UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MATHEUS DA SILVA SERPA

Instruction-Aware Mapping (IAM): A Tool to Mitigate Functional Unit Contention in SMT Processors

Thesis presented in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Advisor: Prof. Dr. Philippe Olivier Alexandre Navaux
Co-advisor: Prof. Dr. Antonio Carlos Schneider Beck

Porto Alegre
September 2023

CIP — CATALOGING-IN-PUBLICATION

Serpa, Matheus da Silva

Instruction-Aware Mapping (IAM): A Tool to Mitigate Functional Unit Contention in SMT Processors / Matheus da Silva Serpa. – Porto Alegre: PPGC da UFRGS, 2023.

126 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2023. Advisor: Philippe Olivier Alexandre Navaux; Co-advisor: Antonio Carlos Schneider Beck.

1. SMT processors. 2. Performance degradation. 3. Resource sharing. 4. Functional unit contention. I. Navaux, Philippe Olivier Alexandre. II. Beck, Antonio Carlos Schneider. III. Título.

“Success is the ability to move
from one failure to another
without loss of enthusiasm.”
— Sir Winston Churchill

ACKNOWLEDGMENTS

I am writing to express my deep thankfulness to my advisors, Prof. Philippe Navaux and Prof. Antonio Beck. Their outstanding guidance and continuous support were crucial to me throughout my journey.

Also, I want to demonstrate my deep gratitude to my parents, Airton and Maria. Their constant support has always been my strong foundation.

To my fiancée, Luana, I will always be thankful for your patience and help during difficult times.

Finally, thanks go to all my colleagues and collaborators at the GPPD research group for their comments, discussions and support for my research.

My sincere thanks to everyone for helping me achieve my success.

ABSTRACT

Modern computing architectures frequently rely on Simultaneous Multithreading (SMT) processors to boost computational throughput and handle parallel applications efficiently. However, the potential of SMT can be compromised by functional unit contention when parallel threads execute similar instructions on the same core. Addressing this issue, this thesis introduces an Instruction-Aware Mapping (IAM) tool that mitigates functional unit contention and enhances resource utilization. Distinct from current solutions, IAM uses a dynamic, transparent mapping strategy that assigns threads to SMT cores based on their real-time instruction profiles, eliminating the need to alter the application source code. The performance of the IAM tool was evaluated using the well-known NAS Parallel Benchmarks (NPB) and Standard Performance Evaluation Corporation (SPEC) benchmarks, as well as SMT-Bench, a microbenchmark developed for SMT performance analysis. These evaluations, conducted on Advanced Micro Devices (AMD) and Intel processors, show an average geometric mean performance increase of 9.8% compared to the Linux operating system scheduler, performing well against round-robin and random mapping methods. IAM's effectiveness is instruction-specific, offering marked performance improvements for compute-centric operations, such as integer, floating-point, and branch instructions. At the same time, its influence is more moderate for memory-bound instructions, particularly load operations. These insights emphasize the importance of dynamic, instruction-specific strategies that can cater to the distinct characteristics of workloads in enhancing SMT performance. This research provides insights that can inspire more in-depth studies into adaptive methods for SMT optimization.

Keywords: SMT processors. Performance degradation. Resource sharing. Functional unit contention.

Mapeamento Instruction-Aware (IAM): Uma Ferramenta para Mitigar a Contenção nas Unidades Funcionais de Processadores SMT

RESUMO

As arquiteturas de computação recorrem a processadores Simultaneous Multithreading (SMT) para melhorar o throughput computacional e gerenciar aplicações paralelas. No entanto, a efetividade do SMT pode ser comprometida pela disputa de unidades funcionais quando threads paralelas executam instruções similares no mesmo núcleo. Em resposta a isso, esta tese introduz a ferramenta Instruction-Aware Mapping (IAM), que mitiga a disputa de unidades funcionais e otimiza a utilização de recursos. Ao contrário de outras soluções, a IAM utiliza uma estratégia de mapeamento dinâmica e transparente que atribui threads aos núcleos SMT com base em seus perfis de instrução em tempo real, sem necessidade de alterar o código-fonte da aplicação. A performance da ferramenta IAM foi testada usando os benchmarks NAS Parallel Benchmarks (NPB) e Standard Performance Evaluation Corporation (SPEC), além do SMT-Bench, um microbenchmark focado na análise de desempenho SMT. Essas avaliações, conduzidas em processadores Advanced Micro Devices (AMD) e Intel, mostram um aumento na média geométrica de desempenho de 9,8% em relação ao scheduler do sistema operacional Linux, com desempenho comparável às implementações round-robin e a outras estratégias de mapeamento. A eficácia da IAM é centrada na especificidade da instrução, mostrando melhorias no desempenho para operações como instruções de inteiro, de ponto flutuante e de desvio, enquanto seu impacto é mais moderado para instruções relacionadas à memória, como operações de carga. Essas descobertas ressaltam a importância de estratégias dinâmicas e adaptativas que levem em conta a natureza das instruções na otimização do desempenho SMT. Este estudo proporciona uma base para pesquisas adicionais sobre métodos adaptativos em SMT.

Palavras-chave: Processadores SMT. Degradação de desempenho. Compartilhamento de recursos. Contenção nas unidades funcionais.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices
BOTS	Barcelona OpenMP Tasks Suite
BT	Block Tridiagonal
BWAVES	Blast Wave Simulation
CFD	Computational Fluid Dynamics
CFS	Completely Fair Scheduler
CG	Conjugate Gradient
CMT	Cache Monitoring Technology
CPI	Cycles Per Instruction
CPU	Central Processing Unit
EP	Embarrassingly Parallel
FADD	Floating-Point Addition
FFT	Fast Fourier Transform
FMA	Fused Multiply-Add
FMUL	Floating-Point Multiplication
FPU	Floating Point Unit
FT	Fourier Transform
GPPD	Parallel and Distributed Processing Group
HWLOC	Hardware Locality
HWPC	Hardware Performance Counter
IAM	Instruction-Aware Mapping
IBM	International Business Machines Corporation

ILBDC	Incompressible Lattice-Boltzmann Method
ILP	Instruction-Level Parallelism
IMAGICK	Image Manipulation
IPC	Instructions Per Cycle
IS	Integer Sort
L1	Level 1 Cache
L2	Level 2 Cache
L3	Level 3 Cache
LLC	Last Level Cache
LSU	Load-Store Unit
LU	Lower Upper Gauss-Seidel Solver
MD	Molecular Dynamics
MG	Multigrid
NAB	Nucleic Acid Builder
NAS	NASA Advanced Supercomputing
NASA	National Aeronautics and Space Administration
NPB	NAS Parallel Benchmarks
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi-Processing
OS	Operating System
PAPI	Performance Application Programming Interface
PPGC	Postgraduate Program in Computing
RR	Round-Robin
SHAP	SHapley Additive exPlanations
SMT	Simultaneous Multithreading
SOS	Sample, Optimize, Symbios

SP	Scalar Pentadiagonal
SPEC	Standard Performance Evaluation Corporation
SWIM	Shallow-Water Equations Solver
UA	Unstructured Adaptive Mesh
VM	Virtual Machine

LIST OF SYMBOLS

$\frac{A}{B}$	Division Operator
$\sum$	Summation
$ExecutionTime_{i,Baseline}$	Execution time of the i^{th} application using the baseline mapping method
$ExecutionTime_{i,Comparison}$	Execution time of the i^{th} application using the comparison mapping method
i	Index of the current application
N	Total number of applications

CONTENTS

1 INTRODUCTION

The proliferation of multicore systems and Simultaneous Multithreading (SMT) has been instrumental in shaping advancements in various fields. The evolution of computer architectures has augmented computational power significantly, facilitating the resolution of increasingly complex issues within modern application domains such as artificial intelligence, data science, big data, bioinformatics, quantum computing, cybersecurity, and large language models (BAKITA et al., 2021; XU et al., 2021; NARAYANAN et al., 2021; KASNECI et al., 2023).

In the current technological environment, many applications operate within shared computing platforms. The scalability of underlying architectures, therefore, becomes paramount. The advent of multicore systems promotes the parallel execution of multiple applications within a shared computational setting, bolstering scalability, resource utilization, and cost efficiency (LIU; CHEN, 2018; TAN; NADEAU; GAO, 2019; ROLOFF et al., 2019; TSAI; HSU; LIN, 2020). However, sharing computational resources introduces additional challenges, especially when considering the diverse needs of modern applications. This necessitates effective resource management to avert performance degradation as the number of simultaneously running applications escalates (ZHANG; CHENG; BOUTABA, 2018; WANG et al., 2019).

This management of scalability is intrinsically tied to the concept of utilization. The transition to multicore and SMT architectures aims to optimize resource utilization, enhancing overall system performance. However, these systems could face suboptimal usage without careful management and intelligent task allocation, potentially leading to significant resource underutilization and performance bottlenecks (PATTERSON, 2018; ASANOVIC et al., 2018; DALLY, 2019; GUPTA; PATRA, 2021; VENKATESH; PATRA, 2022).

Though a holistic approach to resource management across multicore and SMT architectures is essential, this work focuses on thread mapping concerning functional units. Consequently, we will not delve deeply into data mapping techniques, instead focusing on understanding and addressing contention issues linked to functional units.

Furthermore, such environments often grapple with functional unit contention issues. SMT facilitates concurrent issuing of instructions from multiple independent threads to numerous functional units, markedly amplifying resource utilization and overall performance (KALLA et al., 2010; LORENZON; FILHO, 2019; TULLSEN; EGGERS;

LEVY, 1995; WANG et al., 2020; FELIU et al., 2023). SMT's primary goal of enhancing hardware resource utilization could paradoxically lead to performance degradation due to contention for shared resources. This problem is especially pronounced in the context of functional units which handle data operations. These units could become bottlenecks when multiple threads vie for their use concurrently, leading to resource competition.

Addressing these challenges mandates the implementation of effective thread-to-core mapping strategies. Ideally, threads heavily utilizing the same functional units should be distributed to different cores, thereby minimizing contention and optimizing core resource usage. However, determining the optimal mapping can be intricate and computationally demanding, necessitating a nuanced understanding of application behavior and the underlying architecture. This complexity underscores the urgency of developing automated thread-mapping techniques.

