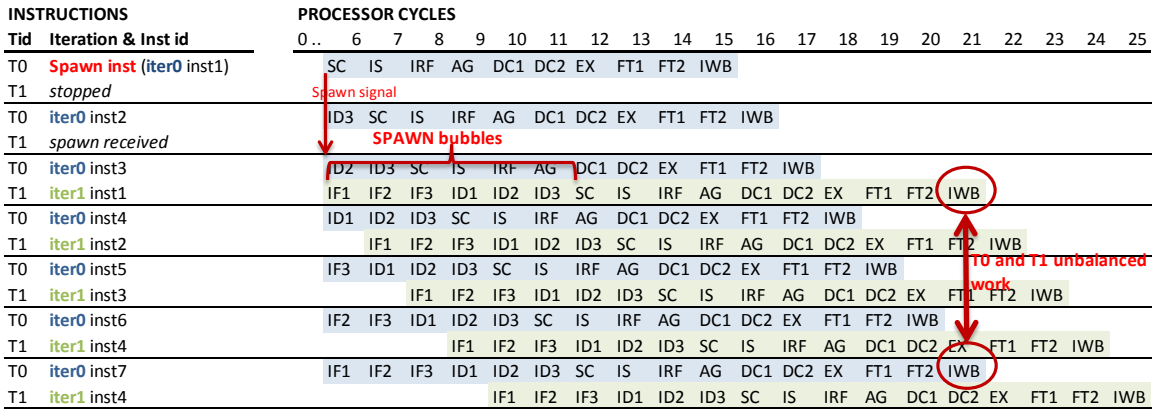


Figure 5.7: Spawn signal bubbles example by using an Atom®-like processor pipeline.

We have also analyzed a second alternative for reducing the spawning bubbles by copying instructions from the original thread Instruction Queue (IQ) to the IQ of the spawned thread. This is a one cycle operation, so that it mimics the behavior of the aforementioned proposal. However, this solution was discarded because of its higher hardware complexity when compared with adding the pre-decoder component at the IF1 stage.

Iterations in LP are handled in hardware as regular regions optimized by the software layer. So that, at the moment of the spawn, a checkpointing of the register file is done and the gate barrier of the store buffer is set. Additionally, the identifier of the hardware thread context initiating the parallelization is stored in a special register, and the iteration counter register is initialized to 0. This counter is used to keep track of the total number of executed iterations and, as we will describe later in the Section 5.3.4.1, it is necessary to guarantee the correct register thread communication that may occur during the execution of the first iteration of the loop.

Finally, when there are no free contexts for spawning a new thread then the software layer executes the loop serially by employing only one thread context. Note that the same code is used to represent the sequential and the parallelized version.

5.3.3 Iteration Ordering

In LP, the code of an iteration is handled as an atomic region by the software layer: either all instructions belonging to the iteration retire or none does. Therefore, if a problem occurs during the execution of the iteration, then a rollback process is initiated and all the work done by it is squashed. The architectural state of the processor is then recovered from the saved state before the execution of the iteration in order to continue execution

from the last correctly executed iteration. Thus, the architectural state of the machine has to be saved after every successful iteration execution.

Moreover, consecutive iterations have to finalize in order to respect the original program order. In other words, iteration *i+1* cannot finalize before iteration *i*, otherwise it is not possible to correctly restore the architectural state of the machine if region *i* fails because iteration *i+1* has speculatively updated it. A special register in the processor keeps track of the total number of executed iterations. This counter is incremented on every iteration finalization.

Since LP overlaps the execution of consecutive iterations and iterations as a whole are committed in strict program order, one thread executes iteration *i* in a non-speculative manner while the other thread executes *i+1* in a speculative manner. In other words, one thread is speculative while the other is not and such behavior ping-pongs between the two threads. Figure 5.8 shows a simple example of a loop parallelized by using two threads that serves us to illustrate better this speculative concept. The speculative instructions are highlighted in grey color. As it can be observed, all instructions in iteration 0 are non-speculative whereas the first instructions of iteration 1 are speculative. However, when iteration 0 successfully commits, then the older not committed iteration in the processor is iteration 1, and its instructions are now non-speculative. Note that at this moment, instructions of iteration 2 in the original thread are speculative. As we will describe later, in Section 5.3.4.2, the concept of speculative thread is crucial for guaranteeing correct thread memory communications. In order to identify it, the speculative thread *id* is always stored in a special register of the processor that is initialized to the non-original thread *id* value at the moment of the spawn.

Threads do not have to be attached to any particular hardware context of the processor. If the original thread is being executed in hardware context 0 at the moment of the spawn, then the non-original thread is executed in hardware context id 1, and vice versa. In fact, the thread *id* is not relevant for the execution of the iterations, what matters is the identification of the thread that initiated the loop and the identification of the spawned thread. The hardware uses a special register that stores the thread *id* of the original thread. In a two hardware context processor, identifying the original thread is enough to also identify the non-original thread. Additional registers for identifying the non-original threads are required in case of having more than two threads.

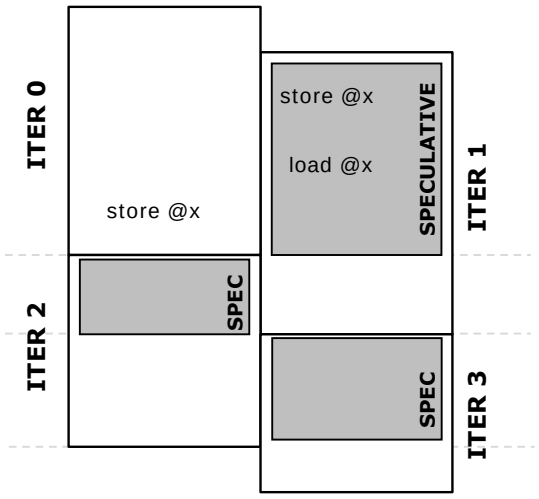


Figure 5.8: Speculative thread concept in loop paralellization.

5.3.4 Communications

The values produced during the execution of one iteration may be consumed later by instructions belonging to the same iteration or by instructions of younger iterations. Therefore, the proposed technique must ensure that the values are correctly communicated across iterations. Values need to be communicated correctly when there is a dependence between a producer instruction and one or more consumer instructions from another iteration. There are two different types of value dependences (a.k.a. communications). On one hand, there are the register dependences, where the communications are done through the register file. On the other hand, there are the memory dependences, where communications are done through the memory hierarchy. Register and memory dependences must be correctly tracked and solved, otherwise the execution of the application will not be correct from a semantic point of view.

In order to study the impact that communications have on the parallel execution of the loops, we have done a simple study that employs the aforementioned off-line tools to understand why the technique does not achieve a 2x speedup when using a 2-way SMT processor with unbounded issue width. In this study, we have also considered two more limiting factors besides the communications. The first is the overhead due to workload unbalance that happens when the loop iterates an odd amount of times and the original thread executes one more iteration than the spawned thread. The second is the overhead introduced to correctly handle the live-outs produced at the end of the loop (see Section 5.3.6 for more details). Figure 5.9 compares the speedup of parallelized loops versus their sequential execution. The legend of the figure is as follows: *real* is the obtained speedup, *memory* is the speedup that can be achieved by not taking into account memory

dependences, *registers* is the speedup achieved by removing register dependences, *finalization* is the speedup when there are no overheads for loop finalization, and *numiters* is the speedup achieved by not having iteration workload unbalance. Assuming regions execute atomically, register dependences between consecutive iterations are the main limiting factor, which reduce the potential speedup by 35% in Spec2000FP and by 25% in Spec2000Int. The second limiting factor is workload unbalance with 12% and 18% for Spec2000FP and Spec2000Int respectively. Memory communications account for an extra 6% and 1% slowdown for Spec2000FP and Spec2000Int respectively. This figures highlights that register communications is the main factor limiting the performance.

Therefore, we have adopted two different approaches for tracking explicitly dependences across iterations. On one hand, since register communications are very common, we have implemented a solution that synchronizes the execution of the threads in order to guarantee correct communication. On the other hand, since memory communications are not so common, we do not explicitly synchronize the threads (we assume there will not be memory dependences between the threads) but we track memory accesses in order to detect memory violations. In case there is a memory violation then the faulty thread is squashed. In the next two sections, we explain these two solutions in more detail.

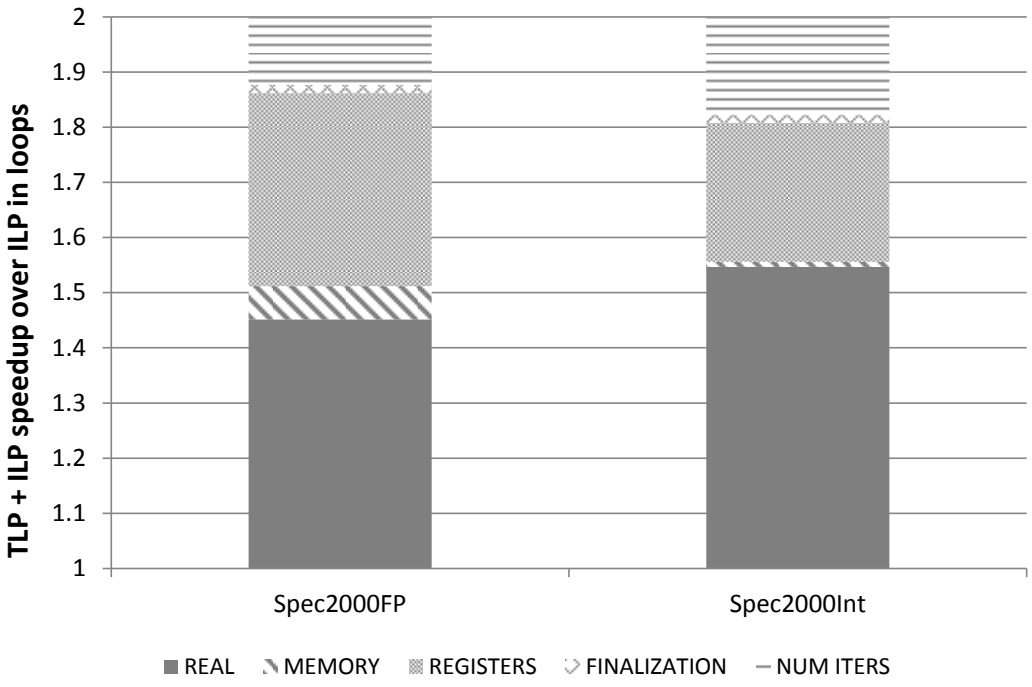


Figure 5.9: Reasons why a 2x speedup is not achieved with speculative loop parallelization using Spec2000 benchmarks.

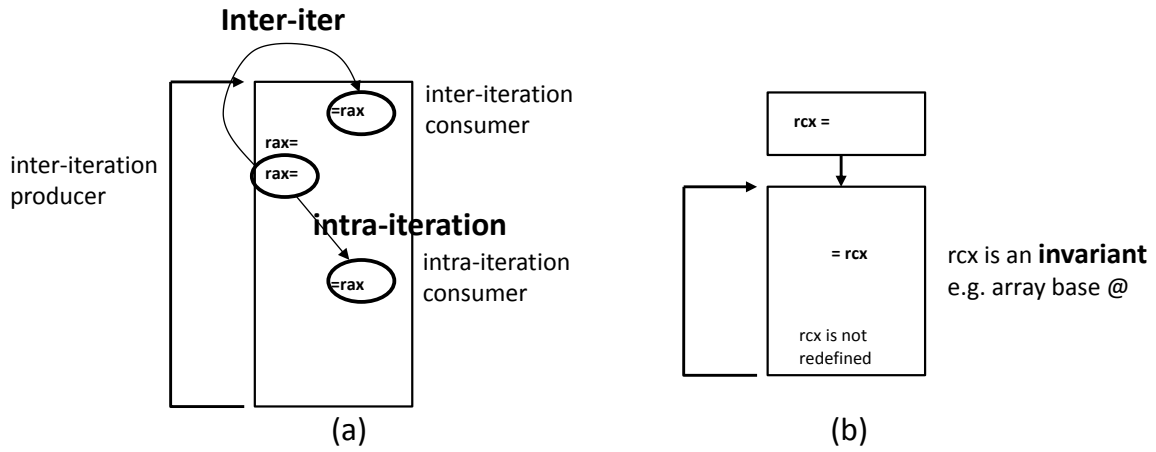


Figure 5.10: Register dependences in loop parallelization.

5.3.4.1 Register Communications

The register dependences in LP are satisfied by synchronizing the execution of the iterations. One consumer instruction cannot execute until its producer instruction generates the corresponding value. In order to synchronize the execution, the software layer tracks the register dependences among the instructions within the loop. It classifies the register dependences as follows:

1. **Invariant dependence:** the value is generated before the execution of the loop and it is not modified during the execution of the loop (the initial value does not vary). The consumer is an instruction of the loop whereas the producer is an instruction previous to the loop.
2. **Intra-iteration dependence:** the value is generated and consumed within the same loop iteration. The producer and the consumer belong to the same iteration.
3. **Inter-iteration dependence:** the value is generated in a previous iteration than the one where it is consumed. The producer belongs to an older iteration than the consumer. Note that since our loops are formed within a superblock the inter-iteration distance is always one. In other words, there cannot be a register value produced in iteration i and consumed in iteration $i+n$ being $n>1$.

We use Figure 5.10 to better explain the three types of dependences. In the left part of the figure (figure a), there is an example of an inter-iteration dependence and another one of an intra-iteration dependence. The intra-iteration dependence occurs because there is one instruction producing a value stored in register RAX, and later there is another instruction within the same iteration that is consuming it. The inter-iteration dependence

occurs because at the beginning of the iteration there is an instruction consuming the value stored in the same register RAX. After the first iteration this value is always generated by an instruction belonging to the immediately previous iteration. This is commonly called a loop-carried dependence and it has always at least 1 consumer before the producer within the iteration. In the right part of the figure (figure b), there is an example of an invariant dependence. In this case, the value generated and stored in RCX register is not modified during the execution of the loop, though it is consumed in all iterations by one static instruction that reads the RCX register.

Therefore, inter-/intra-/invariant consumers and inter-/intra- producers of the loop are key instructions to be clearly identified. Inter-iteration consumers are the first consumer instructions of a register before the register is redefined within the loop body, while the inter-iteration producer is the last producer instruction of the register. On the other hand, intra-iteration consumers are the consumer instructions of a register following the definition of the register in the same iteration, while the intra-iteration producer is the previous producer instruction of the register being consumed in the same iteration. Invariants can be detected by instructions that consume a value that is not defined within the loop. The algorithm used to identify these types of dependences is shown in Figure 5.11.

```
Build registers sets: INVARIANT, INTER, INTRA
For each register in set INVARIANT
      - Set  to 1 the INV bit associated with the corresponding source operands of
        the consumers instructions
For each register in INTRA
      - Set to 0 the INV and INTER bits associated with the corresponding consumers
      - Set to 0 the INTER bit associated with the corresponding producers
For each register in INTER
      - Set to 0 the INV bit associated with the corresponding consumers
      - Set to 1 the INTER bit associated with all the consumers that are
        sequentially before the 1st redefinition of the register → the rest
        (posterior consumers) are marked with a 0
      - Set to 1 the INTER bit associated with the last producer instruction of the
        register → the rest of producers are marked with a 0
```

Figure 5.11: Algorithm used by the software layer to mark producers/consumers properly.