Previous research has identified communication and cache memory contention on SMT processors as significant performance bottlenecks (CRUZ et al., 2014; FELIU et al., 2016; AKTURK; OZTURK, 2019; SERPA et al., 2019; GOMEZ et al., 2020; ZHOU; HU; XIONG, 2020; CHALL; PAUL, 2021; PAN; ZHAI, 2021; WANG; YIN; LI, 2021; RODRIGUEZ; ABELLA; CANAL, 2022; LIN et al., 2022; GAO et al., 2023; YIN; LI, 2023; LIU et al., 2023). Strategies to alleviate such bottlenecks have been proposed and validated by researchers, primarily focusing on multiple single-thread multiprogram workloads, thereby achieving measurable performance improvements. These studies emphasize the need to tackle functional unit contention, particularly when considering parallel processing applications. Functional unit contention occurs when threads from the same or different applications issue similar types of instructions (like arithmetic, memory access, or floating-point operations) that utilize the same functional units.

1.1 Motivation

Hence, devising novel thread mapping methodologies that account for functional unit contention is vital to drive SMT performance towards new horizons of performance and efficacy. However, such a task presents challenges due to performance degradation issues. To analyze the performance degradation in SMT-based systems, we employed a microbenchmark, described in Chapter 4, and examined three different scenarios:

Scenario A: A single thread runs on each core. This scenario avoids interference from co-runners, thus making all functional units in a core fully accessible to the thread.

However, this scenario only runs half the number of threads compared to the following two scenarios, thus reducing the overall system throughput.

Scenario B: Two threads run on the same core, executing the same type of instructions and thus stressing the same functional units.

Scenario C: Two threads run on the same core, but unlike the previous scenario, each thread executes different types of instructions, thus stressing different functional units.

Figure 1.1 presents the performance degradation resulting from resource contention on the different functional units when SMT is enabled. Scenario A serves as a baseline for this comparison. We considered the slowest thread (i.e., the last one to complete execution in cases of imbalance) in each scenario. It is also noteworthy that scenarios B and C always execute twice the number of threads compared to scenario A. Despite its longer total execution time, scenario B performs twice as much computation as scenario A. However, even though scenario C runs twice the number of threads as scenario A, their execution times are very similar. The geometric mean for scenarios B and C were 187.2% and 102.7%, respectively, implying that the overhead of running threads stressing the same functional units is almost double the execution time. Finally, scenario C is the best scenario, wherein multiple applications stressing different units are allocated on the same core, increasing throughput while maintaining an execution time compared to when only one thread operates per core.

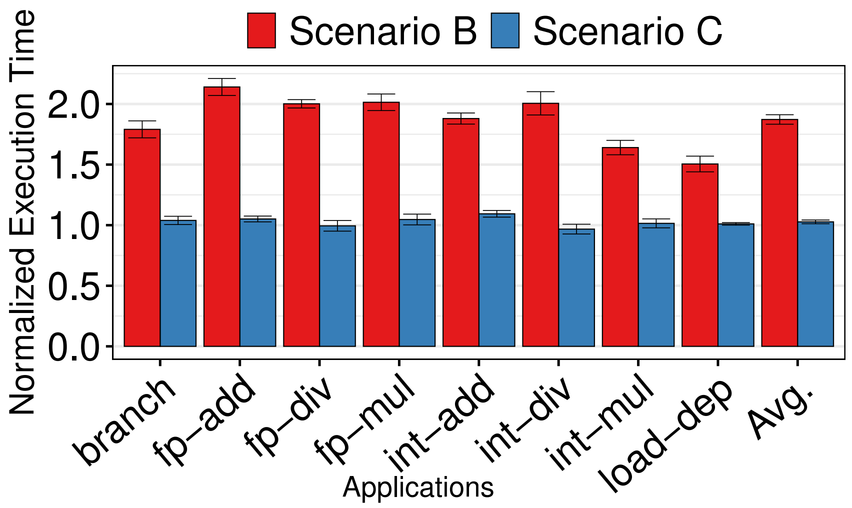


Figure 1.1 – Execution time for the different scenarios normalized to scenario A (100% on the figure).

From these results, we make the following observations:

- The type of operations carried out by each thread on a core directly impacts the core's performance and the overall efficacy of the application.
- While scenario A provides the best single-core performance, it only permits one thread per core, which is suboptimal in SMT-based systems (or any other type of multithreaded cores).
- Scenario C presents the most efficient thread-to-core mapping, with minimal per-thread performance loss and full utilization of all available virtual cores, thereby tapping into the full computing power.
- Scenario B experiences a performance degradation of up to 120%, which is unacceptable for performance-intensive applications. This underscores the importance of considering the workload characterization regarding the instruction type each thread executes when applying thread mapping in SMT-based processors.

Thus, the endeavor should be to develop a thread-to-core mapping close to scenario C to minimize functional unit contention and enhance performance. The present research seeks to contribute significantly by introducing a novel tool designed to map multiple parallel applications onto SMT processors.

1.2 Hypotheses and Objectives

As SMT processors advance modern computing, they face the challenge of functional unit contention. This arises when parallel threads with similar instructions compete for the same core resources. Recognizing the need to optimize SMT performance, we present the following hypotheses:

- Studying thread execution behavior and their associated instruction mix on a core can provide insights into potential contention for functional units.
- Mapping threads to cores considering the instruction mix, may reduce functional unit contention and improve overall system performance.

Given these hypotheses, the primary objective of this thesis is to develop mechanisms to mitigate the functional unit contention in computing platforms. We aim to achieve this objective through the following steps:

- We propose an Instruction-Aware Mapping (IAM) tool that can efficiently map threads to cores considering their instruction mixes. This tool minimizes contention and maximizes system performance in multicore and SMT environments.
- To evaluate the effectiveness of the IAM tool, we utilize different benchmarks and computing systems for our experimental studies.
- We also compare IAM against other existing strategies for thread-to-core mapping, demonstrating its unique strengths and potential for improving system performance in various computing scenarios.

Please note that while the broader field of SMT optimization often involves considerations such as memory or cache, this thesis aims explicitly to alleviate contention at the level of functional units. Consequently, our objectives and hypotheses are confined to this more narrow scope.

1.3 Contributions of this Thesis

This thesis proposes a novel solution to the abovementioned issues - the Instruction-Aware Mapping (IAM) tool. IAM is an online tool that leverages instruction-level information to optimize the mapping of multiple parallel applications onto cores. The core of the IAM tool lies in its ability to understand the workload functional unit characteristics in real time. It dynamically reads hardware performance counters to assess the instructions usage patterns, such as the number of floating-point, integer, branches, loads, and stores operations. This information enables IAM to map threads stressing identical functional units onto different cores intelligently.

The IAM tool aims to optimize the thread-to-core mapping so that functional units are utilized to their maximum potential without causing contention. It is achieved by minimizing the number of threads that simultaneously issue similar instructions to the same functional unit. By doing so, IAM improves the overall performance of SMT processors and ensures efficient usage of computational resources.

The contributions of this research are as follows:

- We develop and introduce SMT-Bench, a microbenchmark designed for strain-specific functional units. This benchmark allows us to evaluate the impact of resource sharing empirically. It provides crucial insights into SMT processors' performance and behavior under various workloads. We complement this with assessments using two widely recognized benchmarks, the NAS Parallel Benchmarks (NPB) and Standard Performance Evaluation Corporation (SPEC), offering a more comprehensive performance analysis across diverse workloads.

- We propose a dynamic, real-time, instruction-aware tool for mapping threads of multiple parallel applications onto cores, the IAM tool (SERPA et al., 2022). The tool leverages the distinctive instruction patterns of these applications, enabling an adaptive mapping strategy that responds to changing workload characteristics as they unfold.

1.4 Limitations and Challenges

While our research aims to bring substantial advancements in thread-to-core mapping for SMT processors, we acknowledge that our tool has inherent limitations and challenges.

The IAM tool heavily depends on the precision and comprehensiveness of topology and instruction-level information. Any inaccuracies in the gathered data may directly impact the efficiency of our thread-to-core mapping tool, leading to suboptimal usage of functional units. Furthermore, as the tool operates in an online environment, it may be susceptible to the dynamic nature of workloads, which can fluctuate in intensity and type over time. Efficiently managing these variations is a significant challenge in terms of overhead and hardware counters limitations.

We also acknowledge the limitations of our microbenchmark, SMT-Bench. While it is designed to stress specific functional units and provide valuable insights into the behavior of SMT processors, its representativeness of all potential workloads could be limited.

1.5 Document Organization

The structure of this thesis is as follows. Chapter 2 provides an in-depth overview of the background related to this research field. In Chapter 3, we survey relevant literature, highlighting significant studies and their relevance to our work. Chapter 4 delves into the specifics of the architecture that underpins our research, detailing the influence of resource sharing and the methodologies employed in evaluating these impacts. In Chapter 5, we present our innovative tool for the online mapping of multiple parallel applications on SMT processors. We delve into the underlying mechanisms and the methodology.

Following this, Chapter 6 delves into the design of our experimental study and presents our findings. We conducted experiments on Intel and AMD processors using SMT-Bench, NPB and SPEC benchmarks. The intent was to offer a robust and comprehensive evaluation of our tool. In these experiments, we compared our tool's performance against the default Linux scheduler, a round-robin mapping, and a random approach. The results significantly underscore the enhancements our tool provides over the comparative methods.

Chapter 7 draws together the key insights garnered from this thesis, discussing our findings and outlining potential directions for future research.

In addition to the main chapters, this thesis also includes two appendices. Appendix A houses the complete experimental results, providing a more exhaustive understanding of our tool's performance under various conditions. Lastly, Appendix B contains a comprehensive summary of this thesis in Portuguese, adhering to the requirement set by the PPGC Graduate Program in Computing.

2 BACKGROUND

A constant push for improved performance and efficiency drives modern computing. This has catalyzed the development of complex computing architectures and technologies. This chapter introduces these key concepts and shines a light on their complexities.

We focus on multicore architectures, an essential facet of contemporary computing. This discussion elucidates the transformative effect of having multiple cores within a single processing unit, enabling parallel processing and significantly augmenting computing speed and power.

The conversation then moves to SMT, which boosts processor efficiency. This approach allows multiple independent threads of execution to use the resources provided by today's processor architectures more effectively. SMT does introduce challenges and complexities, notably in terms of functional units and resource contention, topics we delve into at length.

Subsequently, we dissect the concept of functional units, highlighting their role and the persistent resource contention issue. This exploration underlines how these units can become bottlenecks, mainly when multiple threads compete for the same resources.

The later part of this chapter shifts the spotlight to thread-to-core mapping strategies. This discussion emphasizes the importance of effective mapping in leveraging the full potential of multicore and multithreaded architectures, underlining these strategies' role in optimizing core utilization and efficiently managing threads.

To conclude, we highlight the criticality of hardware performance counters and tools essential in performance measurement and diagnosis. We delve into their role in offering real-time insights into processor operations, identifying performance bottlenecks, and assisting in developing and optimizing resource management strategies.

By shedding light on these fundamental concepts, this chapter lays the groundwork for upcoming discussions on the intricacies of functional unit contention and exploration of our proposed solution. This journey is designed to give the reader a profound understanding of the modern computing landscape and potential solutions to resource contention.

2.1 Modern Computing

Modern computing is a cornerstone in the structure of computer science, enabling the design and use of increasingly powerful computational technologies designed to tackle complex, computationally intensive problems (DONGARRA et al., 2005). The transformative power of modern computing architectures has been showcased across various fields, including climate modeling, genetic sequencing, computational fluid dynamics, and large-scale data analytics, where traditional computing devices would struggle to process the enormous computational requirements within reasonable time frames.

Modern computing has experienced a significant paradigm shift with the advent of multicore processors and SMT technology. This transformation has been driven by the increasing difficulty of enhancing performance using single-core processors, thus spotlighting parallelism as the primary avenue for improving computational performance (HILL; MARTY, 2008; PADUA, 2011).

In this context, multicore systems consist of multiple physical processors, each incorporating various independent cores capable of executing instructions (BAUMANN et al., 2009). On the other hand, SMT is a technology that allows the simultaneous execution of multiple threads on a single core. Such architectures significantly boost computational performance.

However, these architectural advancements introduce new sets of challenges. One of the most prominent is the management of contention for functional units. Functional units are the components within a Central Processing Unit (CPU) core responsible for performing operations as per computer instructions. The efficiency and effective utilization of these units critically influence overall computational performance. In multicore and SMT environments, contention for these units can escalate into a severe performance bottleneck as multiple threads vie for limited execution resources, leading to potential underutilization and degradation of system performance (HENNESSY; PATTERSON, 2011).

Addressing this issue necessitates sophisticated resource management strategies, such as thread-to-core mapping, which dictate how threads are assigned to cores. An optimal strategy can significantly improve performance by reducing contention and maximizing resource utilization. However, traditional mapping strategies have their limitations and often need to consider the specific characteristics of the functional units and the unique requirements of the workload (TAM; AZIMI; STUMM, 2007a).

The IAM tool, which we propose, addresses these challenges directly by adopting a more dynamic and adaptable approach to thread-to-core mapping. At the heart of this tool is the idea of utilizing information about the instruction type of the workload and the contention status of functional units to make informed decisions regarding thread placement. The ultimate objective is optimizing resource utilization and reducing functional unit contention, enhancing overall system performance and efficiency.

2.1.1 Multicore Architectures

In the digital age, multicore systems have become ubiquitous, from the handheld smartphones we use daily to the massive servers underpinning the Internet's infrastructure. The design principle of incorporating multiple computing cores into a single processor chip emerged as a strategic solution to the escalating challenges of power consumption and heat dissipation associated with the relentless pursuit of higher clock speeds (HILL; MARTY, 2008). As the boundaries of frequency scaling became evident, the focus of the computing industry inevitably shifted towards multicore architectures as a route to sustain performance growth.

These systems facilitate the simultaneous execution of multiple threads, paving the way for greater throughput and, consequently, elevated overall system performance. However, the transition to multicore architectures has been fraught with hurdles. They have introduced new complexities in managing contention for shared resources, such as functional units (HENNESSY; PATTERSON, 2011).

Functional units constitute the operational heart of a CPU core, performing the actions mandated by instructions. In multicore environments where multiple threads are executed in parallel, these threads engage in a tug-of-war for the limited functional units. This contention can trigger performance degradation, forcing threads into a waiting state due to the limited availability of execution resources. This can create bottlenecks in processing, leading to inefficiencies and hampering overall processing throughput (LI et al., 2010).

Thus, understanding and mitigating resource contention in multicore architectures is integral to optimizing system performance. As we delve deeper into this chapter, we will discuss further how the proposed IAM tool addresses these challenges by promoting more informed and efficient thread-to-core mapping decisions.

2.1.2 Superscalar Processors

Superscalar processors represent a unique category of CPUs engineered to execute multiple instructions concurrently, utilizing various functional units. This capacity for simultaneous execution allows for the processing of non-dependent instructions in parallel, thus deriving the term superscalar for the architecture. This term is an analogy to a superhighway, which permits the concurrent passage of multiple vehicles (FLYNN; AKENINE-MöLLER; STRID, 1995).

In contrast to scalar processors, which are restricted to executing a single instruction per clock cycle, superscalar architectures break this boundary by handling multiple instructions within the same cycle. Adopting an Instruction-Level Parallelism (ILP) technique is pivotal to this performance advancement. ILP significantly enhances the efficiency of instruction processing by identifying multiple independent instructions and executing them concurrently, in opposition to the sequential approach taken by scalar processors (HENNESSY; PATTERSON, 2011).

Various strategies empower superscalar processors to achieve such a degree of parallelism. One involves employing an advanced instruction decoder capable of simultaneously analyzing and decoding multiple instructions. Once these instructions are decoded, they are distributed to different functional units within the processor, where they are processed independently and concurrently, capitalizing on the processor's full capacity (HENNESSY; PATTERSON, 2011).

Another noteworthy technique utilized in superscalar processors is speculative execution. This strategy allows the processor to predict and execute the subsequent instructions in advance. In doing so, it decreases the overall latency of the instruction processing pipeline, ensuring it remains occupied. If the speculatively executed instructions are unnecessary, the processor discards the calculated results and proceeds with the following instructions. This technique further optimizes resource utilization and increases throughput (HENNESSY; PATTERSON, 2011).

With their impressive computational capabilities, superscalar processors play an essential role in various applications. They excel in scientific simulations and database processing, where fast processing of large volumes of data is a primary requirement. Their usage extends beyond these demanding areas to commonplace devices. Desktop and laptop computers and mobile devices like smartphones and tablets often employ superscalar processors to enhance performance for resource-intensive applications, such as gaming and video editing. This wide range of applications reflects the significant role of superscalar processors in shaping the contemporary computing landscape.

2.1.3 Simultaneous Multithreading

SMT, also known as hyperthreading in Intel processors' context, is a common feature in modern CPU design. It is a product of continuously enhancing resource utilization and computational throughput. By allowing multiple threads to be active on a single core concurrently, SMT effectively employs execution resources that might otherwise be idle with a single thread's operation (TULLSEN; EGGERS; LEVY, 1995).

SMT's primary aim is to leverage the number of active threads on a core to increase the overall utilization of the core's execution resources. It accomplishes this by integrating multiple instruction streams of a unique thread onto the same processor core. Through this process, SMT technology cleverly conceals some latency originating from instruction dependencies and delays in memory access. Figure 2.1 provides a comparative overview of a superscalar processor and an SMT one. It is clear that while superscalar processors may have many unused functional units, SMT processors enhance resource utilization by executing instructions from multiple threads simultaneously.

However, the shift towards SMT comes with its set of challenges. Like multicore systems, implementing SMT introduces new difficulties, with the contention problem for functional units standing out significantly. As the number of concurrently active threads increases, the competition for these limited available units intensifies, potentially leading to performance bottlenecks. This issue is more pronounced in workloads where threads are interdependent, resulting in thread-level contention. This significant factor could hinder system performance (EGGERS et al., 1997).

Given these challenges, the demand for effective thread-to-core mapping strategies and intelligent workload management becomes more urgent. By addressing contention issues, we can fully harness SMT's potential. The proposed IAM tool, which we will discuss further in subsequent sections, aims to accomplish just that, signaling a new age of efficient and dynamic resource management in multicore and SMT environments.

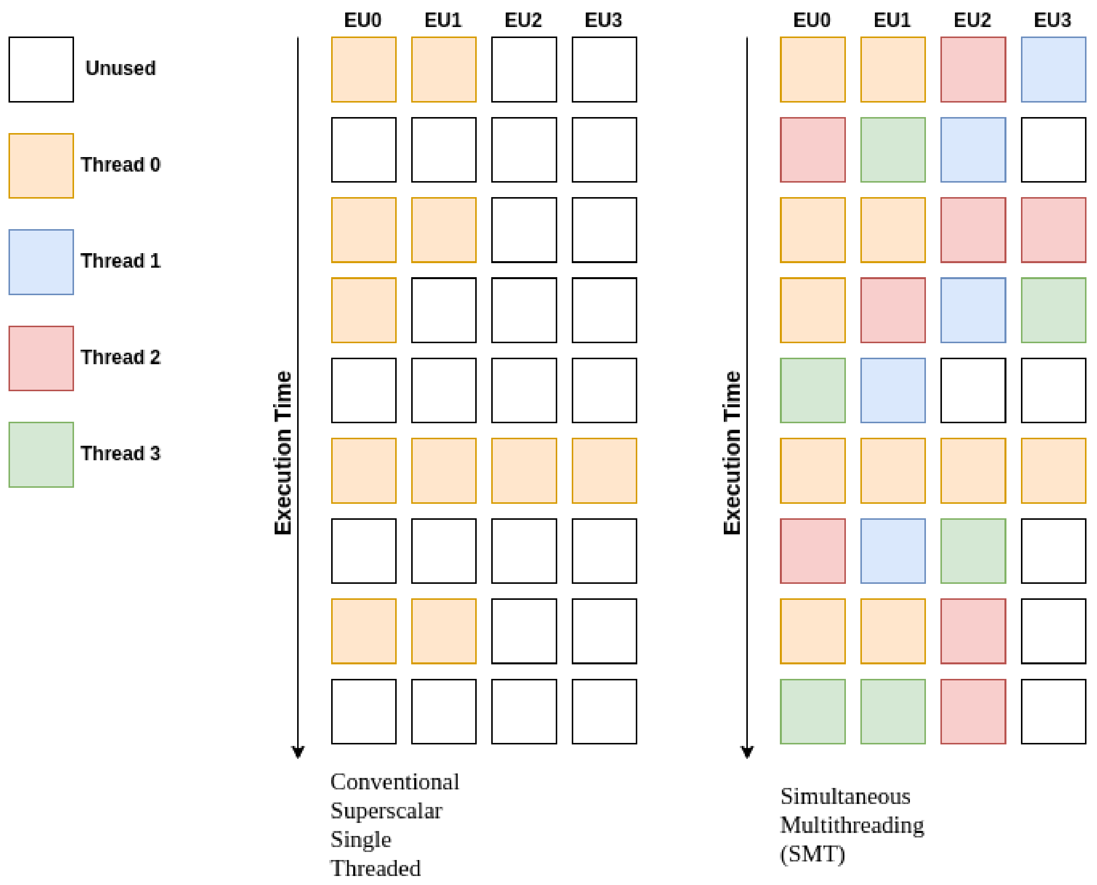


Figure 2.1 – Comparison between a superscalar and a simultaneous multithreading processor. The rows of squares represent the issue slots.