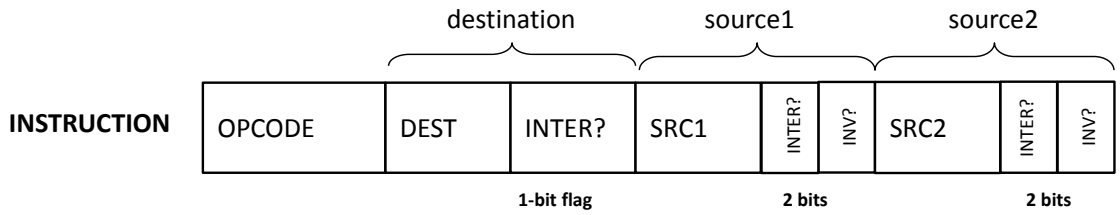


Figure 5.12: Loop parallelization instruction encoding for register dependences.

The loop instructions produced by the software layer are encoded with additional information in order to keep track of the different dependences. As shown in Figure 5.12, each instruction is marked with 5 additional flags besides the opcode and the destination and source register identifiers. One of the flags indicates if the instruction is an inter-producer instruction. This flag is associated to the destination register of the instruction. Moreover, the source operands are also marked to indicate if the instruction is inter-consumer or invariant-consumer for the registers that it consumes. Note that in this case 2 bits (inter and invariant bits) per source register are required.

Since each thread is executed in an SMT context (with its own register file), and register dependences are marked by the software layer, we extend the scoreboard with a simple technique to allow one thread to access the register file of the other in a synchronized manner. Actually, the register file of the original thread initiating the loop parallelization process holds all values before the loop starts[13]. By contrast, the register file of the spawned thread will only contain intra-iteration and inter-iteration produced values. In this case, invariant values will be read directly from the register file of the original thread. Therefore, when executing in loop parallelization mode, the intra-iteration values are produced and consumed locally (in its own register file) within each thread, the inter-iteration values are produced locally (in its own register file) per thread and consumed remotely (from the other register file) with synchronization, and the invariant values are read locally in the original thread and remotely in the case of the spawned thread. Note that all register writes happen locally and register reads may be local or remote.

As commented before, registers are communicated through a scoreboard hardware structure designed to synchronize the reading of the values among the two threads. As depicted in the Figure 5.13, the scoreboard is split into two equal parts, each one for each thread. The registers in the scoreboard have one ready and one sync flag per thread. The ready flag indicates to the thread that the value of the register is ready and it can be read from its own register file, whereas the sync flag indicates that the value of the register is ready and it can be read from the register file of the other thread. If there is a dependence that is not satisfied (may be the local or the remote) then the affected thread cannot continue executing instructions until the dependence is resolved as indicated by the scoreboard, which will check the ready or sync flags depending on the mark associated within its source operands (inter-iteration, intra-iteration, invariant). Note that original

[13] This is not 100% true as will be explained later in the loop finalization section. In reality, values are spread between both register files even before the loop starts its execution. However, we use this simplified version now for clarity purposes.

and spawned thread concept does not match with the local and remote concept. The operational mode is as follows:

- Intra-iteration producer instructions set the ready bit of the corresponding register in the local thread of the scoreboard. This indicates that the value can be read by a local consumer.
- Inter-iteration producer instructions unset the sync bit of the corresponding register in the local thread and set it in the remote one. Unsetting the value in the local thread when executing iteration i is necessary to guarantee correct synchronization. Otherwise, the local thread may read from the remote thread the value generated by the remote thread at iteration *i+1* when executing iteration *i+2*.
- Intra-iteration consumer instructions are stalled until the corresponding ready bit is set in their corresponding local thread register. The values are read from their own register file.
- Inter-iteration consumer instructions are stalled until the corresponding sync bit is set in their corresponding local thread register. In this case, the value is read from the remote register file.
- Inter-iteration consumers of the first iteration in the original thread are handled in a special manner. This situation can be detected because the identifier of the original thread and the number of iterations executed (in this case 0) are stored in special registers upon thread spawning. This case is handled as regular intra-iteration dependence by reading the value from the local register file as soon as the local ready bit is set.
- Invariant consumer instructions are stalled until their corresponding ready bits of local scoreboard are set. The values are always read from the original thread register file.

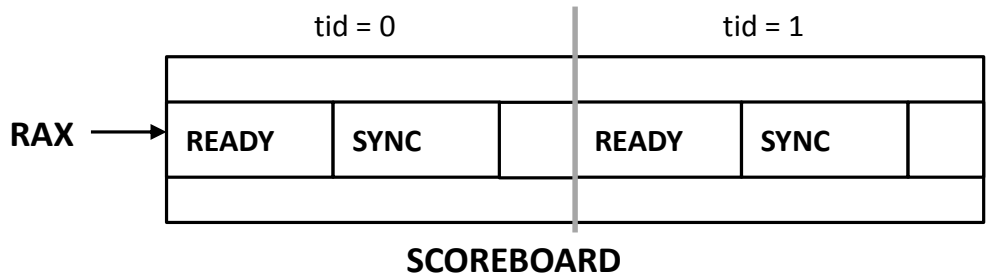


Figure 5.13: Scoreboard for register communication in loop parallelization.

This fine grain register communication mechanism allows better iteration overlap and better instruction scheduling than other techniques such as signal/wait [141]. Moreover, there is no requirement of having FIFO queues to store values since they can be read directly from the register files.

Live-outs must be handled carefully upon loop finalization. This is described in more detail in Section 5.3.6.

5.3.4.2 Memory Communications

As we have presented before, we assume threads do not communicate through memory. However, this cannot be guaranteed and it may be possible that the speculative thread reads a value from memory before it has been generated by the non-speculative thread that is executing the code of the previous iteration (Read After Write (RAW) dependence). Memory violations are detected by using a bloom filter technique [142]. A speculative thread reading an incorrect value is allowed to continue execution until the non-speculative thread detects the memory violation and a rollback process is initiated for squashing the speculative work. The squashed thread will always be the speculative thread because, from a semantic point of view, the iteration being executed by the non-speculative thread is always previous in the original program order.

The memory instructions of one code region (iteration) are not reordered by the software layer on region formation. Moreover, Write After Write (WAW) and Write After Read (WAR) dependences are satisfied correctly because iterations finish in order and stores are also promoted to memory in order by using the gated store buffer. Note that a load can read the local store buffer if a previous store in the iteration has already generated the data (intra-iteration dependence). Also a load can read the remote store buffer looking for data generated by stores of previous non-committed iterations (inter-iteration dependence).

As said earlier, the register file is checkpointed and the gate of the store buffer is set on each iteration starting point (Begin of Iteration, BOI). Therefore, the same mechanism for architectural state checkpointing can be employed to guarantee correct execution of the loop being parallelized. In case of a successful execution of a given iteration, the shadow register file is discarded and the gate of the store buffer is open. On the other hand, in case of an unsuccessful execution of a given iteration, the register file is restored by using the content of the shadow register file and the speculative stores gated in the store buffer are discarded. Therefore, rollbacks may happen because of exceptions and memory

violations. However, unlike exceptions, a memory violation rollback requires unsetting the *Sync* bits in the scoreboard of the other thread and the processor remains in optimized mode ready for re-executing the squashed iteration.

As commented before, a memory violation rollback is raised when the bloom filter structure detects a memory inter-iteration dependence infringement. Load instructions executed by the speculative thread (the thread executing the youngest iteration) update the bloom filter mask indicating the addresses they read. By contrast, store instructions executed by the non-speculative thread (the thread executing the oldest iteration) check the bloom filter mask to detect if a younger load has read the value before it has been generated. In Figure 5.14, there is an example of how the bloom filter is used to track memory dependences. As it can be observed, *load @X* of iteration 1 executes before *store @X* of iteration 0 and it reads a stale value in consequence. Since the load instruction updates the bloom filter mask, the store of iteration 0 is able to see the incorrect remote access and it rollbacks the speculative thread execution. After the rollback of the speculative thread, the execution of the loop is re-established as depicted in the right part of the figure. In this new scenario, *load @X* of iteration 1 is now reading the correct value produced by *store @X* of iteration 0 and so that, no additional rollbacks are required. Note that our mechanism assumes that *load @X* will then execute after *store @X*, but there is no guarantee that this will happen. Hence, it may be the case that the speculative iteration is squashed several times before it is able to proceed. However, we have not seen this happening in our simulations.

Bloom filters may report false positives but not false negatives [142]. This guarantees correct memory violation detection at the cost of killing iterations where there are no real memory violations. The baseline bloom filter accuracy can be improved for the proposed LP scheme in different ways. For instance, one solution is to consider more than one bloom filter (having more than one hashing function). A different solution may be letting the co-designed virtual machine to selectively mark memory instructions that are known not to alias. In this document, we use a regular bloom filter since the accuracy is good for the benchmarks we analyze.

From our studies, we have seen that less than 1.5% of the iterations are squashed due to memory communications when LP is employed. A squash occurs when iteration *i+1* reads a memory value produced by iteration *i*, but the load executes before the store due to the out-of-order execution nature of speculative LP scheme.

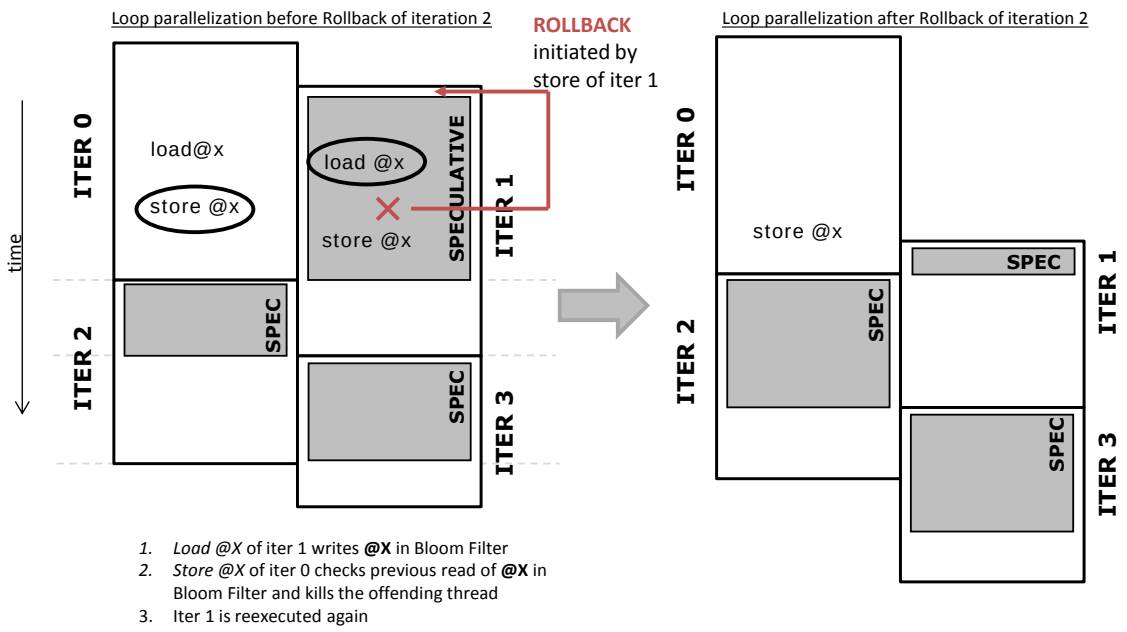


Figure 5.14: Loop parallelization iteration rollback example.

5.3.5 Iteration Finalization

In the proposed LP scheme, correct execution from register and memory dependences point of view is guaranteed by forcing iterations to finalize in program order. After finalizing the execution of one iteration the following two scenarios may occur: a new iteration is executed or the loop execution is completed. As it is shown in Figure 5.15, the iteration finalization is always decided by a branch instruction. If the loopback branch instruction jumps back to the beginning of the loop, then a new iteration is going to be executed. However, if the loopback branch is not taken or if another branch jumps to an address out of the loop, then the whole loop execution is finalized. From a nomenclature point of view, a branch instruction that drives to a new iteration is called loopback branch, whereas a branch that goes out of the loop is called exit. If the exit occurs in the middle of an iteration it is called early-exit.

Potential loop exits have to be identified to ensure the in-order iteration execution and a correct loop finalization. The software layer marks branch instructions by using two additional flags in its encoding representation. One flag is used to identify if the instruction is an iteration finalization instruction when taken (the loopback), the other flag indicates if the loop execution ends when this instruction is taken or when it is not taken. Note that since we build superblock loops, there are no internal branches and the aforementioned flag marks suffice to synchronize the execution of iterations. In other words, all branches may finalize the loop depending on their condition. We call the former

the finalization flag, and the second the exit flag. We can use Figure 5.15 to illustrate how these flags are used. In the case of an early exit branch, the iteration finalization flag is set to 1 and the exit flag is set to taken. By contrast, for a loopback branch, the iteration finalization flag is set to 1 and the exit flag is set to not-taken. Note that all branches within the loop always have the iteration finalization flag set to 1. Moreover, branches that finalize iteration execution are called End Of Iteration (EOI) instructions.

In order to guarantee that iterations are finalized in order, EOI instructions are executed in a special manner. If the executed EOI belongs to a speculative thread, then it has to be stalled until the non-speculative thread successfully commits the execution of the previous iteration. This scenario is shown in the Figure 5.16 by using the iterations 0 and 1 of a generic loop. On the other hand, an EOI instruction can be directly executed and retired if it belongs to the non-speculative thread. One example of this scenario is shown in Figure 5.16 by iterations 2 and 3. The case of loop finalization is described in more detail in Section 5.3.6.

The execution of an EOI instruction is used to compute which of the two threads is speculative. Actually, the threads always change their execution automatically from speculative to non-speculative or from non-speculative to speculative when executing an EOI. In other words, the speculative behavior ping-pongs between the two threads as iterations are finalized. The example of Figure 5.16 helps us to illustrate how the thread execution mode moves from speculative to non-speculative and vice-versa. In the example, iteration 0 starts as non-speculative whereas iteration 1 is speculative. Iteration 1 cannot directly commit because iteration 0 is required to commit before. Once it commits then the thread 0 moves to speculative and the thread 1 to non-speculative. However, just immediately, the thread 1 that was waiting for commit executes the EOI, moving again thread 1 to non-speculative and thread 1 to speculative. Finally, when thread 0 executes the EOI of iteration 2 then it moves to speculative and thread 1 executing iteration 3 moves to non-speculative. Note that, as commented in previous sections, instructions belonging to the older not committed iteration are always non-speculative, whereas instructions belonging to younger iterations not committed are always speculative. Therefore, we can also use the iteration counter and the original thread identifier to determine which thread is speculative at any point of the execution.

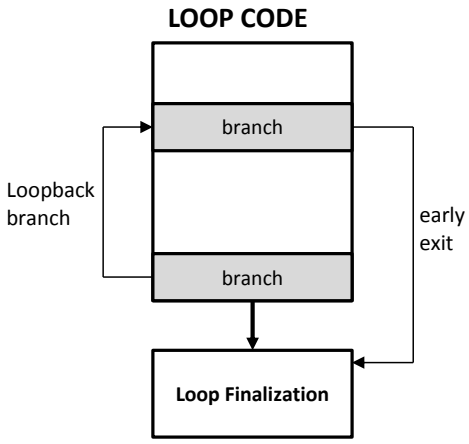


Figure 5.15: Loop finalization branches.

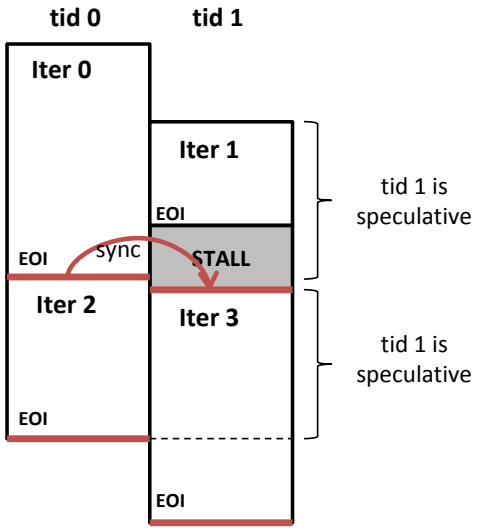


Figure 5.16: Loop parallelization iteration finalization example.