2.1.4 Functional Units and Resource Contention

At the crux of a CPU lie the functional units, the primary components tasked with executing the instructions that collectively form programs. These units encapsulate the vital parts of a processor, including the Arithmetic Logic Unit (ALU), Floating Point Unit (FPU), and Load-Store Unit (LSU), among others. From carrying out basic arithmetic and logical operations (the forte of the ALU) to tackling floating-point computations (the domain of the FPU) and handling memory read/write tasks (the responsibility of the LSU), these units have a wide array of operations under their belt. The synergistic interplay among these units and their operational efficiency significantly influence the comprehensive performance of a computing system (HENNESSY; PATTERSON, 2011).

In modern multicore processors and SMT processors like the AMD Zen. Functional units are shared resources among the two threads that can run on a core. This shared nature creates potential contention when multiple threads vie to execute simultaneously. Conflict emerges when multiple threads seek simultaneous access to a shared resource that cannot service all requests simultaneously. In the context of functional units, this scenario unfolds when an excessive number of threads demand similar computational tasks, leading to an overload of a specific functional unit (EGGERS et al., 1997)

Figure 2.2 elucidates the architecture of AMD Zen (CLARK, 2016; SINGH et al., 2017), a model demonstrating this contention scenario. In the Zen design, cores can execute two threads, courtesy of SMT (TULLSEN; EGGERS; LEVY, 1995; TULLSEN et al., 1996; EGGERS et al., 1997). Each core has private Level 1 (L1) and Level 2 (L2) caches. The L1 caches, divided into data and instruction cache, competitively share 32KB and 64 KB capacity between the threads on the same core. The L2 cache, L1 inclusive, reserves 512 KB per core. A Last Level Cache (LLC), shared among all cores, acts as an exclusive victim cache with 2048 KB allocated per core. The execution engine is split between integer and floating-point, housing four ALUs, two branch units, one integer mult, one integer div, and four 128-bit floating-point units divided into Fused Multiply-Add (FMA), Floating-Point Addition (FADD), and Floating-Point Multiplication (FMUL).

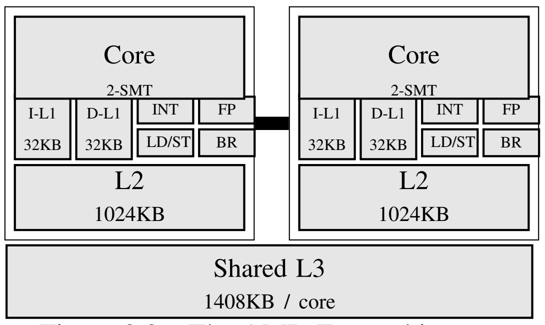


Figure 2.2 – The AMD Zen architecture.

Several factors, including the processor's architecture, the workload, and the thread scheduling policy employed by the operating system or runtime environment, can influence contention's severity. Despite some level of contention, the ability to execute multiple threads in parallel can lead to a higher overall throughput than sequential thread execution (TULLSEN et al., 1996). However, minimizing contention is a crucial step toward boosting performance.

Numerous strategies aim to manage contention for functional units, ranging from the relatively straightforward to the highly sophisticated. A simple method involves balancing the load across various functional units, which could entail scheduling threads such that their demands on different functional units are approximately equal. This can be daunting due to most workloads' dynamic and unpredictable nature.

A more advanced and informed approach involves designing intelligent scheduling algorithms that consider the system's current state, the status of functional units, and the characteristics of the awaiting threads. This strategy lies at the heart of our proposed IAM tool.

IAM strives to reduce contention and maximize the utilization of functional units by making informed scheduling decisions based on the instruction characteristics of the workload. By predicting the functional units likely to be in high demand based on the type of instructions in the incoming threads, IAM can map threads to cores to distribute the load evenly across all functional units.

IAM's overarching goal is to optimize CPU resource use, mitigate functional unit contention, and enhance overall computing performance. With the increasing trend towards multicore and SMT-enabled processors, the efficient management of functional unit contention using methods like IAM becomes paramount for performance optimization.

2.1.5 Resource Utilization in Computing Environments

The contemporary computing landscape is marked by utilizing on-demand resources, often facilitated through virtualization technologies, such as virtual machines (VMs) or containers. These resources span processing power, storage, networking, and more, providing flexible, scalable, and cost-effective solutions.

One of the primary advantages of these modern computing paradigms is the capacity to modulate the amount of computing resources per demand rapidly. This ensures that an application or service operating on such a platform can handle expanding workloads without compromising performance. When the workload diminishes, the resources can be scaled back, ensuring cost efficiency. Thus, adept usage of scalability is pivotal for optimizing performance and cost-effectiveness.

However, like in multicore systems and SMT, contention can pose a significant challenge in these environments. While conflict can arise at various levels, functional unit contention at the CPU level can persist as an essential concern. Multiple VMs or containers running on the same physical host may vie for the same functional units, leading to potential performance degradation. This contention problem could be complex, as users typically require more visibility or control over the underlying hardware resources.

Efficient resource utilization is critical to addressing contention in these scenarios. Resource utilization concerns maximizing the performance of applications given the available resources. Concerning functional units, this involves managing the assignment of workloads to cores and threads to minimize contention. As discussed earlier, our proposed solution, the IAM tool, could be leveraged to address this issue.

Currently, IAM is tailored for handling functional unit contention at the level of a single physical machine. However, the foundational principles of IAM could be adapted to broader contexts. For instance, a service provider could implement similar instruction-aware scheduling at the VM or container allocation level to reduce functional unit contention across multiple physical hosts. This strategy underscores how IAM's concepts could extend beyond single-machine environments, offering a pathway to more efficient resource utilization in the rapidly evolving landscape of on-demand computing environments.

2.2 Thread-to-Core Mapping Strategies

The process of thread-to-core mapping forms the bedrock of operating systems (OS) and the hardware they control. It enables the execution of parallel tasks on a multicore CPU (TAM et al., 2009). This process entails the assignment of threads to the system's available cores. An effective thread-to-core mapping strategy can substantially impact system performance, especially with thread synchronization, load balancing, and contention issues.

Classic thread-to-core mapping strategies encompass static and dynamic mapping. Static mapping involves assigning threads to cores at the start of the program execution and maintaining this assignment throughout. While simple and predictable, this strategy needs more adaptability to cope with dynamic changes in workload. Conversely, dynamic mapping allows the OS to reassign threads based on system workload and performance metrics. This offers flexibility but incurs overhead due to the continuous monitoring and reassignment processes (BLAGODUROV et al., 2010).

The mapping of threads to cores directly impacts functional unit contention. A poorly planned mapping strategy could result in multiple threads vying for the same functional units, negatively affecting system performance. For example, in SMT systems, two threads mapped onto a single physical core could contend for shared functional units. This contention can lead to performance degradation, particularly troublesome for performance-sensitive applications (SNAVELY; WOLTER; CARRINGTON, 2001).

However, given that hardware configurations can vary significantly, it is crucial to adapt the IAM tool accordingly. For instance, in systems with heterogeneous cores, an optimal mapping strategy might involve assigning threads with instruction mixes demanding higher computational resources to more powerful cores. Likewise, in systems with hardware multithreading, the process might involve cautiously assigning threads to different hardware threads to minimize contention for shared functional units (DELIMITROU; KOZYRAKIS, 2013; SAEZ et al., 2012).

The design and execution of effective thread-to-core mapping strategies can significantly influence the performance of multicore systems. Considering the types of instructions processed by each thread and utilizing this information to reduce functional unit contention—as proposed in the IAM tool—represents a promising approach to this issue. This strategy highlights the importance of sophisticated mapping techniques in optimizing performance and reducing contention in modern multicore architectures.

2.2.1 Hardware Performance Counters

Hardware Performance Counters (HWPCs) are inbuilt elements within processors that facilitate the tracking of system operations (ANDERSON; LAZOWSKA; LEVY, 1989). These counters permit monitoring numerous events, including the total executed instructions, cache hits and misses, branch predictions, and more. HWPCs are crucial in interpreting the processor's performance, identifying potential bottlenecks, and steering optimizations.

HWPCs offer valuable insights into the activity within the processor's functional units. For instance, HWPCs can reveal the frequency of usage of specific functional units, how often they encounter stalls due to data dependencies or cache misses, and how contention impacts performance. This information allows system architects and software developers to comprehend how effectively the functional units are used and identify factors impeding their optimal use (SHEN et al., 2008).

In diagnosing and alleviating functional unit contention, HWPCs are vital. They can recognize patterns such as increased cache misses or a high frequency of pipeline stalls, which can pinpoint contention between threads vying for the same functional units. Such information can steer optimization strategies that reduce contention, enhancing the system's overall performance (KAMBADUR et al., 2012).

HWPCs are central to the proposed IAM tool. The IAM tool relies on HWPCs to gather data on the instruction mix of each thread. This data empowers the tool to make intelligent decisions regarding thread-to-core mapping to minimize functional unit contention. Therefore, HWPCs provide the foundation for the IAM tool, enabling it to enhance system performance (MUCCI et al., 1999) effectively. In this context, we utilize the Performance Application Programming Interface (PAPI) library, which delivers a consistent interface and methodology for collecting performance counter data from a computer system. PAPI hardware counters are a library feature that provides direct access to the hardware-level counters in most modern microprocessors. These counters offer insights into low-level hardware events like cache misses, branch mispredictions, and cycles per instruction. By leveraging PAPI hardware counters, developers and system architects can measure these events to collect detailed performance data about their programs or systems. This data can assist in identifying bottlenecks, optimizing code, or more effectively balancing system workloads (BROWNE et al., 2000).

Despite their utility, using hardware performance counters presents several challenges. These challenges include a limited number of available counters, the overhead associated with performance monitoring, and the complexity of interpreting the data they provide. Additionally, counter data can be affected by several factors, such as system noise, resource contention, and scheduling policies, making it difficult to isolate the cause of performance issues. Consequently, using these counters should be well planned and supplemented with other system analysis and debugging tools (SHAMEEM; JASON, 2005).