5.3.6 Loop Finalization

Loop Finalization occurs when an iteration finalization branch exits the loop. Although all branch instructions are EOI, in this particular case such a branch acts as an End Of Loop (EOL) instruction. Once the loop finishes, live-out values generated during execution are spread among both register files. Since the original thread will continue with the execution, its register file must be correctly updated as if the whole loop were executed in single-thread mode. Otherwise, instructions reading values generated during the execution of the loop may consume stale data and hence, the correctness of the application execution will be compromised.

Therefore, some additional tasks have to be done before continuing with the application execution once a loop is exited. First, the spawned thread has to be squashed freeing the

corresponding hardware context in the processor. Second, the original thread local register file has to be updated with the live-out values generated by the other thread because these values are stored in the remote register file.

In Figure 5.17, we present how the live-out register values should be handled at the finalization of a generic loop. The code of the loop is shown in the left part of the figure. It is formed by two basic blocks, A and B. The first BB ends in the "jeq EXIT" branch instruction that it is at the same time an early exit of the loop. The second basic block ends in the loopback branch "jne LOOP". Basic block A defines registers ECX, ESI and EAX, whereas basic block B defines ESI, EDI, and EBX. Note that EDX is an invariant in the loop. ESI and EDI are used as intra-iteration dependences and ECX, EAX, and EBX are used as inter-iteration dependences. The following four different scenarios may happen depending on which thread ends the loop and from which branch instruction:

1. Original thread ends the loop execution:
    a. Iteration finalization from the early exit: the live-outs that are defined in basic block B by the non-original thread that are not re-defined in basic block A (in this case EDI and EBX, since ESI is also defined in basic block A) are located in the non-original thread register file and they need to be moved somehow to the register file of the original thread. This case is only valid if the loop is not exited in the first iteration. Otherwise, the non-original thread has not produced any value.
    b. Iteration finalization from the loopback branch: nothing additional to be done since all register values generated in the loop have been generated by the original thread.
2. Non-original thread ends the loop execution:
    a. Iteration finalization from the early exit: the non-original thread generates the live-out values corresponding to basic block A. All of them need to be moved from the register file of the non-original thread to the register file of the original thread.
    b. Iteration finalization from the loopback branch: the non-original thread generates the live-out values corresponding to both A and B basic blocks. ECX, ESI, EAX, EDI, and EBX register values needs to be moved from the register file of the non-original thread to the register file of the original thread.

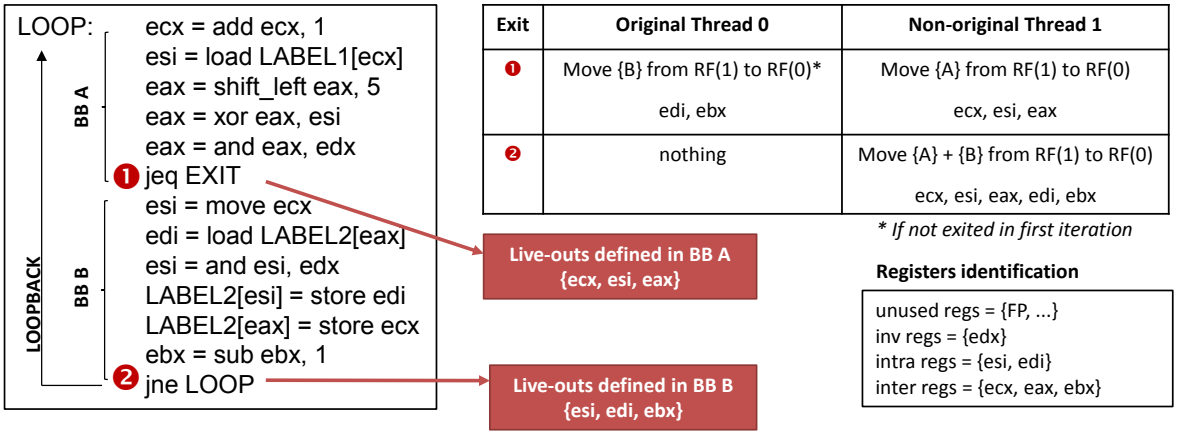


Figure 5.17: Loop finalization example.

In order to avoid adding explicit special move instructions to move values from one register file to the other, we propose a new hardware extension that allows registers to spread between both register files. The idea is to establish a dynamic binding between threads and registers, having a hardware table structure that indicates in which register file is the latest value produced for each register. In such a way, the movement of values is not required and the original thread will continue accessing the remote register file for those registers that are live-outs generated by the spawned thread. Therefore, this structure needs to be accessed even after the loop parallelization process.

The idea of the dynamic binding between threads and registers is to mark where the current value of a register resides: whether it resides in the local or in the remote register file. This is achieved by attaching a "where" bit W mask to each register. A value of 0 for a given register indicates that both threads use their local register, whereas a value of 1 indicates that both threads use their remote register. An example is shown in Figure 5.18 where thread 0 uses the local register for registers r1 and r3, and the remote register file for registers r2, r4, and r5. The same applies to thread 1 that uses the local register file for r1 and r3 and the remote one for r2, r4, and r5.

Applying this concept to the speculative loop parallelization approach, it is just a matter of appropriately updating the W mask. Once a parallelized loop is exited, the W mask update is performed in such a way that reflects where the live values of each register reside. Note that in this case any of the two threads are able to continue the execution after loop finalization, as opposed to the static scheme, in which the original thread is always in charge to continue execution.

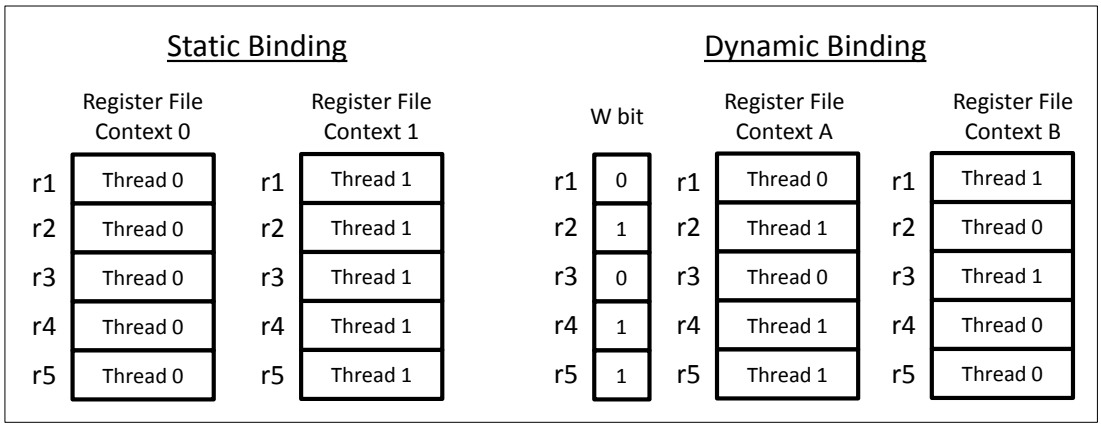


Figure 5.18: The traditional static binding between registers and threads is shown on the left. On the right, a dynamic binding is achieved by using a Where bit mask W. For each register R_i, W_i identifies whether the local (0) or remote (1) register is used. Note how in this case we refer to contexts A and B and not contexts 0 and 1. For example, when thread 0 uses register R2 it will access the copy in context B, while thread 1 will access the copy in context A.

The updated W mask can be generated in different ways. One option is to encode it within each branch instruction in the loop (mask can be computed when code is generated for the loop). Another alternative is to have these masks in memory. However, we think the best implementation for updating the W mask is based on using simple hardware structures.

The idea is to have two additional masks referred to as DEF masks, each one assigned to a thread as shown in Figure 5.19. DEF masks implement one bit per register and describe which registers have been written by the corresponding thread in the currently executed iteration. These masks are initialized to zeros whenever a thread starts executing a new iteration. Each time an instruction writes to a register, its corresponding definition mask DEF is updated with a 1 for that particular register and thread. After one thread finishes one iteration, the corresponding mask DEF is flashed to another new mask called W_{next}. W_{next} accounts for who was the last writer for each register. Since our scheme guarantees that iterations finish in order, W_{next} is an accumulative mask updated after each iteration. Note that Begin of Loop, Begin Of Iteration, End Of Iteration, and End Of Loop are identified by flow marking specific instructions in the loop body. This flow marks are used by the hardware to detect the boundaries of the loop and the iterations. After the execution of the loop, W and W_{next} are combined using a XOR-like function to generate the final W mask. Note that the proposed scheme is very simple and low-complex because it only uses three additional masks that are updated by employing very simple logic functions.

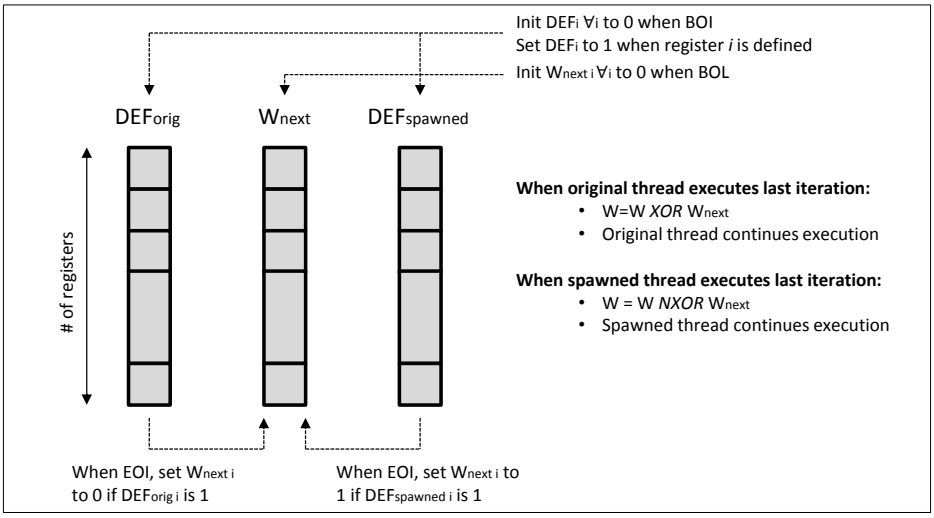


Figure 5.19: Structures used to compute mask W. Dotted lines describe how and when these structures are initialized and updated.

Finally, it is worth to remark that the dynamic binding between registers and threads is totally transparent to the normal execution mode of the processor (where normal mode refers to periods when the processor is not executing a parallel loop). In this case, each thread works independently with the registers indicated by the W mask. Note in addition that the mechanism is also transparent to the Operating System and context switches in consequence.

5.3.7 Exception Handling

The proposed LP scheme supports precise exceptions. However, unlike in traditional hardware/software co-designed virtual machines, in a loop parallelization process there may be two code regions (iterations) being executed simultaneously when the exception is raised. Although we consider one iteration as speculative and the other as non-speculative, none of them has been already committed, so that, both of them may be rolled back and the architectural state reestablished to the latest committed point for both threads. After the exception is solved, the failing iterations are executed in non-optimized mode.

Moreover, loop parallelization mode is enabled after the iteration raising the exception is correctly executed and committed. This approach guarantees forward progress by ensuring safe execution of the failing iterations. In this sense, an exception is treated similarly to an EOL instruction. In particular, the bloom filter and the scoreboard structures are initialized again removing all activity from the previous execution of the loop. Otherwise, the threads may read stale values from the other register file or incur in false memory violations.

Component	Size	Type	Description
2xGated SB[14]	16 entries	Hw	Each thread has its own Gated Store Buffer
2xRegister File[14]	32 int 34 fp	Hw	Each thread has its own Register File
Scoreboard	32 entries	Hw	This structure tracks the register dependences (intra, inter and invariant) between two threads
Bloom Filter	1024 bits	Hw	This structure tracks the RAW memory dependences between the two threads
Original thread register	1 bit	Hw	Register with the context id of the thread that initiates the loop parallelization
Speculative thread register	1 bit	Hw	Register with the context id of the spawned thread (only valid when parallelizing loops by using two contexts ids)
Iteration number register	16 bits	Hw	Register that holds the number of the latest valid committed iteration
DEF original mask	66 bits	Hw	Original thread register definition mask. It uses one bit per register to describe which registers have been written by the original thread in the currently executed iteration.
DEF spawned mask	66 bits	Hw	Spawned thread register definition mask. It uses one bit per register to describe which registers have been written by the spawned thread in the currently executed iteration.
W mask	66 bits	Hw	Where bit mask. This mask indicates per each register if its value resides in the local register file or in the remote register file.
W next mask	66 bits	Hw	Where next bit mask. This mask accounts for who was the last writer for each register.
EOI flag	1 bit per instruction	Both Hw/Sw	End Of Iteration flag associated with an instruction. Software is in charge of assigning this flag to corresponding instruction
Inter/Intra/Invariant flags	5 bits per instruction	Both Hw/Sw	Instructions flags associated to register operands (source and destination) to track inter/Intra/Invariant-iteration register dependences. Software is in charge of setting these flags
Iteration finalization flag	1 bit per instruction	Both Hw/Sw	Instruction flag indicating if the instruction is a branch that may finalize the execution of the current iteration (it depends on the iteration exit flag value). Software is in charge of initializing this flag
Iteration exit flag	1 bit per instruction	Both Hw/Sw	Instruction flag indicating if the instruction is branch that may finalize the execution of the current iteration when it is taken or not-taken. Software is in charge of initializing this flag

Table 5.1.Summary of hardware/software additions to support Loop Parallelization in a co-designed processor.

[14] Already implemented in SMT processors

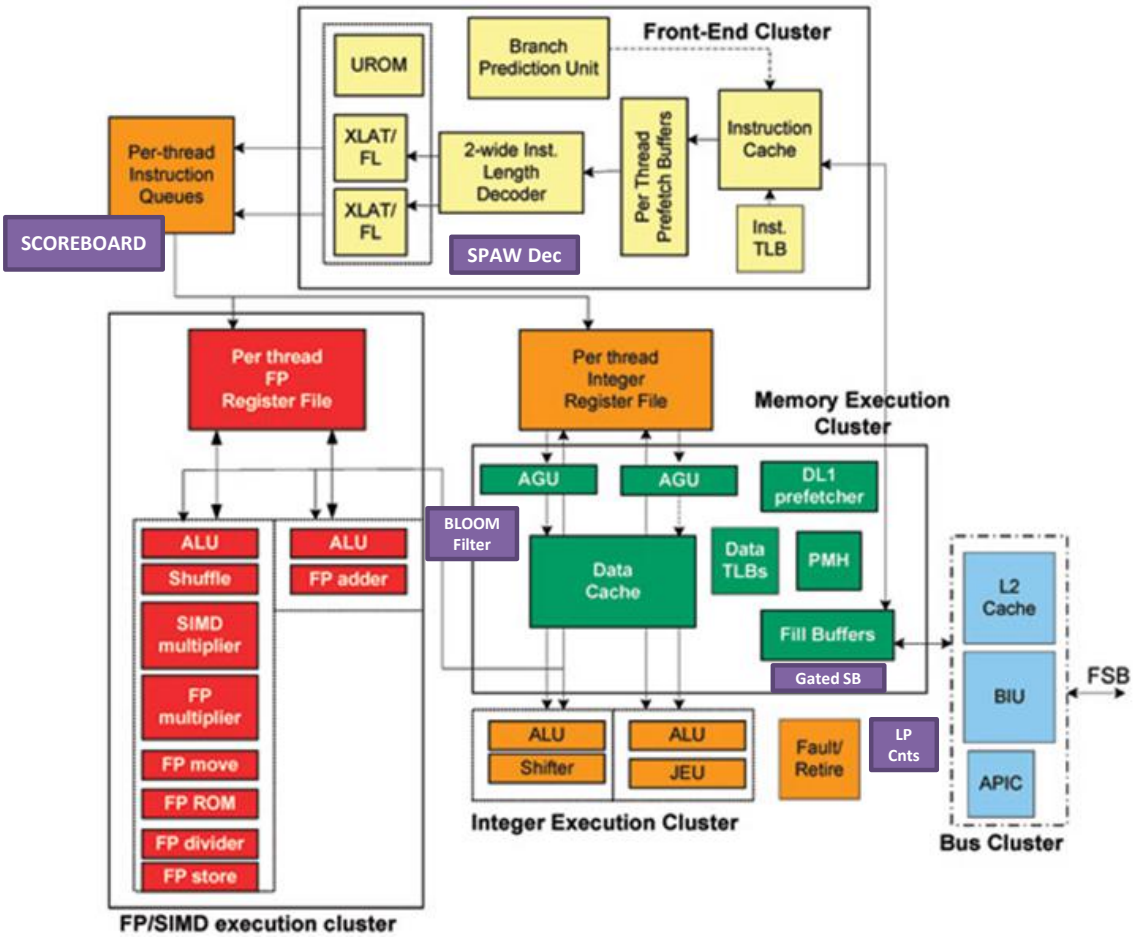


Figure 5.20: Atom-like architecture with the Loop Parallelization mechanism additions (original block diagram from [143]).

5.3.8 Loop parallelization mechanisms summary

The LP scheme presented in this document requires implementing modifications in both hardware and software components of the hardware/software co-designed processor. Table 5.1 summarizes the main modifications required to support loop parallelization on top of a traditional hardware/software co-designed processor that supports SMT. The first column of the table shows the component (or modification) required, the second column shows the size of the component, the third column indicates if the modification is in hardware (Hw), software (Sw), or in both hardware and software (Hw/Sw), and the fourth column is a short description of the component. Note that all these modifications have been commented in more detail in previous sections of this chapter.

In Figure 5.20, we show the Intel® Atom® architecture high level block diagram with the additions for supporting the LP scheme. The *Spawn* instruction decoder component is added to the Front-End cluster and it is accessed before the instruction decoder. The

Scoreboard is placed close to the per-thread *Instruction Queues* because it also controls the issue of the instructions based on whether their consumed register values have been already produced. The *Bloom Filters* and the *Gated Store Buffers* (Gated SB) are placed in the Memory Execution cluster. Both structures are accessed with the physical addresses of the memory accesses. Finally, we place the counters required to keep track of the number of iterations close to the Fault and Retire logic. These counters are updated once the instructions marked with the EOI flag retire.

5.4 Optimizing the Regions to Parallelize

In order to improve the performance of the loops selected for parallelization, we have considered two different types of optimizations. The software layer is in charge of applying these optimizations when it is optimizing the selected loop for parallelization.

The first optimization consists of a list scheduling code reordering technique focused on adapting the original codes to be more efficiently executed in the hardware. The software layer knows in detail the relevant characteristics of the hardware and it is able to dynamically transform the codes in order to improve the usage of the processor resources. For instance, attending to the core characteristics described in Section 5.3.1, it may reorder the original code instructions in order to take advantage from the particular Atom® pipeline that allows back-to-back memory to ALU operations. Given the generality of this instruction scheduling algorithm we have applied it both to the baseline (sequential execution) and the loop parallelization. Moreover, note that in an in-order processor the impact of this type of optimizations is important, because the instruction scheduling is done by the software and not by the hardware like in an Out of Order processor.

The second group of optimizations is focused on minimizing the impact of the inter-iteration register dependences. As commented in Section 5.3.4.1, register data dependences are satisfied in a synchronized manner in order to guarantee correct execution. Such synchronization stalls the execution of the thread that requires a register value generated by the other thread and the distance between the producer and consumer instructions dictates the overlap that can be achieved. In other words, we need to reduce the impact of recurrences in order to have better performance. In order to better illustrate this, we can use the example of Figure 5.21 (left) that shows the execution of the first three iterations of a loop with an inter-iteration register dependence. The dependence is over register EBX that is consumed by the first instruction and produced by the last instruction of the loop. Note that this dependence makes the first instructions of the next iteration to be stalled

until the value of register EBX is produced by the last instruction of the previous iteration. Therefore, iterations are executed sequentially and no parallelization can be exploited. We have considered the following optimizations to minimize the impact of inter-iteration dependences:

- Instruction reordering with special emphasis to reduce the length of the recurrences. We call this optimization as *Recurrence Reordering* and it is described in more detail in Section 5.4.1.
- Convert biased branches to asserts to allow more aggressive reordering. We call this optimization as *Atomicity* and it is described in more detail in Section 5.4.2.
- Usage of temporal register to break false anti-dependences. We call this optimization as *Anti-dependences* and we describe it in more detail in Section 5.4.3.

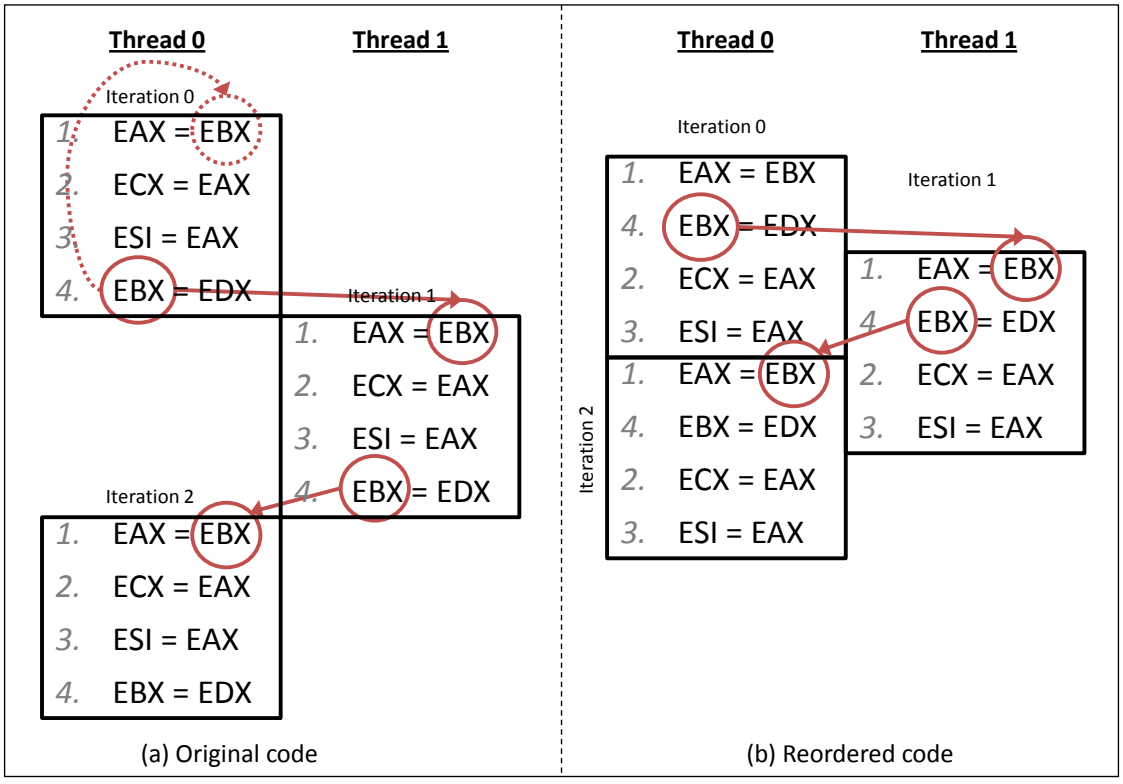


Figure 5.21: Example of inter-iteration dependence based on instruction scheduling for loop parallelization. Figure (a) shows the original code and how it is executed in loop parallelization mode given the inter-iteration dependences. Figure (b) shows the optimized code and how it is executed in loop parallelization mode.

5.4.1 Recurrence Reordering

Besides the aforementioned list scheduling algorithm, we have also considered another instruction scheduling algorithm focused on minimizing the impact of the register inter-iteration recurrences. The main idea is to reduce the length of the inter-iteration dependences by moving the consumer and producer instructions as close as possible. This is achieved by giving more priority in the list scheduling algorithm to instructions that generate a register value that is required by the next iteration.

We illustrate how this reordering optimization works with the example in Figure 5.21. In the figure, we have numbered the static instructions that belong to the region being parallelized from 1 to 4. As it can be observed, the inter-iteration recurrence dependence occurs because instruction 4 produces register EBX that is later consumed by instruction 1. This dependence avoids the iterations to be executed in a parallel manner as shown in the left part of the figure. In the right part of the figure, we show the execution of the first three iterations of the loop once the list scheduling algorithm is applied. In this case, instruction 4 is reordered just after instruction 1. This movement is possible because instructions 2 and 3 have no dependences with instruction 4 (they only have a RAW dependence for register EAX with instruction 1). This simple reordering allows overlapping the execution of the instructions belonging to consecutive iterations as shown in the left part of the figure.

In order to guarantee that the data live outs of the regions are correctly generated, instructions can only be reordered within basic blocks. This limits the benefits that can be obtained by applying this technique when regions have multiple exists.

5.4.2 Atomic Execution of Regions

This optimization converts biased branches to asserts in order to reduce the number of early exists. This allows a more aggressive reordering of the instructions with higher chances to reduce the distance between an inter-iteration producer and its consumer. Actually, the jump instructions that are good candidates to be transformed into assert instructions are those which their outcomes can be predicted with very high confidence. For instance, jump instructions with high percentage of possibilities to be not-taken. In such a way, the objective of inserting the assert instruction is to check if the jump prediction is satisfied. Otherwise, if the check fails, this means that the optimization applied is incorrect and the region needs to be rollback. In order to guarantee forward progress, the loop is then executed in a less optimized version or even in normal execution mode.

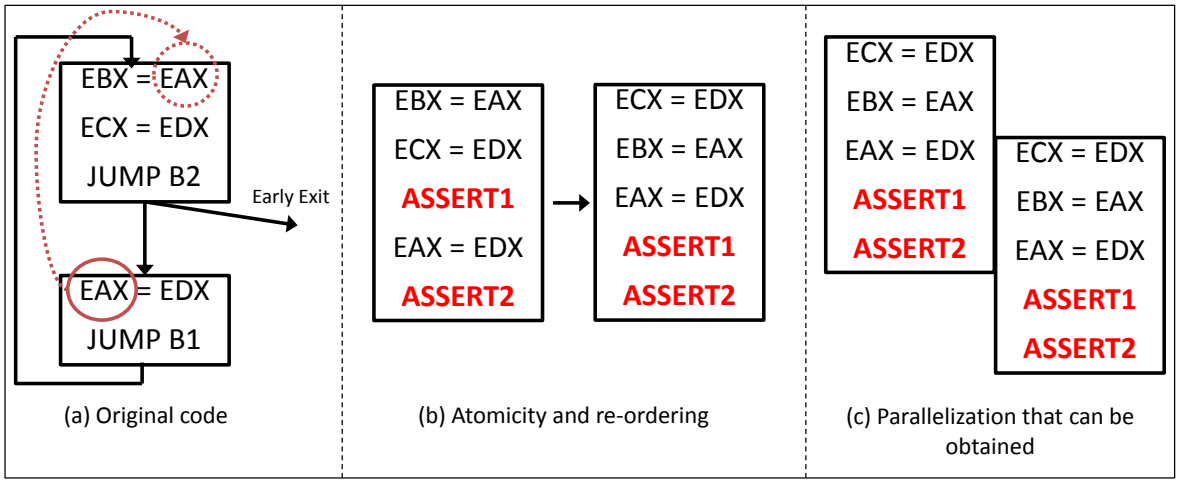


Figure 5.22: Example of atomicity optimization for loop parallelization. Figure (a) shows the original code and the inter-iteration recurrence dependence. Figure (b) shows the resultant code after applying the atomicity optimization. Figure (c) shows how the resultant optimized code is executed when loop parallelization execution mode is enabled.

We use Figure 5.22 to describe how this optimization works. In the figure, we show an example of an original code region in the left part. In the middle part of the figure, we show how the optimization is applied to the code region, converting the early exist branch into an assert instruction, and in the right part, we show the parallelism that can be obtained when executing it in loop parallelization mode. As it can be observed, the original loop is formed by two basic blocks. The first one contains an early exit and the second contains a loopback branch. The instruction of the second basic block is not allowed to be moved over the "Jump B2" instruction that marks the separation among the two basic blocks because it would overwrite EAX incorrectly. As it is depicted in the figure, by converting the jump instructions into asserts we can handle the two basic blocks as if they were one single basic block, allowing the instruction that produces the EAX register value to be scheduled one cycle earlier. Finally, in the right part of the figure, we can see that applying this instruction movement, we can increase the parallelization of the region by overlapping the execution of 3 instructions over a total of 5.

5.4.3 Breaking Anti-Dependences by Using Temporal Registers

The third optimization that we have considered is also focused on reducing the distance between the consumer and the producer of the register inter-iteration dependence. In this case, the main objective is to increase the possibilities of code reordering (instruction scheduling) by breaking key anti-dependences using temporal registers. An anti-dependence (also kown as WAR) occurs when an instruction requires a value that is later

updated. Anti-dependences could be removed by copying the value causing the dependence into a new register (in our case a temporal register) and applying renaming.

Figure 5.23 shows how this optimization can improve the parallel execution of a loop. In the left part of the figure, we show the original code and how it is executed (just the first 3 iterations) in LP mode. As it can be observed, the dependence between the first instruction and the last instruction of the iteration does not allow any parallelism across iterations. Moreover, we cannot apply the previous aforementioned optimizations because there are anti-dependences that do not allow the instructions to be reordered. In other words, instruction I3 cannot be moved ahead of instruction I2 because I2 uses a previous definition of EAX. The proposed optimization breaks these anti-dependences by using temporal registers. In the right part of the figure we show the resultant code. Note that the instruction "EAX=EBX" has been replaced by "TMP1=EBX" and the instruction "ECX=EAX" has been replaced by "ECX=TMP1". In other words, the first instances of register EAX have been renamed to register TMP1. By doing this, all the instructions in the loop can be executed in parallel.