Hardware performance counters are crucial tools for understanding the intricate operations within processors, identifying performance bottlenecks, and creating optimization strategies. Their critical role in diagnosing and mitigating functional unit contention, providing the data needed to guide optimizations like the IAM tool, is invaluable. However, the challenges associated with their usage highlight the need for further research and development in this area.

2.3 Summary

This chapter has meticulously examined functional unit contention in several computing contexts, including multicore systems, SMT, and superscalar processors. We highlighted the profound impact of contention in these areas. We underscored the critical role that intelligent thread-to-core mapping strategies and efficient resource utilization play in mitigating performance degradation associated with resource contention.

Our discussion began with multicore systems, where we demonstrated how contention for functional units might emerge when threads running on separate cores compete for shared resources. This contention could limit the processing speed and efficiency of these systems. We then introduced the concept of superscalar processors, an architectural design that allows for the concurrent execution of multiple instructions. Despite significantly augmenting processing power, these systems can also experience contention when multiple instructions compete for the same functional units.

Next, we turned our attention to SMT, where the contention for functional units becomes a reality due to the concurrent execution of multiple threads on the same physical core. However, by intelligently mapping threads to cores or hardware threads, considering the mix of instructions, minimizing such contention and maximizing system performance is possible.

We explored various thread-to-core mapping strategies, emphasizing their direct influence on functional unit contention and the subsequent impact on overall system performance. Our investigation covered static and dynamic mapping strategies, outlining the strengths and limitations of each approach, and highlighted the need for a strategy that incorporates both best features.

Finally, we emphasized the critical role of hardware performance counters in diagnosing and mitigating functional unit contention. These counters provide valuable insights into the system's inner workings and form the foundation of the proposed IAM tool. Despite some challenges associated with their use, hardware performance counters are invaluable tools in our approach to effectively address functional unit contention.

In the upcoming chapter, we will conduct a comprehensive survey of related work in this field. We will compare our proposed IAM tool with existing strategies for managing contention and maximizing performance in multicore, superscalar, and SMT environments. This comparative analysis will further highlight the unique strengths and potential of the IAM tool in enhancing performance across various computing systems.

3 RELATED WORK

Resource contention in SMT processors has long been recognized as a significant contributor to performance bottlenecks. The literature in this field is rich with varied strategies and methodologies to overcome these challenges. This chapter aims to comprehensively survey these works, encompassing different facets of the problem and the proposed solutions.

We consider a multifaceted classification for these studies to render a meaningful exploration of the subject. We delve into the nuances of how these solutions operate, whether they focus on multithreaded or single-threaded applications, as this distinction can significantly influence their effectiveness. Recognizing that the context of operation plays a critical role, we also examine if these methods are designed for real or simulated environments.

A noteworthy consideration is the requirement for hardware modifications in several of these solutions. While some hardware-based approaches offer promising results, the need for changes poses challenges regarding practicality and feasibility. It becomes even more complex when these alterations are specific to certain architectures, limiting the solution's universality.

We further evaluate the solutions based on their dependency on architecture. While effective in specific environments, architecture-dependent solutions may need more versatility in diverse architectural landscapes. Thus, architecture-independent solutions that maintain effectiveness across various platforms hold a distinctive advantage in broad application.

In addition, the timing of applying these solutions is another critical factor. We scrutinize whether these methods are designed for online execution used in real-time during process execution for offline execution, implemented when the system is not actively running tasks.

We organize our discussion into two main sections to provide a comprehensive understanding. The first section focuses on software-based solutions strategies that require no hardware modifications and operate primarily through software improvements or algorithms. The second section is devoted to hardware-based solutions that necessitate changes or enhancements to the hardware itself. This dichotomy allows us to present a balanced view of the field, demonstrating the strengths and weaknesses inherent in each approach.

Through this comprehensive exploration, we aim to provide readers with a thorough understanding of the existing landscape of solutions to resource contention in SMT processors. This understanding can be a foundation for developing new, innovative strategies to enhance SMT processors' performance and efficiency.

3.1 Software-based approaches

These strategies work at the operating system level or above and do not require hardware changes. They often involve scheduling algorithms that attempt to optimize the assignment of threads to cores based on various metrics and considerations.

The study by Bulpin and Pratt (BULPIN; PRATT, 2005) harnesses the power of performance counters to devise a symbiotic co-scheduling strategy for simultaneous multithreaded processors. While their work contributes to thread scheduling in multithreaded environments, it fails to consider the types of instructions executed by threads and how this diversity influences the usage of different functional units. In our proposed research, we go beyond this standard approach by implementing an instruction-aware mapping strategy that actively considers the instruction types of threads. This allows us to effectively address and mitigate functional unit contention, resulting in a more comprehensive solution for optimizing performance.

Fedorova et al. (FEDOROVA; SELTZER; SMITH, 2007) proposed an operating system scheduler to ensure performance isolation. In their proposal, threads running in parallel with similar cache miss rates get equal cache allocations. The shared cache is allocated based on demand, so if the threads have identical needs, they will have similar cache allocations.

The work by Tam et al. (TAM; AZIMI; STUMM, 2007b) presents a scheduling scheme that organizes threads based on online-detected data-sharing patterns gleaned through hardware performance counters. The methodology actively identifies data-sharing ways and categorizes threads accordingly. Consequently, the scheduler aims to map threads from the same cluster onto the same processor or processors in proximity to minimize remote cache accesses for shared data. However, this approach, while valuable, is tightly intertwined with memory access patterns and data sharing without considering the inherent variability in the types of instructions that threads execute and how these instructions stress different functional units. The proposed tool in our thesis advances this premise by focusing on mapping threads based on instruction patterns, thereby addressing functional unit contention, a critical aspect that the method by Tam et al. (TAM; AZIMI; STUMM, 2007b) overlooks. Our tool is intended to balance hardware utilization and mitigate functional unit contention optimally.

Jiang et al. (JIANG et al., 2008) propose a reuse distance based on a locality model that proactively predicts scheduled processes' performance. The prediction is used in runtime scheduling decisions. They employed the proposed locality model in designing cache-contention-aware proactive scheduling that assigns processes to the cores according to the predicted cache-contention sensitivities. However, the predictive model must be constructed for each application through an offline profiling and learning process.

Tian et al. (TIAN; JIANG; SHEN, 2009) propose an A*-search-based algorithm to accelerate searching for optimal schedules. They formulated optimal co-scheduling as a tree-search problem and developed an A*-based algorithm to find the optimal schedule. The authors reduced constraints on finding optimal scheduling by allowing threads of different lengths. Further, they designed and evaluated two approximation algorithms, namely A*-cluster and local-matching. The A*-cluster algorithm is a derivative of the A*-search-based algorithm that employs online adaptive clustering. It trades accuracy for scalability. On the other hand, the local-matching algorithm applies graph theory to find the best schedule at a given time without any provision for the upcoming schedules. Although optimal scheduling algorithms are costly and inefficient for practical purposes, they can provide insights to enhance the practical scheduling algorithms and associated complexities.

Feliu et al. (FELIU et al., 2012) first studied how cache hierarchy contention affects multicore architecture performance. Afterward, Feliu et al. (FELIU et al., 2016) present a bandwidth-aware scheduler for SMT multicores. They use Instructions Per Cycle (IPC) to estimate progress and bandwidth on LLC and main memory. Their scheduler improves the performance of SPEC CPU 2006 by up to 6.7% over the Linux scheduler. However, the authors do not evaluate parallel applications as the NAS benchmark.

Feliu et al. (FELIU et al., 2020) propose a technique that identifies SMT-adverse applications and schedules them in isolation on a dedicated core, reducing the resource contention on these specific applications. Other cores will execute three applications when it happens, which may cause additional contention. It simplifies the probabilistic model proposed by Eyerman et al. (EYERMAN; EECKHOUT, 2010), which is infeasible since it would require much more per-core hardware counters than the processors can deliver. Feliu et al. mitigate this using per-core Cycles Per Instruction (CPI) stacks available in International Business Machines Corporation (IBM) POWER processors. However, it limits its applicability since most AMD and Intel processors still need this feature. Also, they proposed strategies to schedule sequential applications, which differ significantly from those of scheduling multiple parallel applications.

Kundan et al. (KUNDAN; ANAGNOSTOPOULOS, 2021) introduce a priority-aware scheduling methodology optimized for single-threaded, profile-dependent application execution on chip multicore processors. This methodology, tested on a real platform and designed with no specific architecture dependence, demonstrates an online, operational mode to respond to the needs of running applications dynamically. It effectively improves the performance of up to 4 applications, employing a progress-aware scheduling mechanism that ensures resource availability without causing resource starvation for low-priority applications. The innovative approach leverages profiling information to manage and optimize application resource distribution.

Pi et al. (PI; ZHOU; XU, 2022) introduce Holmes, a single-threaded, profile-dependent approach operating in a user-space environment to diagnose SMT interference and dynamically adjust CPU scheduling to colocate jobs in multi-tenant systems efficiently. Deployed on a real platform, Holmes employs an online IAM scheduler that, independently of architecture specifics, uses hardware performance events to measure SMT interference on memory access and dynamically allocates CPU cores in response. While Holmes and IAM strive to mitigate interference, they target different types of interference and employ distinct mechanisms to achieve their goals.

Kundan et al. (KUNDAN et al., 2022) introduce a methodology that leverages a multithreaded application approach to enhance multiprocessor performance, primarily by curbing contention for shared resources such as LLC and main memory. This is achieved via the implementation of performance-aware, contention-minimizing scheduling policies. The study uses comprehensive, fine-grained application characterization methodologies, harnessing the power of HWPCs and Cache Monitoring Technology (CMT). These technologies aid in developing static and dynamic contention-aware scheduling policies. These policies, which are profile-dependent, base their operational efficacy on the prevailing pressure on the resources. The designed contention-aware policies are meant to function at the software level, negating architecture dependencies, and require real-time profile data obtained from the processors' HWPCs and CMT to make informed scheduling decisions. Importantly, Kundan et al.'s research employs real platforms for the execution of the study, eschewing simulated environments, thus bolstering the credibility and applicability of their results in practical, real-world scenarios. Additionally, the study emphasizes an online approach, making it possible to continually adapt and respond to changing application behavior and system conditions, further enhancing the system's overall performance.

Zhao et al. (ZHAO et al., 2023) introduce an innovative task-scheduling technique tailored for single-threaded applications on multi-core systems. This technique is fundamentally profile-dependent, utilizing empirical observations as a vital input to maximize system throughput. Operating at a higher abstraction level, it strategically assigns tasks to cores, guided by a predictive model derived from profiling data. This online methodology ensures real-time responsiveness and adaptability to dynamic workloads. While this approach has no specific architectural dependence, which underscores its broad applicability, it leverages cache efficiently for performance enhancement. The authors rigorously validated their system in a simulated environment and on a real-world computing platform, proving its practicality and effectiveness.