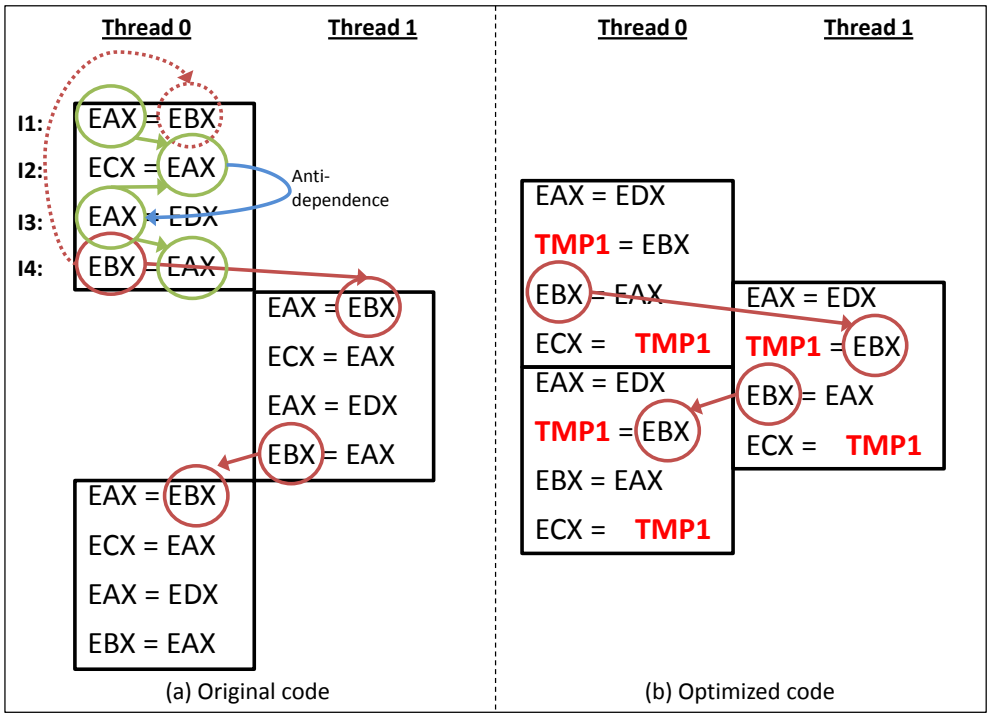


Figure 5.23: Example of register temporal usage for breaking anti-dependences optimizations. Figure (a) shows the original code execution in loop parallelization mode. Figure (b) shows the execution in loop parallelization mode of the resultant optimized code after breaking the anti-dependences for the data inter-iteration recurrence.

5.4.4 Combining the Optimizations

All optimizations presented in this section are orthogonal and can be applied simultaneously forming more complex optimizations. However, applying some of them together does not guarantee better results than applying the optimizations in a separated manner. For instance, the use of temporal registers to break anti-dependences reduces the length of the inter-iteration dependences at a potential increase of the execution time of individual iterations. Hence, optimizations should be combined depending on the characteristics of each individual loop.

In Section 5.5, we analyze the execution performance impact of all the different optimizations presented in this section.

5.5 Evaluation

5.5.1 Simulation Framework

The LP scheme has been implemented on top of an in-house research simulator that models a hardware/software co-designed processor. The development of this simulator has required modeling the software and the hardware components. Both components have been designed to work in a collaborative manner. The software models a hardware/software co-designed virtual machine that, among other things, it is able to detect hot code (including loop regions), to optimize the detected hot regions and to store them in the code cache region for their later reuse. On the other hand, the hardware component models an Intel® Atom® processor and it is based on the architecture described in Section 5.3.1. This cycle accurate simulator allows 4 instructions (μops) to be issued and retired simultaneously and it models the Intel ® x86 ISA in very high detail, including interrupts, exceptions, APIC support, etc. Unfortunately, such high detail on modeling the Intel® x86 ISA and some specific problems with the x86 emulator regarding special ISA features have made impossible the implementation of a realistic memory and branch predictor models. Thus, the LP paradigm has been evaluated using perfect memory and branch prediction. Note that these simplifications do not necessarily work in favor of the presented numbers. In fact, the LP paradigm using a perfect cache is not able to provide performance benefits when the application has important amount of Memory Level Parallelism (MLP), a key contributor factor to performance for code parallelization [10] [130].

We have summarized the more relevant configuration parameters for the simulator in Table 5.2 and Table 5.3. Table 5.2 describes the hardware parameters and Table 5.3 the

software parameters. The first column of each table shows the parameter name and the second column shows the value used in the simulator.

We have performed simulations using 12 programs from the SPEC2000Int benchmarks suite. The selected programs are *bzip2, crafty, eon, gap, gcc, gzip, mcf, parser, perlbmk, twolf, vortex,* and *vpr*. Each benchmark is simulated by using a set of representative traces of 10 million instructions. These traces have been obtained by gathering the program information directly from real processor activity. For each trace there are two simulation phases. During the first phase, the software layer identifies the loops and optimizes them for execution. In the second phase, the loops are executed in the hardware simulator and relevant statistics are collected.

Hardware Simulator Parameters	
Parameter	**Configuration**
Multithreading	2 SMT
Issue queue	32 entries (16x2 threads)
Issue	4 µops (2 per thread in case that both thread issue instructions)
Retire	4 µops
Pipeline Stages	16 stages
Data Cache Read Ports	1 port
Data Cache Write Ports	1 port
Branch Predictor	Not modeled
Cache Hierarchy	Not modeled

Table 5.2: Hardware simulator configuration parameters.

Software Simulator Parameters	
Parameter	**Configuration**
Loop Detection Threshold	50 executions
Super-Block Formation Branch Bias Threshold	80%
Code Cache	Unbounded

Table 5.3: Software simulator configuration parameters.

5.5.2 Results

In this section, we evaluate the LP scheme by employing the previously presented simulation framework. First we analyze the impact on performance of the thread spawning mechanism. Later we present the performance numbers for the LP paradigm with and without applying the different presented optimizations. We conclude this section by comparing the different optimizations and showing an upper-bound of the performance that can be achieved when the best is considered for each particular loop.

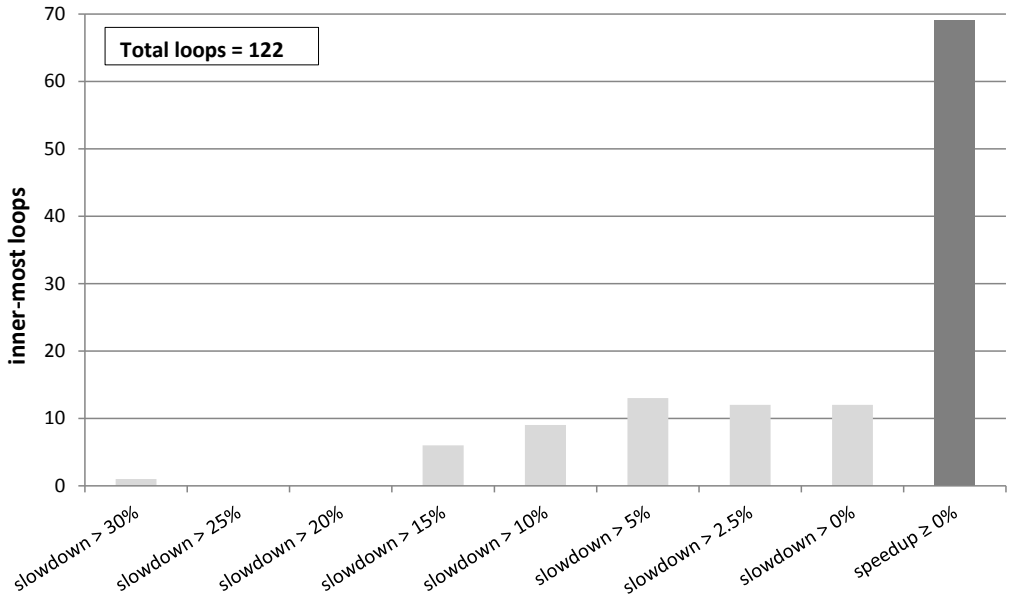


Figure 5.24: Spec2000Int inner-most loops speedups for loop-parallelization with respect to sequential execution using *Spawn-at-execute*. Efficient SPAWN mechanism is not implemented. The y-axis represents the number of loops bucketed based on their performance (x-axis).

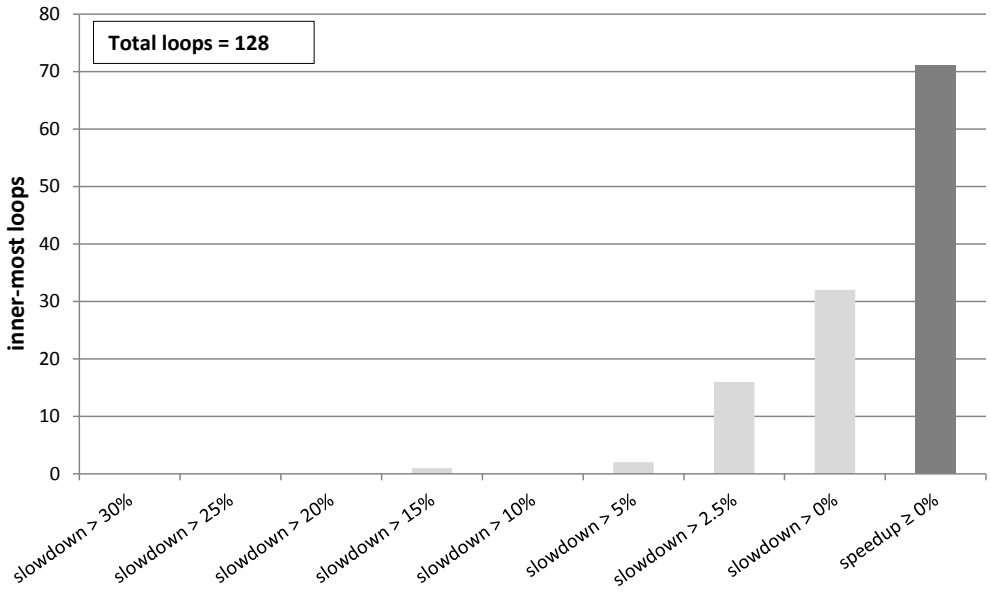


Figure 5.25: Spec2000Int inner-most loops speedups for loop-parallelization with respect to sequential execution using *Spawn-at-fetch*. Efficient SPAWN mechanism is implemented. The y-axis represents the number of loops bucketed based on their performance (x-axis).

5.5.2.1 Removing Thread Spawning Bubbles

In this section, we evaluate two different proposals for thread spawning. The first one consists on activating the fetch of the spawned thread once the Begin Of Loop (BOL) instruction is issued, while the second one consists on activating the fetch of the spawned thread once the BOL instruction is fetched. This second approach can be achieved by pre-decoding the BOL instruction in the first stages of the pipeline[15], as described in Section 5.3.2. We refer to both options as *Spawn-at-execute* and *Spawn-at-fetch* respectively.

The evaluation has been done for each individual loop of the Spec2000Int benchmarks that has been detected as hot. We have compared the required execution time of each loop with and without LP support and by using both thread spawning approaches. In Figure 5.24, we show a histogram with the number of loops classified according to the relation between their execution time for *Spawn-at-execute* and *Spawn-at-fetch* compared to the sequential execution. The categories of the histogram range from slowdowns greater than 30% to speedups and the y-axis shows the amount of loops. As it can be observed there are a total of 122 loops identified across all benchmarks. We can see that 53 loops have slowdowns and 69 speedups. From the slowdowns, there is one loop with a slowdown greater than 30% and 6 with slowdowns greater than 15%. Most of these slowdowns are consequence of waking up the second iteration too late for loops with low trip counts.

In Figure 5.25, we show the same histogram when using *Spawn-at-fetch*. Unlike the results presented in Figure 5.24, in this case, only one loop presents a slowdown greater than 15%. Although there are still 51 loops that present slowdowns, all of them are in the range of 0% to 5%. As it can be observed, when comparing Figure 5.24 and Figure 5.25, advancing the spawn of the second thread to the fetch stage of the processor has a positive effect on the performance of most of the loops. In addition, the slowdowns observed when compared to sequential execution of the loops are consequence of other limiting factors not related to the thread spawning overheads, such as the inter-iteration register dependences for example. These other limiting factors are described later in more detail.

5.5.2.2 Loop Parallelization Performance without Optimizations

In this section, we evaluate the LP scheme when not applying any of the optimizations oriented to maximize the parallelization of the loop iterations and when applying code reordering only. In Figure 5.26, we show the speedup of the detected loops in the

[15] A similar solution can be achieved by flashing the contents of the issue queue from one thread to the other, once the BOL instruction is issued. This solution is also mentioned in Section 5.3.2

Spec2000Int benchmarks for different techniques. The baseline executes the loop iterations sequentially without applying code reordering. The techniques analyzed are as follows: sequential execution with instruction reordering (Baseline+Reordering), LP and LP with instruction reordering (LP+Reordering). The techniques presented do not show performance improvements for crafty, gap, and twolf applications. By contrast, they present high speedups in bzip2, eon, vortex, and vpr applications. On average, LP+Reordering gets 14.7% speedup, whereas Baseline+Reordering gets 8.2% and LP 6%. In most of the benchmarks, LP+Reordering is the best option. Moreover, LP alone presents worse performance numbers than the Baseline+Reordering approach. This means that instruction reordering based on the processor characteristics improves both the baseline and LP approaches significantly. For this reason, the baseline presented in the rest of this chapter considers this instruction reordering optimization always enabled by default.

Finally, Figure 5.26 shows that LP without reordering performs better than the reordering optimization in eon, gcc, gzip, and vortex. In eon and vortex the effect of combining both optimizations is additive, whereas for gcc and gzip it is not. In other words, for gcc and gzip the benefits from instruction reordering are hidden within the LP optimization benefits. By contrast in mcf, parser, and vpr the speedups come from the instruction reordering optimization and the LP alone does not improve performance at all. As we show in next section, the loops in these benchmarks suffer from register inter-iteration dependences and require additional optimizations in order to improve their performance.

5.5.2.3 Loop Parallelization Performance with Optimizations

In this section, we evaluate the optimizations designed to reduce the impact of inter-iteration dependences. As previously described in section 5.4, we consider three optimizations: (1) Code reordering with the objective of moving the producer and the consumer as close as possible (*Recurrence-Reordering* optimization), (2) Atomic regions to allow instruction reordering over branch instructions (*Atomicity* optimization), and (3) usage of temporal registers to break anti-dependences (*False Anti-Dependences* optimization).

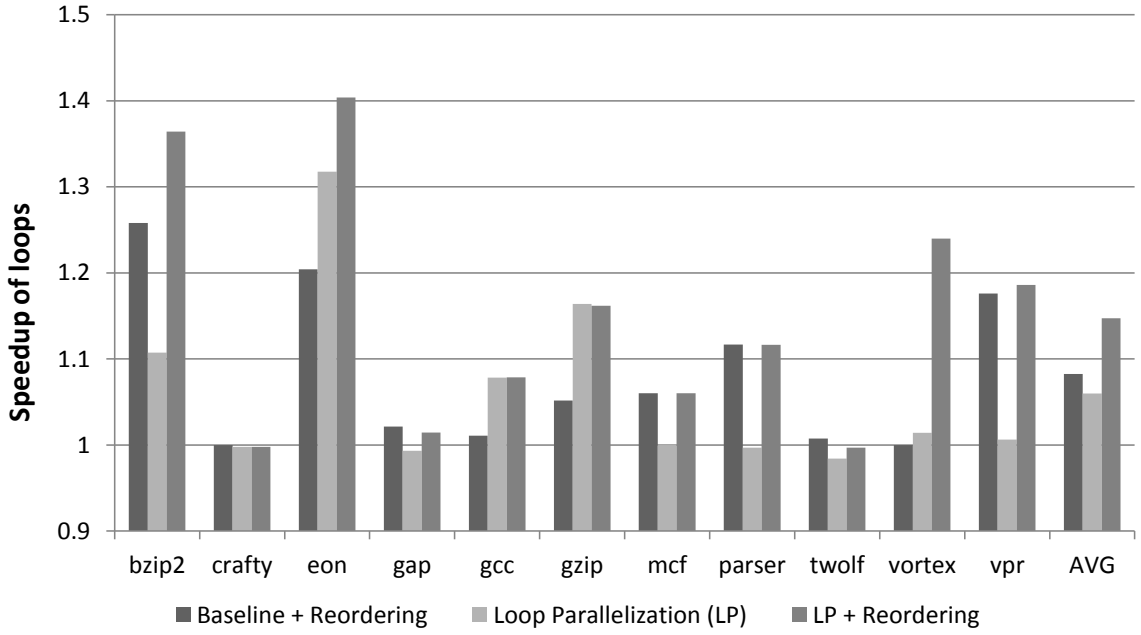


Figure 5.26: Loop parallelization performance with no optimizations applied to the regions.

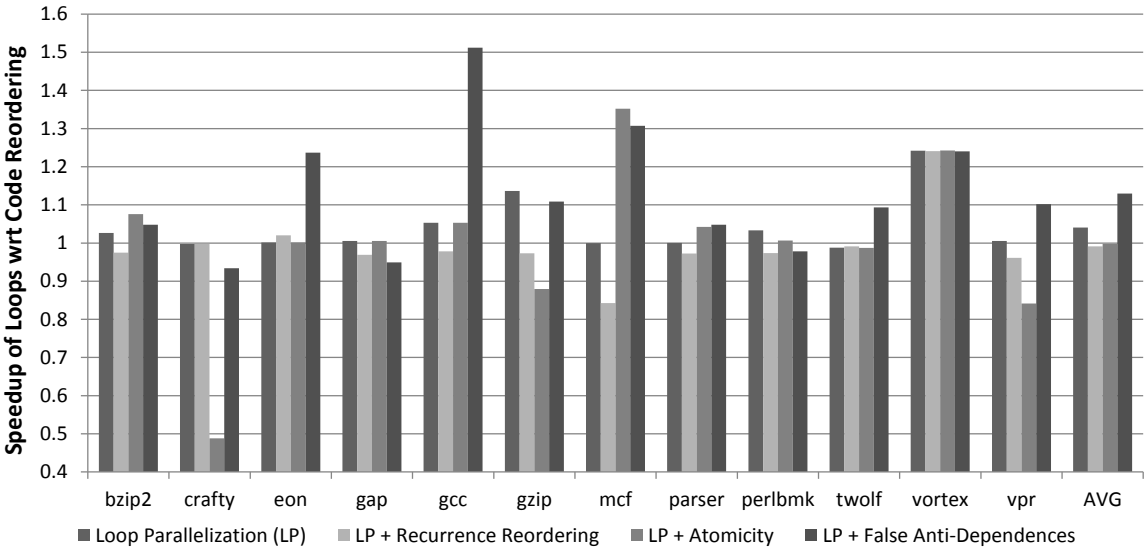


Figure 5.27: Speedup of inner-most loops with respect to code reordering.

In Figure 5.27, we show the speedups obtained for Spec2000Int loops when using the three optimizations. As commented before, the baseline is the sequential execution with code reordering and we have also included the baseline LP scheme (LP without any optimization) in the figure. As it is observed, *LP* achieves 4.1% speedup when compared

to the sequential execution of the loop iterations[16], *Atomicity* and *Recurrence-Reordering* do not improve performance in general, and *False Anti-Dependences* achieves 13% speedup on average. In general, this last optimization is the best option for most of the benchmarks. However, there are some benchmarks where *Recurrence-Reordering* and *Atomicity* perform better than *False Anti-Dependences*. For instance, *Atomicity* is the best option in bzip2 and mcf. In fact, the proposed optimizations affect differently to each individual loop. Therefore, applying the optimizations globally to all loops does not guarantee the best results. In some cases, the proposed optimizations reduce the distance between instructions in an inter-iteration dependence at a potential increase of the execution time of the individual iterations. In these situations, it maybe more convenient to select a different approach to improve the performance.

In order to analyze the best optimization for each individual loop, and hence, to determine the upper-bound performance that we can achieve by applying the LP scheme, we have applied all the optimizations and their combinations to all detected loops of the Spec2000Int benchmarks. Thus, in this study we have the following optimizations:

- Code Reordering (CR): simple instruction reordering based on processor characteristics.
- Atomicity (AT): this optimization includes code reordering and the atomicity optimization.
- Recurrence Reordering (RR): it includes code reordering and the inter-iteration dependence based reordering.
- False Anti-Dependences (AD): it includes code reordering and the usage of temporal registers to break anti-dependences.
- Atomicity + Recurrence Reordering (AT + RR)
- Atomicity + False Anti-Dependences (AT + AD)
- False Anti-Dependences + Recurrence Reordering (AD + RR)
- Atomicity + False Anti-Dependences + Recurrence Reordering (AT + AD + RR).

Therefore, we have simulated eight versions of the same loop by using the eight different combinations. Later, we have chosen the best version for each individual loop (the one that executes the loop in less time) using an offline analysis of the statistics. Since this

[16] Numbers are slightly different from the ones presented in Section 5.5.2.2 because of some enhancements applied to the simulation infrastructure regarding x86 emulation support.

selection is done in an offline manner, and it is the best optimization per each individual loop, we call it as the Oracle scheme. Figure 5.28 summarizes the results obtained for the Spec2000Int benchmarks. LP scheme achieves a 26.3% speedup when compared to a baseline with code reordering. However, the aforementioned optimizations may also help increasing the performance of the baseline configuration. When comparing the Oracle LP scheme against and Oracle baseline, where the best version of each individual loop is also considered for the baseline, the speedup is reduced to 16.5%.

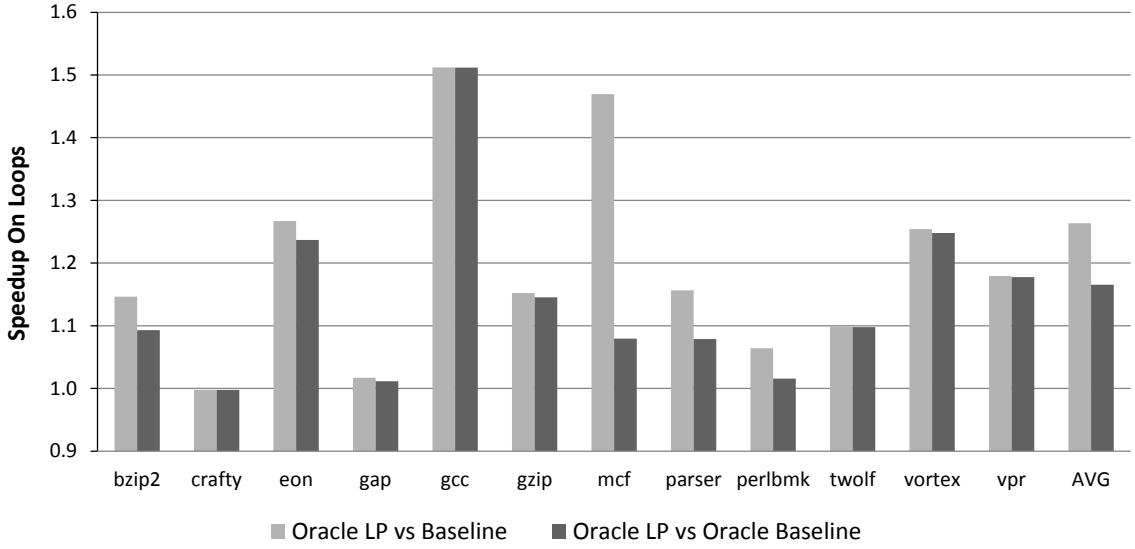


Figure 5.28: Loop parallelization speedup on loops for Spec2000Int benchmarks.

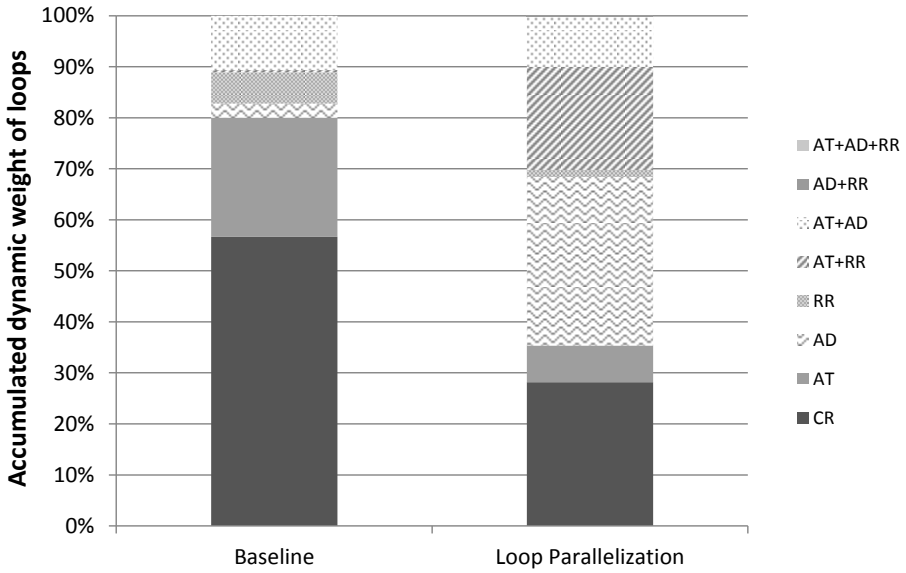


Figure 5.29: Accumulated dynamic weight of loop across different optimizations applied to the code.

In order to understand the importance of each optimization, Figure 5.29 summarizes what version is chosen for each individual loop by the offline Oracle for the baseline configuration and for the LP scheme. As it can be seen, *Code-Reordering* and *Atomicity* optimizations accumulate 80% of the loops for the baseline configuration. On the other hand more optimizations are crucial to improve performance for LP. The *False Anti-Dependences* optimization is the best optimization for 33% of the loops, followed by *Atomicity + Recurrence-Reordering* that is the best for 20% of the loops. Finally, *Atomicity + False Anti-Dependences* combined accumulates 9.8%. This numbers correlate with the ones shown in Figure 5.27, since *False Anti-Dependences* is the optimization that performs better globally for all loops.

5.6 Conclusions & Future Work

Instruction level parallelism (ILP) has been improved significantly during the last decades. However, this type of parallelism is difficult to exploit because it requires complex hardware structures that lead to high power consumption and design complexity. In this scenario, chip multiprocessors (CMPs) and simultaneous multithreading schemes (SMT) are good alternatives to improve performance while keeping a reasonable power budget. These schemes consider the execution of multiple parallel applications at the same time exploiting what is known as thread level parallelism (TLP). Unfortunately, designing techniques to exploit TLP and ILP is not straight-forward.

In this chapter, we have proposed Loop Parallelization (LP), a hardware/software co-designed scheme that uses multiple thread contexts to improve the execution of single-threaded codes. The proposal detects loop codes and optimizes them to be executed in parallel by using two SMT threads. LP makes use of already existing processor resources and it executes the loops locally (within the same processor). For this reason, overheads are small, which allows LP to even parallelize small loops or loops that iterate just a few times. Moreover, the hardware and the software collaborate to guarantee correct loop execution. Threads communicate register values in a synchronized manner and memory dependences are tracked in order to ensure that the outcome of the loop execution matches the outcome of the original code it represents. Finally, LP is only enabled when there is a free context in the processor. If there are no free contexts then the loop is executed as in the original sequential order with no additional overheads because of LP.

LP is a very attractive scheme for improving the performance of single threaded applications. As it has been shown, LP improves the execution time of the loops in Spec2000Int benchmarks by more than a 26% by employing two threads when compared

to a baseline that does not consider any optimization. Moreover, we have also demonstrated that LP can be combined with other optimizations techniques. In fact, the efficient code reordering optimization that takes into account the processor characteristics is able to improve both, the baseline and the loop parallelization codes, such an enhanced code reordering and other optimizations aimed to increase the execution overlap. Even when considering the most optimized baseline, LP scheme is still able to improve it by 16% on average across all benchmarks.

LP scheme does not require increasing the processor design complexity because it exploits most of the existing SMT resources from the Intel® Atom® processor. Moreover, the required hardware additions have been designed with small overheads since LP scheme has been designed to do not penalize loops that iterate just a few times, which are important from our analysis data. We have demonstrated that most of the detected loops are improved, or in the worst case they only present small slowdowns, no matter if their number of iterations is low or if they present small trip-counts.

In this chapter, we have proposed and evaluated the LP proposal. However, several extensions can also be explored as future work. For example, the simulator can be enhanced in order to support realistic branch predictor and memory models and the results can be extended by including other benchmarks suites. Moreover, it would also be interesting to analyze the power consumption of the technique.

Regarding the technique per se, it is necessary to work on heuristics especially designed to select the best optimization to be applied to each particular detected loop. Although we have evaluated all the possible optimizations, we have used an Oracle solution that makes decision in an offline manner. This offline decisions should be done at run-time based on the characteristics of the loop.

In this chapter, we have also commented several times the possibility of extending the technique to support more than two simultaneous threads but we have not fully defined the proposal. It would be also very interesting to consider it, mostly for loops that exhibit high parallelism between iterations.

One idea that has not been evaluated yet and may improve significantly the performance of LP is to not attach threads to iterations in a static manner. In such a way, threads are able to execute as many consecutive iterations (*i+1, i+2, i+3, …*) as possible during the time that the other thread is stalled while executing iteration i (due to a cache miss, TLB miss, etc.). This solution requires adjustments of the already presented mechanisms.

Finally, it would be of special interest to combine the LP scheme with other speculative multithreading techniques in order to continue exploiting TLP and ILP at the same time in a multicore processor. In this scenario, the code generated by the speculative multithreading technique can even be optimized when executed in a particular core processor by the LP proposal.

Chapter 6

Conclusions

The main objective of this thesis was to improve current hardware/software co-designed processors by reducing their hardware and software overheads and by improving their performance. To do so, we have proposed three different techniques.

First, we have proposed a novel hot code detector that is able to detect high quality hot code regions at very low cost. The pillar of the proposal is a novel replacement policy, called LIU, which outperforms the results that can be achieved by other more traditional replacement policies. The LIU proposal achieves 85.49% hot code coverage and it only requires a 128 entries table to implement the profiler. Other profilers require at least 1024 entries to achieve similar coverage results. Moreover, the proposal only increases the processor area by 1% and the total power by less than 0.87%. This profiler contributes positively to improve processor single thread performance since it detects the code regions that contribute more to the program execution faster than other approaches.

Second, we have proposed an efficient hardware/software register checkpointing that requires half the hardware resources than other hardware techniques and that does not affect performance. The proposal makes use of already existing processor resources, such as the non-architectonic registers that are present in most of nowadays processor and it does not require additional hardware structures to be implemented. In particular, the proposal saves significant amount of power and area when compared to a traditional shadow register file approach and it is only 1% slower. Although the small performance degradation, the reduction in the area and power makes a processor implemented with this scheme a more appealing candidate for a multiprocessor approach.

Finally, we have presented a loop parallelization scheme, referred as LP, which improves the performance of simple in-order processors by 16% without requiring complex hardware

additions. The proposal detects loop codes and optimizes them to be executed in parallel by using two SMT threads. LP makes use of already existing processor resources. The hardware and the software collaborate to guarantee correct loop execution. Threads communicate register values in a synchronized manner and memory dependences are tracked to guarantee correct program order execution. This technique is a clear example on how single thread performance can be improved without impacting processor hardware requirements by employing hardware/software co-designed techniques.

These three techniques cover three key points of the hardware/software co-designed processors main operational flow: the detection of hot code, its optimization, and its execution in the hardware. By means of these techniques we have improved the design of current hardware/software co-designed processors by reducing software and hardware overheads (LIU and HRC) and by improving performance (LP). In addition, LIU and HRC also allow the implementation of optimizations designed to continue improving performance.

Therefore, we believe that a multicore processor based on the technologies described in this thesis is a good approach to continue exploiting thread level parallelism without sacrificing single thread performance.