Shi et al. (SHI et al., 2023) propose Alioth, a machine learning-based performance monitoring system designed specifically for multithreaded, multi-tenancy applications in public cloud environments. The central problem addressed is colocation interference, where shared resources can lead to considerable application performance degradation. To address this, they generate a comprehensive dataset from extensive colocation experiments on real platforms and train a machine-learning model to predict performance degradation. This model, which is profile-dependent, uses denoising auto-encoders to recover lost data, a domain adaptation neural network for transfer learning, and a SHapley Additive exPlanations (SHAP) explainer to automate feature selection and enhance interpretability. This approach differs from the IAM as it leverages machine learning techniques instead of explicit algorithms and rules. The paper promotes an online, dynamic mapping strategy, consistently learning from incoming data to enhance predictions. Although the solution primarily relies on software, it does engage low-level hardware metrics for feature generation. Lastly, it is noteworthy that the system operates with no architecture dependence, making it versatile across various hardware configurations.

3.2 Hardware-based approaches

These techniques require specific hardware features or changes to the hardware itself. They involve novel processor architectures or leverage particular hardware features, such as new performance counters.

Snavely et al. (SNAVELY; TULLSEN, 2000) introduced a symbiotic scheduler called Sample, Optimize, Symbios (SOS) simultaneous multithreaded processor. It identifies the characteristics of threads that are scheduled through sampling. SOS runs in two distinct phases: the sample phase and the symbiosis phase. It gathers information about threads running together in different schedule permutations during the sample phase. After this sample phase, SOS picks the schedule that is predicted to be optimal and proceeds to run this schedule in the symbiosis phase. The performance metrics of a schedule are gathered through hardware counters. SOS employs many predictors to identify the best schedule. One interesting result provided by Snavely et al. is that IPC alone is not a good predictor. Threads with higher IPCs monopolize system resources which can be detrimental to threads with lower IPCs. The limitation of this work is that it tries many schedules during the sample phase to predict the best schedule to be executed in the symbiosis phase. The sample phase would be much longer for workloads of many threads exceeding the available hardware resources. In such a scenario, threads can change their characteristics not reflected during the symbiosis phase. Therefore, the symbiosis phase would need to be more accurate due to the change in execution characteristics of threads during the sample phase. A limited number of samples can be used to avoid a more extended sample phase; however, the probability of missing better schedules is increased in this case.

Settle et al. (SETTLE et al., 2004) developed a memory monitoring tool providing statistics in simultaneous multithreaded processors. The authors used the statistics regarding threads memory accesses to build a scheduler that minimizes capacity and conflict misses. L2 cache accesses are monitored for each thread to generate per-thread cache activity vectors. These vectors indicate the sets that are accessed most of the time. The intersection of these vectors specifies the sets likely to be conflicting. This information is then used in scheduling decisions.

Cazorla et al. (CAZORLA et al., 2004) introduced a dynamic resource control mechanism that uses pending L1 data misses as a classification method. The tool monitors the usage of resources by each thread and tries to allocate resources, avoiding monopolization somewhat. It classifies threads based on cache access patterns as fast and slow. Then, it allocates the resources to these groups accordingly. They simulated it for multiple single-thread applications and showed a possible hardware implementation of their policy.

El-Moursy et al.'s work (EL-MOURSY et al., 2006) is an insightful contribution to the discourse on SMT performance optimization. The authors propose a sophisticated scheduling algorithm where threads are dispatched to processors contingent on the count of ready instructions. This approach attempts to distribute threads undergoing incompatible phases across different processors, a strategy aimed at maximizing hardware utilization and mitigating performance degradation due to resource contention. Their methodology incorporates hardware performance counters to garner requisite data for evaluating the compatibility of different thread phases. Hardware performance counters provide low-level insights into processor activities, capturing data points like the number of executed instructions, cache misses, branch mispredictions, and many other performance-related statistics. Despite its innovation, the approach by El-Moursy et al. (EL-MOURSY et al., 2006) also presents limitations that our proposed work addresses. First, it assumes an intimate knowledge of thread phases, which might not always be accessible, especially in dynamic, real-time applications. It requires constant monitoring and profiling of the running threads, which can introduce considerable overhead and potentially offset any performance gains achieved. Furthermore, their approach needs to explicitly consider the instruction types executed by each thread and their impact on different functional units, which is a focal point of our research.

Cruz et al. (CRUZ et al., 2016) propose an extension of the memory management unit to improve memory accesses' locality. The authors analyzed the memory access behavior in hardware, providing information to the operating system to perform an online mapping.

Akturk et al. (AKTURK; OZTURK, 2019) propose a cache-hierarchy-aware scheduler for multiple sequential applications, which balances the number of accesses to the L1 cache, reducing the number of evictions on shared caches which eventually limits the performance. Akturk et al. work are very similar to Settle et al. (SETTLE et al., 2004); the main differences are that Akturk considers only L1 cache while Settle correlated different cache levels and metrics with the IPC of several applications.

Aceituno et al. (ACEITUNO et al., 2021) introduces a task model that accounts for the interference a task can create on other tasks executing on distinct cores due to memory contention. This model, rooted in single-threaded operations, calculates interference independently of specific profiling data, exhibiting a profile-independent nature. Utilizing a simulation environment, the study circumvents the constraints of hardware architecture dependence, providing broader applicability. Their proposed online scheduling algorithm dynamically adapts to system states, providing real-time task-core assignment solutions. The research further delves into and compares various partitioning algorithms, suggesting three strategies for practical task-core assignments. The primary objective is to maximize the count of scheduled tasks and minimize total interference. Compared to the IAM tool, Aceituno et al. (ACEITUNO et al., 2021) focus on mitigating memory contention. At the same time, IAM strategically targets functional unit contention, emphasizing instruction-aware management.

Chen et al. (CHEN; TSAY, 2021) present an online process scheduling algorithm optimized for heterogeneous multi-core systems characterized by large cores and energy-efficient small cores. As a critical differentiator, this work focuses on single-threaded, profile-independent applications, representing a common yet challenging workload type. The authors identify the limitations of previous heuristic-based algorithms, which primarily aimed to schedule high scaling factor processes on large cores. These approaches often need to pay more attention to the value of assigning long-running processes to such cores, a gap the proposed algorithm seeks to fill. This methodology leads to more efficient use of system resources, transcending the constraints of specific architectural dependencies. This approach was evaluated using single-threaded applications from SPEC 2006, and the testing was performed in a simulated environment, making it both versatile and broadly applicable in various contexts.

Diavastos et al. (DIAVASTOS; CARLSON, 2022) presents a processor reliant on architecture that aims for heightened efficiency and precision in instruction scheduling in single-threaded applications. This approach is profile-independent, harnessing real-time load delay tracking rather than relying on prior profiling or predetermined access latencies. The system dynamically adapts to ongoing operations, learning from recurring memory access delays to accurately predict instruction issue times. The results presented in this paper are derived from a simulation model, offering a theoretical understanding of the system's efficacy and scalability.

3.3 Summary of Related Work

Examining the related work as outlined in Table 3.1, we observe a broad landscape of methodologies to optimize thread scheduling. Our proposed tool sets itself apart in this diverse field through several distinguishing characteristics.

Table 3.1 – Comparison of IAM tool with related work.

Proposal	Multithreaded application	Real platform	No architecture dependence	Online
(SNAVELY; TULLSEN, 2000)	✓			✓
(SETTLE et al., 2004)			✓	✓
(CAZORLA et al., 2004)			✓	✓
(BULPIN; PRATT, 2005)		✓		✓
(EL-MOURSY et al., 2006)			✓	✓
(FEDOROVA; SELTZER; SMITH, 2007)	✓	✓		✓
(TAM; AZIMI; STUMM, 2007b)	✓	✓		✓
(JIANG et al., 2008)		✓	✓	
(TIAN; JIANG; SHEN, 2009)	✓	✓	✓	✓
(EYERMAN; EECKHOUT, 2010)	✓		✓	✓
(FELIU et al., 2012; FELIU et al., 2016)		✓	✓	✓
(CRUZ et al., 2016)	✓		✓	✓
(AKTURK; OZTURK, 2019)			✓	✓
(FELIU et al., 2020)		✓		✓
(KUNDAN; ANAGNOSTOPOULOS, 2021)		✓	✓	✓
(ACEITUNO et al., 2021)			✓	✓
(CHEN; TSAY, 2021)			✓	✓
(PI; ZHOU; XU, 2022)		✓	✓	✓
(KUNDAN et al., 2022)	✓	✓	✓	✓
(DIAVASTOS; CARLSON, 2022)				✓
(ZHAO et al., 2023)		✓	✓	✓
(SHI et al., 2023)	✓	✓	✓	✓
Instruction-Aware	✓	✓	✓	✓

One category of existing works, represented by Settle et al. (SETTLE et al., 2004), Cazorla et al. (CAZORLA et al., 2004), and El-Moursy et al. (EL-MOURSY et al., 2006), centers on co-scheduling strategies for multiple sequential applications. The common drawback here is the necessity for hardware modifications, often impractical or costly. Our tool, however, avoids this limitation, offering a software-driven approach that does not require such changes.

Akturk et al. (AKTURK; OZTURK, 2019) put forth methods for calculating inter-thread contention using specific metrics unavailable in most contemporary processors. The primary focus of their mechanisms lies on inter-thread cache contention, serving memory-bound applications effectively. Nevertheless, the same approach can yield sub-optimal outcomes when faced with applications involving heavy usage of floating-point and integer units. Our methodology addresses this shortfall by considering a more comprehensive array of factors in the scheduling process.

Work by Tam et al. (TAM; AZIMI; STUMM, 2007b) and Cruz et al. (CRUZ et al., 2016) delved into the realm of parallel applications. While valuable, these solutions present limitations. Cruz et al.'s method calls for hardware modifications for its implementation. Tam et al.'s solution is architecture-specific, potentially hampering its performance across different architectures. In contrast, our tool demonstrates its versatility as a hardware-agnostic solution, requiring only a single hardware counter, thus enhancing its broad adaptability.

Feliu et al.'s work (FELIU et al., 2020) utilized a simplified version of Eyerman et al. (EYERMAN; EECKHOUT, 2010)'s probabilistic model. Unfortunately, requiring more per-core hardware counters than processors can supply makes this approach unsuitable in practical settings. Furthermore, their mitigation strategy's reliance on per-core CPI stacks, exclusive to IBM POWER processors, curtails its applicability to other popular architectures like AMD and Intel.

Kundan et al. (KUNDAN; ANAGNOSTOPOULOS, 2021) proposed a priority-based scheduling strategy. Although insightful, it may overlook certain resource contention types. Aceituno et al. (ACEITUNO et al., 2021) offered a hardware-implemented solution, yielding effective results but requiring hardware modifications that could be prohibitive.

Chen et al. (CHEN; TSAY, 2021) developed a performance prediction model that, while robust, demands an exhaustive offline profiling stage and may struggle with generalizing to new workloads. Pi et al. (PI; ZHOU; XU, 2022) used reinforcement learning for thread scheduling in a different approach. Although promising, the method's considerable training time, computational overhead, and potential sensitivity to changing workloads pose challenges.

Several recent solutions, including those by Kundan et al., (KUNDAN et al., 2022), Diavastos et al. (DIAVASTOS; CARLSON, 2022), and Zhao et al. (ZHAO et al., 2023), deploy hardware features to estimate resource contention. Despite their potential, these methods presuppose the availability of specific hardware counters that may not be universally present across processor types.

The work of Shi et al. (SHI et al., 2023) ventured into AI, employing machine learning techniques to enhance scheduling decisions. The promise of Artificial Intelligence (AI) in this context is considerable. However, the requirement for large-scale training data and the model's black-box nature, which might complicate troubleshooting, can present practical limitations.

While many existing proposals predominantly focus on single-thread multiprogram workloads (AKTURK; OZTURK, 2019; CAZORLA et al., 2004; EL-MOURSY et al., 2006; FELIU et al., 2020; JIANG et al., 2008; SETTLE et al., 2004; KUNDAN; ANAGNOSTOPOULOS, 2021; ACEITUNO et al., 2021; CHEN; TSAY, 2021; PI; ZHOU; XU, 2022; KUNDAN et al., 2022; DIAVASTOS; CARLSON, 2022; ZHAO et al., 2023), our research concentrates on parallel applications, a category more relevant in the context of cloud providers and data centers.

Numerous proposed mechanisms lean heavily on memory information or call for hardware modifications (AKTURK; OZTURK, 2019; CAZORLA et al., 2004; SETTLE et al., 2004; CRUZ et al., 2016; EL-MOURSY et al., 2006; ACEITUNO et al., 2021; KUNDAN et al., 2022; DIAVASTOS; CARLSON, 2022; ZHAO et al., 2023). Such requirements may not always be practical or feasible. Our tool offers a comprehensive strategy to mitigate performance degradation from shared functional units when running parallel applications.

Several solutions are reliant on specific architectures (FELIU et al., 2020; TAM; AZIMI; STUMM, 2007b; ACEITUNO et al., 2021; KUNDAN et al., 2022; DIAVASTOS; CARLSON, 2022). In sharp contrast, our instruction-aware tool collects data from hardware counters commonly available across a multitude of architectures, thereby maximizing its versatility and real-world applicability.

4 METHODOLOGY FOR THREAD MAPPING IN SMT PROCESSORS

This chapter delves into the performance implications of sharing diverse resources on SMT processors. The overarching objective is to elucidate how resource sharing and contention can significantly impact overall performance and to present compelling reasons for strategic thread mapping to mitigate such issues.

Following the analysis of functional unit contention, we focus on a specialized microbenchmark we developed. This microbenchmark is specifically designed to stress distinct processor functional units. By using this tool, we can demonstrate that the type of operations performed by each thread has a profound influence on both performance and contention levels. It offers insight into functional unit utilization and contention dynamics in an SMT context.

Subsequently, we present compelling evidence to illustrate that mapping threads which utilize the same resources on a single core can lead to significant contention. Such contention can effectively diminish performance, potentially negating the benefits offered by SMT. However, we further argue that intelligent mapping strategies can mitigate these performance issues.

This chapter underlines the importance of resource-conscious thread mapping on SMT processors by revealing the intrinsic connection between the type of operations executed by threads, resource sharing, and the resulting contention. Furthermore, it demonstrates the potential performance gains that can be achieved by effectively managing functional unit contention.

4.1 Experimental Design

This section elucidates the experimental methodology employed to gauge the performance impact of functional unit contention and the effectiveness of our proposed mitigation tool.

For the workloads, we utilized SMT-Bench, the Open Multi-Processing (OpenMP) implementation of the NPB version 3.4 (BAILEY, 2011) and the SPEC OpenMP 2012 benchmark (MÜLLER et al., 2012). These workloads offer diverse computational profiles, allowing us to explore the performance implications under various scenarios.

In addition to the performance metrics derived from the workloads, we delved deeper into processor behavior by gathering data from hardware counters. This was achieved using PAPI (TERPSTRA et al., 2010), a powerful tool that provides access to many processor hardware counters.

The experimental evaluations were performed on two distinct machines. Firstly, the `chiclet` machine, a part of the Grid'5000 cluster, was utilized. Each node of this machine is outfitted with two AMD EPYC 7301 processors. Each processor houses 16 physical cores, allowing running 32 threads concurrently when employing simultaneous multithreading (TULLSEN; EGGERS; LEVY, 1995). This machine operates under the Linux kernel, version 4.19. Secondly, the `phoenix` machine from the GPPD cluster was also employed for experiments. Each node of this machine is equipped with 2 x Intel Xeon Gold 5317, which provides 12 cores per processor. This translates to a potential of executing 48 threads simultaneously when utilizing Hyper-threading technology. This machine operates under the Linux kernel, version 5.10. More detailed specifications are included in Table4.1.

Table 4.1 – Execution Environment

Parameter	AMD	Intel
Microarchitecture	Zen	Ice Lake
Processor	2 × AMD EPYC 7301	2 × Intel Xeon Gold 5317
	2 × 16 cores	2 × 12 cores
	2-SMT cores, 64 threads	2-SMT cores, 48 threads
Caches/processor	16 × 32 KByte L1	12 × 48 KByte L1
	16 × 512 KByte L2	12 × 1280 KByte L2
	64 MByte L3	18 MByte L3
Memory	128 GByte DDR4	128 GByte DDR4
Environment	Linux 4.19	Linux 5.10
	GNU C Compiler 9.1.0	GNU C Compiler 10.2.1

We always execute two applications for the experiments, each with one thread per core. As the applications have different execution times, we use as a metric the weighted speedup calculated as in Equation 4.1, where N is the number of applications and baseline is the Linux scheduler. We also use the restart when the policy is finished by restarting the fastest application until the slowest one has finished. We restrict ourselves to two applications since the processor is 2-SMT. However, our tool is generic and could be evaluated with many applications and threads per core.

$$\sum_{i=1}^{N} \frac{ExecutionTime_{i,Baseline}}{ExecutionTime_{i,Comparison}} \tag{4.1}$$

The experimental results were obtained through the geometric mean of 30 random executions with exclusive access to the machine. The results compare the weighted speedup obtained with our tool to the ones obtained with the Linux scheduler and a Round-robin mapping. The baseline for the weighted speedup calculation is the Linux scheduler, which will always have a speedup of 2, despite the number of running applications by two.

We compare the performance of our proposed tool with three existing thread mapping methodologies: the Linux Completely Fair Scheduler (CFS), a Round-Robin (RR) mapping technique, and a random mapping strategy. These established methods, tested on AMD and Intel processors, provide a robust comparison platform owing to their distinct approaches to managing threads and CPU resources.

The CFS is the default scheduling policy used by the Linux kernel (PABLA, 2009). The underlying principle of CFS is the equitable distribution of CPU resources among all active threads. This means that a solitary process running on a system will have full access to the processor's computational capacity, thus utilizing 100% of the available processing power. Conversely, if two processes run simultaneously, CFS ensures that each process receives precisely half, or 50%, of the processor's physical power. While this strategy ensures fairness among the tasks being executed on the system, it does not consider other influential factors like the distinct instruction patterns of different workloads. In contrast, our proposed tool incorporates this crucial workload characteristic into scheduling decisions.

The RR mapping, on the other hand, adopts a straightforward approach to distribute threads across cores in a circular order starting from the 0th core to the (number of virtual cores - 1). Notably, in a scenario with two applications, the RR mapping strategy places threads belonging to the same application onto the same core. This characteristic of RR mapping presents a stark contrast to our proposed tool, which aims to minimize contention by strategically distributing threads that stress different functional units across cores.

Random mapping presents an unstructured approach to thread placement, where threads are randomly assigned to cores. This strategy allocates threads to any available core without considering specific workloads or hardware characteristics. While this approach may sometimes result in efficient allocation due to pure chance, it generally lacks consistency and predictability in performance. There is no guarantee that threads executing similar instructions will not contend for the same core resources, which can lead to severe performance degradation.

4.1.1 SMT-Bench

The increasingly popular SMT processors have sparked a new wave of interest in their unique ability to share functional units and caches between threads executing on the same core. Although this shared utility remains concealed from software-level applications, the underlying hardware resources are limited. When multiple threads vie for the same functional unit, the processor's hardware scheduler is tasked with deciding which thread's instructions should be issued and which should be held back. To unravel and comprehend the intricacies and implications of resource sharing on SMT processors, we have introduced *SMT-Bench*, a meticulously assembled kernel designed to exert specific pressure on different hardware units. The complete source code and the details of SMT-Bench can be accessed at <https://gitlab.com/msserpa/SMT-bench>.

SMT-Bench harnesses the power of the PAPI (JOHNSON et al., 2012; TERPSTRA et al., 2010; WEAVER et al., 2012). PAPI is an instrumental tool that facilitates access to an array of processor hardware counters, which are invaluable for detailed performance analysis. Some of these kernels draw inspiration from the microbenchmarks proposed by Alves et al. (ALVES et al., 2015).

SMT-Bench is designed to operate with both software and hardware-implemented mapping policies. These policies aim to alleviate the potential negative impact of resource sharing, thereby boosting application-level and overall system performance. Achieving these performance gains requires a critical understanding of the effects of executing threads comprising similar instructions under various conditions. However, garnering this insight from existing benchmarks or real-world applications poses a formidable challenge due to their diverse instruction issues, which exert variable pressure on different functional units during runtime. This is where our approach, which employs precise measurements using bespoke microbenchmarks, shines.

SMT-Bench offers a unique solution to the intrinsic complexities of gauging the effects of threads issuing similar instructions under various conditions. It provides a platform to scrutinize these instruction patterns in isolation, allowing us to discern their performance implications under different mapping strategies. SMT-Bench, therefore, offers an unprecedented insight into the impact of varying thread behaviors on hardware resource contention and overall performance.