This chapter is organized as follows. In Section 6.1, we present the main contributions of this thesis, and in Section 6.2, we describe some recommendations about new areas to be explored to continue on the definition of an efficient hardware/software co-designed processor.

6.1 Original Contributions

The development of this thesis has generated the following original contributions:

- A detailed classification on how basic blocks execute has been performed for the LIU Profiler. This classification has served to define which basic blocks are of higher priority for the profiler and need to be detected as soon as possible for optimization.
- A novel replacement policy for the LIU Profiler has been proposed. This new replacement policy is called LIU (from Least Intensively Used) and it combines the recency and the frequency of the accesses. It has been designed by taking into account the aforementioned basic block classification proposed in this thesis. The LIU replacement policy outperforms the results that can be achieved when considering other traditional replacement policies for detecting hot code when

employed in a profiler hardware table. Moreover, the LIU replacement policy could be employed in other environments because of its open and general design (for instance, similar proposals are implemented for page replacement in Operating Systems and for web page replacing in servers). The LIU replacement policy has been patented [144].

- A novel hardware/software hot code profiler, referred as the LIU Profiler, has been designed and implemented. The LIU Profiler is able to detect hot code regions in a very fast and efficient manner by using a simple hardware table for code profiling and a software algorithm to build code regions for optimization. The proposal introduces minimal overheads in the system and it is able to achieve better coverage than other solutions that require more hardware resources for their implementation. Moreover, the LIU profiler presents high accuracy when detecting hot code, prioritizing the detection of basic blocks that contribute more to the coverage of the applications over those that contribute less.
- An efficient hardware implementation of the LIU replacement policy has also been proposed. We call this implementation as the pLIU replacement policy (p stands for *pseudo*). The pLIU replacement policy demonstrated similar coverage numbers than the original LIU when employed for detecting hot code but it requires significantly less hardware resources to be implemented in the processor. The LIU Profiler that makes use of the pLIU only requires 1% extra area overhead when implemented on top of an Intel® P54C processor.
- An efficient hardware/software register checkpointing scheme has been defined and implemented. This proposal does not require important hardware additions and it uses the software combined with the hardware to implement very efficient checkpointing and recovery mechanism. It makes use of already existing processor resources, such as the non-architectonic registers that are present in most of nowadays processors. The proposal saves significant amount of power and area when compared with other hardware solutions. Moreover, it is only 1% slower than these solutions when implemented on top of an Intel® Atom®-like processor.
- A novel loop parallelization scheme for exploiting ILP in simple in-order SMT processors has been proposed. The proposal decomposes the loops into multiple threads in order to speedup its execution by means of parallelizing them. It requires simple hardware enhancements to support fast and low-cost synchronization across the threads. The loop parallelization scheme significantly improves the execution of loops when compared to their sequential execution.

- An efficient and fast thread spawning generator technique has been developed for the loop parallelization scheme. This technique minimizes the delay from fetch to execution of the instruction belonging to the new spawned threads. We call this feature Spawn-at-fetch and it is crucial in the proposed loop parallelization to not suffer from slowdowns when executing small loops and loops that iterate just a few times.
- Finally, a fast register communication scheme for SMT threads has been presented. This proposal has been proposed for loop parallelization but it can also be used for executing speculative threads in code that is not inside loops. In this case, the same synchronization/communication mechanisms proposed in this thesis can be used to access register values from the different thread register files. Moreover, the register communication also involves a dynamic binding between threads and registers that can be used to specify live-outs registers. The communication scheme has been patented [145].

6.2 Future Work

As it has already been commented, the different techniques exposed in this manuscript may be subjected to some improvements and extensions. In this section, we summarize all of them briefly and we address the reader to the specific chapter of each technique to get more details.

Regarding the LIU profiler, it would be very interesting to understand the effects of multi-tasking in a real execution environment. Note that the LIU profiler has been studied in a single thread execution environment and that the context switch impact has been studied with a simple analytical model.

Moreover, the LIU profiler chapter only covers the part of detecting hot code but not the software required to build the final code regions. Therefore, as future work, it would be very interesting to study how the code regions can be built by employing the profiled information from the LIU profiler.

Regarding the hardware/software register checkpointing technique, we have emplaced as future work the interactions that the proposal may have with other optimization techniques that make use of non-architectural registers. Moreover, we are also interested on continuing exploring heuristics to improve the register allocation mechanisms that interact with the new inserted move instructions.

Regarding the loop parallelization scheme, it is necessary to continue working on heuristics to select the best optimizations to be applied to each particular loop. We have done these analyses in an offline manner and it would be good to analyze alternatives to make these decisions at run-time.

Moreover, the loop parallelization scheme can be extended to support more than two simultaneous threads, which is very appealing for loops that exhibit high parallelism between iterations.

Finally, we have proposed as future work to do not attach iterations and threads in a static manner for loop parallelization. In such a way, threads are able to execute as much consecutive iteration as possible during the time that the other thread is stalled.

Bibliography

[1] S. Shankland y M. Kanellos, «Intel to elaborate on new multicore processor,» 2003. [En línea]. Available: http://news.zdnet.co.uk/hardware/chips/0,39020354,39116043,00.htm.

[2] C. Farivar, «Intel Developers Forum roundup: four core now, 80 cores later,» 2006. [En línea]. Available: http://www.engadget.com/2006/09/26/intel-developers-forum-roundup-four-cores-now-80-cores-later/.

[3] D. Geer, «Chip Makers Turn to Multicore Processors,» nº May, 2005.

[4] S. Bisson, «Azul announces 192 core Java appliance,» 2006. [En línea]. Available: http://www.itpro.co.uk/servers/news/99765/azul-announces-192-core-java-appliance.html.

[5] AMD, «Multi-core processors - The next evolution in computing,» Advanced Micro Devices, Inc., 2005.

[6] Intel, «A new era of architectural innovation arrives with dual-core processors,» Technology@Intel Magazine, 2005.

[7] R. Kalla, B. Sinharoy y J. M. Tendler, «IBM Power5 chip: a dual-core multithreaded processor,» IEEE Micro, 2004.

[8] S. Palacharla, N. P. Jouppi y J. E. Smith, «Complexity-Effective Superscalar Processors,» de *Proceedings of the 24th annual international symposium on Computer architecture (ISCA '97)*, 1997.

[9] H. Zhong, S. A. Lieberman y S. A. Mahlke, «Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications,» de *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, 2007.

[10] C. Madriles, P. López, J. M. Codina, E. Gibert, F. Latorre, A. Martínez, R. Martínez y A. González, «Boosting Single-thread Performance in Multi-core Systems through

Fine-grain Multi-Threading,» de *The 36th International Symposium on Computer Architecture (ISCA '09)*, 2009.

[11] C. García, C. Madriles, J. Sánchez, P. Marcuello, A. González y D. M. Tullsen, «Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices,» de *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05)*, 2005.

[12] E. Ipek, M. Kirman, N. Kirman y J. F. Martinez, «Core fusion: accommodating software diversity in chip multiprocessors,» de *Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07)*, 2007.

[13] J. E. Smith y R. Nair, Virtual Machines: Versatile Platforms for Systems and Processes, Elsevier, 2005.

[14] J. Dehnert, «Transmeta Crusoe and Efficeon: Embedded VLIW as a CISC Implementation,» Vienna, 2003.

[15] T. Halfhill, *Transmeta Breaks x86 Low-Power Barrier. Microprocessor Report,* 2000.

[16] A. Klaiber, «The Technology Behind the Crusoe Processors,» 2000.

[17] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl y G. Chrysos, «ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors,» *Proceedings of the 30th annual Internation Symposium on Microarchitecture (MICRO 30),* pp. 292-302, 1997.

[18] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal y W.-m. W. Hwu, «A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization,» de *Proceedings of the 26th internation symposium on Computer Architecture (ISCA '99)*, 1999.

[19] A. Semin, «Intel Next Generation Nehalem Microarchitecture,» Intel Corporation, EMEA, 2009.

[20] D. Kanter, «AMD's Bulldozer Microarchitecture,» Real World Technologies, 26 August 2010. [En línea]. Available: http://www.realworldtech.com/bulldozer/.

[21] R. F. Cmelik, D. R. Ditzel, E. J. Kelly y C. B. Hunter, «Combining Hardware and Software to Provide an Improved Microprocessor». US Patent Patente 6,031,992, Feb. 2000.

[22] Y. Almog, R. Rosner, N. Schwartz y A. Schmorak, «Specialized Dynamic Optimizations for High-Performance Energy Efficient Microarchitecture,» de *Procs. of 4th International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[23] J. E. Smith, S. Sastry, T. Heil y T. M. Bezenek, «Achieving high performance via co-designed virtual machines,» de *Proceedings of the internationl Workshop on Innovative Architecture (IWIA '98)*, 1998.

[24] K. Ebcioglu, E. Altman, M. Gschwind y S. Sathaye, «Dynamic Binary Translation and Optimization,» *IEEE Transactions on Computers,* vol. 50, nº 6, pp. 529-548, 2001.

[25] T. M. Conte, B. A. Patel y J. S. Cox, «Using branch handling hardware to support profile-driven optimization,» *Proceedings of the 27th annual internationl symposium on Microarchitecture (MICRO 27),* pp. 12-21, 1994.

[26] T. M. Conte, K. N. Menezes y M. A. Hirsch, «Accurate and Practical Profile-Driven Compilation Using the Profile Buffer,» *Proceedings of the 29th Annual International Symposium on Microarchitecture,* 1996.

[27] T. Heil y J. E. Smith, «Relational profiling: enabling thread-level parallelism in virtual machines,» *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture (MICRO 33),* pp. 281-290, 2000.

[28] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan y C. Zilles, «Hardware atomicity for reliable software speculation,» de *Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07)*, 2007.

[29] E. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritts, P. Ledak, D. Appenzeller, C. Agricola y Z. Filan, «BOA: The Architecture of A Binary translation Engine,» IBM Research Report RC 21665 (97500), 1999.

[30] K. Ebcioglu y E. R. Altman, «DAISY: Dynamic Compilation for 100% Architectural Compatibility,» de *Proc. of the 24th International Symposium on Computer Architecture (ISCA '97)*, 1997.

[31] J. E. Smith y R. Nair, «An Overview of Virtual Machine Architectures,» de *Virtual Machines: Architectures, Implementations and Applications*, Morgan Kaufmann Publishers, 2004.

[32] Intel, «Intel 64 and IA-32 Architectures Optimization Reference Manual,» 2011.

[33] Wikpedia, «Windows RT,» [En línea].

[34] Wikipedia, «Virtualization,» [En línea]. Available: http://en.wikipedia.org/wiki/Virtualization.

[35] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli y J. Yates, «FX!32: A Profiler-Directed Binary Translator,» de *IEEE Micro (18)*, 1998.

[36] V. Bala, E. Duesterwald y S. Banerjia, «Dynamo: A transparent dynamic optimization system,» de *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2000.

[37] D. Bruening, T. Garnett y S. Amarasinghe, «An Infrastructure for Adaptive Dynamic Optimization,» de *International Symposium on Code Generation and Optimization (CGO '03)*, 2003.

[38] Wikipedia, «Binary translation,» [En línea]. Available: http://en.wikipedia.org/wiki/Binary_translation.

[39] L. Harrison, «Transmeta Crusoe,» Processor Presentation Series, 2005.

[40] Wikipedia, «Input/Output,» 2012. [En línea]. Available: http://en.wikipedia.org/wiki/Input/output.

[41] W.-m. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm y D. M. Lavery, «The super block: An effective technique,» *The Journal of*

Supercomputig - Special issue on instruction-level parallelism, vol. 7, nº 1-2, pp. 229-248, 1993.

[42] M. C. Merten, A. R. Trick y R. D. Barnes, «An architectural framework for runtime optimization,» *IEEE Transactions on Computers,* vol. 50, nº 6, pp. 567-589, 2001.

[43] G. Ammons, T. Ball y J. R. Larus, «Exploiting Hardware Performance Counters With Flow and Context Sensitive Profiling,» de *Prog. Lang. Design and Impl*, 1997.

[44] S. Eranian, «The hardware-based performance monitoring interface for Linux,» [En línea]. Available: http://perform2.sourceforge.net.

[45] R. S. Cohn, D. W. Goodwin y P. G. Lowney, «Optimizing Alpha Executables on Windows NT with Spike,» *Digital Technical Journal,* pp. 3-20, 1998.

[46] E. Duesterwald y V. Bala, «Software profiling for hot path prediction: Less is more,» de *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[47] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars y B. R. Childers, «Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems,» de *International Symposium on Code Generation and Optimization (CGO'07)*, 2007.

[48] H.-S. Kim y J. E. Smith, «Hardware Support for Control Transfers in Code Caches,» de *36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, 2003.

[49] K. Hazelwood y M. D. Smith, «Code Cache Management Schemes for Dynamic Optimizers,» de *Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architecture (INTERACT '02)*, 2002.

[50] R. M. Karp, «Reducibility Among Combinatorial Problems,» de *Complexity of Computer Computations*, 1972.

[51] K. Hazelwood y M. D. Smith, «Generational Cache Management of Code Traces in Dynamic Optimization Systems,» de *Proceedings of the 36th International*

Symposium on Microarchitecture (MICRO 36), San Diego, CA, 2003.

[52] J. A. Baiocchi, B. R. Childers, J. W. Davidson y J. D. Hiser, «Reducing Pressure in Bounded DBT Code Caches,» de *International Conference on Compilers Architecture and Synthesis for Embedded (CASES 08)*, 2008.

[53] J. A. Baiocchi, B. R. Childers, J. W. Davidson, J. D. Hiser y J. Misurda, «Fragment Cache Management for Dynamic Binary Translators in Embedded Systems with Scratchpad,» de *International Conference on Compilers Architecture and Synthesis for embedded (CASES 07)*, 2007.

[54] K. Hazelwood y M. D. Smith, «Managing bounded code caches in dynamic binary optimization systems,» de *ACM Transactions on Architecture and Code Optimization (TACO)*, 2006.

[55] Y. Sun y W. Zhang, «Improving code caching performance for Java applications,» de *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS '08)*, Miami, FL, 2008.

[56] K. Casey, M. A. Ertl y D. Gregg, «Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters,» de *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2007.

[57] D. F. Bacon, S. L. Graham y O. J. Sharp, «Compiler transformations for high-performance computing,» *ACM Computing Surveys,* pp. Volume 26, Issue 4, 1994.

[58] J. Cocke, «Global Common Subexpression Elimination,» de *Proceedings of a Symposium on Compiler Optimization*, 1970.

[59] S. S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997.

[60] A. V. Aho, R. Sethi y J. D. Ullman, Compilers: Principles, Techniques and Tools, 1986.

[61] M. D. Smith, N. Ramsey y G. Holloway, «A Generalized Algorithm for Graph-Coloring Register Allocation,» de *Proceedings of the ACM SIGPLAN 2004*

conference on Programming Language Design and Implementation, 2004.

[62] M. C. Golumbic, R. B. Dewar y C. F. Goss, «Macro Substitutions in MICRO SPITBOL - a Combinatorial Analysis,» de *Proceedings of the 11th Southeastern Conference on Combinatorics, Graph Theory and Computing, Congressus Numerantium, Utilitas Math*, Winninpeg, Canada, 1980.

[63] J. E. Smith y A. R. Pleszkun, «Implementation of Precise Interrupts in Pipelined Processors,» de *Proceedings of the 12th Annual International Symposium on Computer Architecture (ISCA '85)*, 1985.

[64] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan y C. Zilles, «Hardware Atomicity for Reliable Software Speculation,» de *Proceedings of the 34th annual international symposium on Computer Architecture (ISCA'07)*, San Diego, 2007.

[65] W. Ahn, Y. Duan y J. Torrellas, «DeAliaser: Alias Speculation Using Atomic Region Support,» de *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS'13)*, 2013.

[66] H. Akkary, R. Rajwar y S. T. Srinivasan, «Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors,» de *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, 2003.

[67] M. J. Wing y G. P. D'Souza, «Gated Store Buffer for an Advanced Microprocessor». US Patente 6011908, 2000.

[68] K. Hazelwood y J. E. Smith, «Exploring code cache eviction granularities in dynamic optimization systems,» de *International Symposium on Code Generation and Optimization (CGO '04)*, 2004.

[69] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson y S. Lie, «Unbounded Transactional Memory,» de *HPCA '05 Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[70] M. Herlihy y J. E. B. Moss, «Transactional memory: architectural support for lock-free data structures,» de *Proceedings of the 20th annual international symposium on*

computer architecture (ISCA '93), 1993.

[71] R. Rajwar, M. Herlihy y K. Lai, «Virtualizing Transactional Memory,» de *Proceedings of the 32nd annual international symposium on Computer Architecture (ISCA '05)*, 2005.

[72] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift y D. A. Wood, «LogTM-SE: Decoupling Hardware Transactional Memory from Caches,» de *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, 2007.

[73] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic y J. Torrellas, «Cherry: Checkpointed Early Recycling in Out-of-order Microprocessors,» de *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture (MICRO 35)*, 2002.

[74] Wikipedia, «Atomicity (Database Systems),» Wikipedia, [En línea]. Available: http://en.wikipedia.org/wiki/Atomicity_(database_systems).

[75] G. Rozas, A. Klaiber, D. Dunn, P. Serris y L. Shah, «Supporting Speculative Modification in a Data Cache». United States Patent Patente 7,225,299, May 2007.

[76] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi y K. Hazelwood, «Pin: Building customized program analysis tools with dynamic instrumentation,» de *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '05)*, 2005.

[77] N. Nethercote y J. Seward, «Valgrind: A framework for heavyweight dynamic binary instrumentation,» de *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '07)*, 2007.

[78] T. Ball y J. R. Larus, «Efficient Path Profiling,» *Proceedings of the 29th Annual International Symposium on Microarchitecture,* pp. 46-57, 1996.

[79] J. R. Larus, «Whole Program Paths,» de *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation (PLDI '99)*, 1999.

[80] S. Tallam, X. Zhang y R. Gupta, «Extending Path Profiling across Loop Backedges and Procesdure Boundaries,» de *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*, 2004.

[81] M. Arnold y B. G. Ryder, «A Framework for Reducing the cost of Instrumented Code,» *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation,* pp. 168-179, 2001.

[82] R. Joshi, M. D. Bond y C. B. Zilles, «Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems,» *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04),* pp. 239-250, 2004.

[83] T. Yasue, T. Suganuma, H. Komatsu y T. Nakatani, «An Efficient Online Path Profiling Framework for Java Just-In-Time Compilers,» de *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, 2003.

[84] T. M. Conte, B. A. Patel, K. N. Menezes y J. S. Cox, «Hardware-Based Profiling: An Effective Technique for Profile-Driven Optimization,» *International Journal of Parallel Programming,* pp. 187-206, 1996.

[85] R. Nair y T. J. Watson, «Dynamic path-based branch correlation,» de *Proceedings of the 28th annual internation symposium on Microarchitecture (MICRO 28)*, 1995.

[86] P. P. Chang, S. A. Mahlke y W.-m. W. Hwu, «Using profile information to assist classic code optimizations,» *Software - Practice and Experience,* vol. 21, nº 12, pp. 1301-1321, 1991.

[87] E. Rotenberg, S. Bennet y J. E. Smith, «Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching,» de *Proceedings of the 28th annual ACM/IEEE international Symposium on Microarchitecture (MICRO 29)*, 1996.

[88] S. J. Patel y S. S. Lumetta, «rePLay: A Hardware Framework for Dynamic Program Optimization,» *IEEE Transactions on Computers,* vol. 50, nº 6, pp. 590-608, 2001.

[89] B. Sprunt, «Pentium 4 Performance Monitoring Features,» *IEEE Micro,* vol. 22, nº

4, pp. 72-82, 2002.

[90] S. M. Inc, «UltraSPARC User's Manual,» 1997.

[91] J. L. Henning, «SPEC CPU2006 benchmark descriptions,» *ACM SIGARCH Computer Architecture News,* vol. 34, nº 4, pp. 1-17, 2006.

[92] J. L. Henning, «SPEC CPU2000: Measuring CPU Performance in the New Millennium,» *Journal,* vol. 33, nº 7, pp. 28-35, 2000.

[93] S. Hu y J. E. Smith, «Reducing Startup Time in Co-Designed Virtual Machines,» de *Proceedings of the 33rd annual international symposium on Computer Architecture (ISCA '06)*, 2006.

[94] A. Shankar, S. S. Sastry, R. Bodík y J. E. Smith, «Runtime specialization with optimistic heap analysis,» de *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*, 2005.

[95] S. Hu y J. E. Smith, «Using Dynamic Binary Translation to Fuse Dependent Instructions,» de *Proceedings of the international symposium on Code generation and optimization (CGO '04)*, 2004.

[96] B. Fahs, T. Rafacz, S. J. Patel y S. S. Lumetta, «Continuous Optimization,» de *Proceedings of the 32nd annual international symposium on Computer Architecture (ISCA '05)*, 2005.

[97] T. Ball y J. R. Larus, «Optimally Profiling and Tracing Programs,» *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 16, nº 4, pp. 1319-1360, 1994.

[98] R. Love, «The Linux Process Scheduler,» *informIT,* 2003.

[99] A. Gal y M. Franz, «Incremental Dynamic Code Generation with Trace Trees,» Technical Report, 2006.

[100] M. Jaragh y A. Hasswa, «Implementation, Analysis and Performance Evaluation of the IRP-Cache Replacement Policy,» *Microelectronics Reliability Journal,* 2005.

[101] D. Lee, J. Choi, H. Choe, S. H. Noh, S. L. Min y Y. Cho, «Implementation and performance evaluation of the LRFU replacement policy,» *EUROMICRO 97,* pp. 106-111, 1997.

[102] Wikipedia, «Priority Encoder,» [En línea]. Available: http://en.wikipedia.org/wiki/Priority_encoder.

[103] Wikipedia, «Decoder,» [En línea]. Available: http://en.wikipedia.org/wiki/Decoder.

[104] Wikipedia, «Intel P54C Pentium Processor,» [En línea]. Available: http://en.wikipedia.org/wiki/Pentium.

[105] I. Corporation, «Pentium® Processor Datasheet,» Intel Corporation, 1997.

[106] S. Wilton y N. P. Jouppi, «An Enhanced Access And Cycle Time Model for On-Chip Caches,» Technical Report 93/5, DEC WRL, 1994.

[107] Intel, «Intel(r) Atom TM Processor 230 Series,» Intel Corporation, 2010.

[108] I. Corporation, «Intel(r) Atom(tm) Processor Z5xx Series,» Intel Corporation, 2011.

[109] M. Gschwind y E. R. Altman, «Precise Exceptions Semantics in Dynamic Compilation,» Procs. of the 2002 Symposium on Compiler Construction (CC '02), 2002.

[110] K. Krewell, «Transmeta gets more efficeon,» Micro-processor Report, 2003.

[111] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber y J. Mattson, «The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges,» de *International Symposium on Code Generation and Optimization (CGO '03)*, 2003.

[112] E. Altman, K. Ebcioglu, M. Gschwind y S. Sathaye, «Efficient Instruction Scheduling with Precise Exceptions,» IBM Research Report RC 22957 (97495), 1999.

[113] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture, Intel Corporation, 2009.

[114] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel y S. S. Lumetta, «Performance Characterization of a Hardware Mechanism for Dynamic Optimization,» de *Procs. of 34th International Symposium On Microarchitecture (MICRO 34)*, 2001.

[115] R. Rosner, Y. Almog, M. Moffie, N. Schwartz y A. Mendelson, «Power Awareness through Selective Dynamically Optimized Traces,» de *Proceedings of the 31st International Symposium on Computer Architecture (ISCA 21)*, 2004.

[116] S. Hack, D. Grund y G. Goos, «Register Allocation for Programs in SSA-Form,» de *Proceedings of the 15th International Conference on Compiler (CC'06)*, 2006.

[117] G. Chen y M. D. Smith, «Reorganizing Global Schedules for Register Allocation,» de *ICS'99 Proceding of the 13th International Conference on Supercomputing*, 1999.

[118] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal y W.-m. W. Hwu, «Compiler code transformations for superscalar-based high performance systems,» de *Proceedings of the 1992 ACM/IEEE conference on Supercomputing (Supercomputing '92)*, 1992.

[119] J. W. Davidson y S. Jinturkar, «Memory access coalescing: a technique for eliminating redundant memory accesses,» de *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94)*, 1994.

[120] V. H. Allan, R. B. Jones, R. M. Lee y S. J. Allan, «Software Pipelining,» *ACM Computing Surveys (CSUR),* vol. 27, nº 3, pp. 367 - 432, 1995.

[121] S. Balakrishnan y G. Sohi, «Program Demultiplexing: Data flow based Speculative Parallelization of Methods in Sequential Programs,» de *Proceedings of the International Symposium on Computer Architecture*, 2006.

[122] M. Cintra, J. Martinez y J. Torrellas, «Architectural Support for Scalable Speculative Parallelization in Shared Memory Systems,» de *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[123] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery y J. Shen, «Speculative Precomputation: Long Range Prefetching of Delinquent Loads,» de

Proceedings of the 28th International Symposium on Computer Architecture, 2001.

[124] T. Johnson, R. Eigenmann y T. Vijaykumar, «Min-Cut Program Decomposition for Thread-Level Speculation,» de *Proceedings of Conference on Programming Language Design and Implementation*, 2004.

[125] P. Marcuello y A. González, «Thread-Spawning Schemes for Speculative Multithreaded Architectures,» de *Proceedings of the Symposium on High Performance Computer Architectures*, 2002.

[126] T. Ohsawa, M. Takagi, S. Kawahara y S. Matsushita, «Pinot: Speculative Multi-threading Processor Architecture Exploiting Parallelism over a wide Range of Granularities,» de *Proceedings of the 38th International Symposium on Microarchitecture*, 2005.

[127] N. Vachharajani, R. Rangan, E. Raman, M. Bridges, G. Ottoni y D. August, «Speculative Decoupled Software Pipelining,» de *Proceedings of the Conference on Parallel Architecture and Compilation Techniques*, 2007.

[128] C. Zilles y G. Sohi, «2001,» de *Proceedigns of the 28th International Symposium on Computer Architecture*, Execution-Based Prediction Using Speculative Slices.

[129] C. Zilles, S. Lieberman y S. Mahlke, «Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications,» de *Proceedings in the International Symposium on High Performance Computer Architecture*, 2007.

[130] C. Madriles, P. López, J. Codina, E. Gibert, F. Latorre, A. Martínez, R. Martínez y A. González, «Anaphase: A fine-grain thread decomposition scheme for speculative multithreading,» de *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[131] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm y D. Tullsen, «Simultaneous multithreading: a platform for next-generation processors,» de *IEEE Micro*, 1997.

[132] F. Warg y P. Stenstrom, «Dual-thread Speculation: A Simple Approach to Uncover Thread-level Parallelism on a Simultaneous Multithreaded Processor,» de *International Journal of Parallel Programming*, 2008.

[133] S. Wallace, B. Calder y D. M. Tullsen, «Threaded multiple path execution,» de *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[134] H. Akkary y M. A. Driscoll, «A dynamic multithreading processor,» de *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 31)*, 1998.

[135] P. Marcuello y A. González, «Exploiting speculative thread-level parallelism on a SMT processor,» de *Proceedings of the International Conference on High Performance Computing and Networking*, 1999.

[136] I. Park, B. Falsafi y T. N. Vijaykumar, «Implicitly-multithreaded processors,» de *Proceedings of the 30th annual international symposium on Computer architecture (ISCA'03)*, 2003.

[137] V. Packirisamy, S. Wang, A. Zhai, W.-C. Hsu y P.-C. Ywe, «Supporting speculative multithreading on simultaneous multithreaded processors,» de *Proceedings of the 13th international conference on High Performance Computing (HiPC'06)*, 2006.

[138] A. Zhai, C. B. Colohan, J. G. Steffan y T. C. Mowry, «Compiler optimization of scalar value communication between speculative threads,» de *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems (ASPLOS X)*, 2002.

[139] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum y C. D. Plychronopoulos, «On the Perofrmance Potential of Different Types of Speculative Thread-Level Parallelism,» de *Proceedings of the 20th annual international conference of Supercomputing*, 2006.

[140] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank y R. A. Bringmann, «Effective Compiler Support for Predicated Execution Using the Hyperblock,» de *Proceedings of the 25th annual International Symposium on Microarchitecture (MICRO 25)*, 1992.

[141] A. Zhai, C. B. Colohan, J. G. Steffan y T. C. Mowry, «Compilter Optimization of Scalar Value Communication Between Speculative Threads,» de *Proceedings of the*

10th international conference on Architectural support for programming languages and operating systems (ASPLOS'02), 2002.

[142] B. H. Bloom, «Space/Time Trade-offs in Hash Coding with Allowable Errors,» *Communications of the ACM,* vol. 13, nº 7, pp. 422 - 426, July 1970.

[143] G. Gerosa, S. Curtis, M. D'addeo, B. Jiang, B. Kuttanna, F. Merchant, B. Patel, M. Taufique y H. Samarchi, «A Sub-1W to 2W Low-Power IA Processor for Mobile Internet Devices and Ultra-Mobile PCs in 45nm Hi-κ Metal Gate CMOS,» de *IEEE International Solid-State Circuits Conference (ISSCC 2008)*, 2008.

[144] P. Lopez, J. F. Sánchez, J. M. Codina, E. Gibert, F. Latorre, G. Magklis, P. Marcuello y A. González, «A Replacement Policy For Hot Code Detection». US Patente 201001155247, 6 May 2010.

[145] E. Gibert, J. M. Codina, F. Latorre, A. Piñeiro, P. López y A. González, «Communicating Between Multiple Threads In A Processor». US Patente 20100005277, 7 January 2010.

[146] V. J. Reddi, D. Connors, R. Cohn y M. D. Smith, «Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications,» de *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*, 2007.

[147] S. Gochman, A. Mendelson, A. Naveh y E. Rotem, «Introduction to Intel(r) Core TM Duo Processor Architecture,» Intel Technology Journal, 2006.

[148] L. Harrison, «Prof. Luddy Harrison Home Page,» 2005. [En línea]. Available: http://www.cs.uiuc.edu/homes/luddy/PROCESSORS/TransmetaCrusoe.pdf.

[149] T. Krazit, «Transmeta Hype Suffers Hardware Reality [PCWorld],» 6 September 2004. [En línea]. Available: http://www.pcworld.com/article/117685/article.html. [Último acceso: 14 February 2013].

[150] F. Warg y P. Stenstrom, «Dual-thread speculation: a simple approach to uncover thread-level parallelism on a simultaneous multithreaded processor,» de *International Journal of Parallel Programming*, 2008.

[151] J. L. Ayala y A. Veidenbaum, «Reducing Register File Energy Consumption using,» de *1st Workshop on Application Specific Processors (WASP) held in conjuntion with 35th Annual International Symposium on Microarchitecture*, Istanbul, Turkey, 2002.

[152] W. T. F. Encyclopedia, «Memory Disambiguation,» [En línea]. Available: en.wikipedia.org/wiki/Memory_disambiguation.