As shown in Figure 4.1, SMT-Bench comprises eight kernels, each designed to represent a different functional unit in an SMT processor. The y-axis in the figure illus-

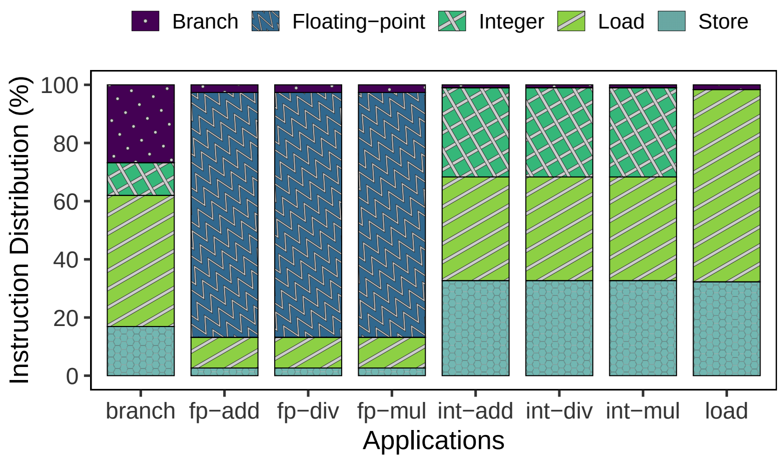


Figure 4.1 – Instruction distribution for different applications.

trates the instruction distribution of each kernel. At the same time, the x-axis denotes the kernels themselves. The stacked bars within the figure visually depict the distribution of different instruction types.

The kernels include `int-add, int-div, and int-mul`, which are tailored to stress the integer units. For both floating-point and integer kernels, the iterations loop was unrolled four times to enhance the stress levels. The `load` kernel implements a loop that traverses a linked list, waiting for each load to complete before starting the next, thereby stressing both the load and store units.

The kernels are designed to simulate various application conditions, ranging from compute-intensive tasks to memory-bound workloads. Therefore, they enable a comprehensive analysis of SMT processor performance under multiple types of applications. Furthermore, they offer a benchmark for analyzing the effectiveness of tools like IAM for mitigating the contention of functional units.

For instance, the `branch` kernel combines if-else, and switches construct to generate branch behavior. Figure 4.1 shows that it issues a balanced mix of instructions, with a large proportion of branch instructions (26.8%) and load instructions (45.1%), along with a substantial number of store instructions (16.9%) and integer instructions (11.3%). Notably, this kernel does not involve any floating-point operations.

In contrast, the `fp-add`, `fp-div`, and `fp-mul` kernels heavily stress the floating-point units by executing 32 independent operations inside a loop. These kernels issue minimal memory operations, control hazards, and data dependencies, achieving a throughput that approaches the ideal scenario. Each of these kernels contains a high proportion of floating-point instructions (84.2%) and a minor fraction of branch (2.6%), load (10.5%), and store instructions (2.6%). Importantly, they do not execute any integer operations.

The `int-div`, `int-add` and `int-mul` applications, on the other hand, focus primarily on integer operations. They comprise a small count of branch instructions (1.0% of total instructions) but a substantial proportion of load (35.6%), store (32.7%), and integer instructions (30.7%). These applications do not involve any floating-point instructions.

Lastly, the `load` application is highly memory-bound, with a significant percentage of load instructions (66.1%) and store instructions (32.3%). It has a lower rate of branch instructions (1.6%) and does not execute integer or floating-point instructions.

With this diverse suite of kernels, SMT-Bench provides a comprehensive benchmark for evaluating the performance of SMT processors under different workloads. This includes compute-intensive and memory-bound tasks, offering a broad perspective on performance characteristics. It allows us to analyze and understand the effectiveness of different strategies, such as the IAM tool, in mitigating contention for functional units in SMT processors.

4.1.2 NAS Parallel Benchmarks

The NPB is a collection of programs designed to help evaluate the performance of parallel supercomputers. They are derived from Computational Fluid Dynamics (CFD) applications and consist of five "kernel" benchmarks and three "pseudo-application" benchmarks. Here, we provide a detailed breakdown of each benchmark, which we used to stress test our solution for mitigating functional unit contention. Please refer to Table 4.2 for a comprehensive overview of the NPB, including their names, acronyms, and the language in which they were written.

- **CG (Conjugate Gradient)**: The Conjugate Gradient benchmark uses a conjugate gradient method to approximate the smallest eigenvalue of a defined, large and sparse symmetric matrix. This benchmark is an excellent test for irregular memory access patterns and long-distance communication between threads, especially those employing matrix multiplication by a structureless vector. It reflects real-world scenarios where data is sparse and irregularly distributed.
- **EP (Embarrassingly Parallel)**: The Embarrassingly Parallel benchmark tackles the problem of generating pairs of Gaussian distributions and tabulating their values in successive annuli squares. It is unique among the NPB benchmarks, as it

has virtually no inter-thread communication, focusing exclusively on stressing the performance of floating-point operations. EP presents the perfect conditions for benchmarking parallel computation without communication overhead.

- **FT (Fourier Transform)**: The Fourier Transform benchmark solves the problem of three-dimensional Fourier transformation in a parallel manner using the Fast Fourier Transform (FFT) algorithm. This benchmark rigorously tests high-distance communication, mainly exhibiting an all-to-all core communication pattern. This is crucial in evaluating the performance and efficiency of parallel algorithms that involve extensive data exchange, such as FFT.
- **IS (Integer Sort)**: The Integer Sort benchmark leverages the Bucket Sort algorithm to sort a vector of integer numbers. It poses random memory access challenges and tests integer operations performance and communication. Thus, it reflects the demands of algorithms that require heavy integer computation and data exchange.
- **MG (Multigrid)**: The Multigrid benchmark solves a simplified multigrid calculation problem, testing short- and long-distance communications. MG requires highly structured long-distance communication, making it valuable in understanding the performance of parallel algorithms with hierarchical or grid-like data structures.
- **UA (Unstructured Adaptive Mesh)**: The Unstructured Adaptive Mesh benchmark is designed to solve a problem of heat transfer in a cubic domain, discretized using an unstructured adaptive mesh. This benchmark showcases the adaptability and performance of parallel computing resources in handling unstructured and dynamically changing computational tasks.
- **BT (Block Tridiagonal)**: The benchmark solves a synthetic CFD problem with multiple 5x5 tridiagonal block equation systems with non-dominant diagonals. It represents a more constrained parallelism scenario than other CFD applications (like LU and SP), which makes it beneficial in testing the system's performance under limited parallelism conditions.
- **LU (Lower Upper Gauss-Seidel Solver)**: The Lower Upper Gauss-Seidel Solver benchmark solves a synthetic CFD problem using a sparse triangular linear equations system with 5x5 blocks. LU involves global data dependencies and involves several short communications. The LU benchmark is a good proxy for real-world CFD applications that involve intricate data dependencies and demand efficient communication.

- **SP (Scalar Pentadiagonal)**: The Scalar Pentadiagonal benchmark solves systems of pentadiagonal scalar equations with non-dominant diagonals. Like LU, SP involves global data dependencies but contrasts with sporadic and more prolonged communications. It tests the performance of parallel computing systems in managing complex data dependencies and varied communication patterns.

Table 4.2 – Overview of the OpenMP version of the NPB.

Acronym	Name	Language
BT	Block Tri-diagonal solver	Fortran
CG	Conjugate Gradient	Fortran
EP	Embarrassingly Parallel	Fortran
FT	Discrete 3D fast Fourier Transform	Fortran
IS	Integer Sort	C
LU	Lower-Upper Gauss-Seidel solver	Fortran
MG	Multigrid on a sequence of meshes	Fortran
SP	Scalar Penta-diagonal solver	Fortran
UA	Unstructured Adaptive mesh	Fortran

4.1.3 SPEC OMP 2012 Benchmark

The Standard Performance Evaluation Corporation's (SPEC) OMP2012 suite is an industry-standard collection of benchmarks designed to assess the performance of parallel computing systems using OpenMP, an API supporting multi-platform shared-memory parallel programming in C, C++, and Fortran. Here, we provide a detailed breakdown of each benchmark, which we used to evaluate our solution for mitigating functional unit contention. For further details on the SPEC OMP®2012 benchmarks, including their identifiers, names, and the programming language used, please consult Table 4.3.

- **350.md (Molecular Dynamics)**: This benchmark simulates molecular dynamics, which model particles' complex interactions and movements under various physical forces. It is a comprehensive test of a system's ability to handle computationally intensive floating-point operations and manage memory access efficiently. By mimicking the realistic dynamics of molecular systems, this benchmark offers a solid evaluation platform for modern systems focusing on scientific and engineering applications.
- **351.bwaves (Blast Wave Simulation)**: This benchmark models the propagation of blast waves in a turbulent medium. It not only stresses the floating-point computa-

tional performance of the processor but also presents a considerable challenge to the cache and memory subsystems. The accuracy and speed of such simulations can be critical in various research and industry applications, including defense, aerospace, and energy production.

- **352.nab (Nucleic Acid Builder)**: This benchmark employs the Nucleic Acid Builder (NAB) to emulate the behavior of complex molecules. It provides valuable insights into the system's performance when running sophisticated molecular dynamics applications, which are fundamental in pharmaceuticals, bioinformatics, and materials science.
- **357.bt331 (Block Tri-diagonal solver)**: Derived from the NAS Parallel Benchmarks, this benchmark resolves fluid dynamics problems using block tridiagonal systems. It comprehensively tests a system's ability to perform floating-point operations and manage memory access effectively. It demonstrates its potential performance in simulations crucial for aeronautical, automotive, and climate research.
- **358.botsalgn (Multiple Sequence Alignment)**: Part of the Barcelona OpenMP Tasks Suite (BOTS), this benchmark features a collection of kernels and applications emphasizing different aspects of task parallelism. Specifically, it includes an alignment kernel for multiple sequence alignment, an important problem in computational biology.
- **359.botsspar (Sparse Matrix Operation)**: Also part of the BOTS, this benchmark focuses on sparse matrix operations, thereby testing the system's performance when dealing with irregular memory access patterns. Sparse matrix computations are commonplace in various scientific, engineering, and information retrieval applications.
- **360.ilbdc (Incompressible Lattice-Boltzmann Method)**: This benchmark simulates an incompressible fluid flow around a cylinder using the lattice-Boltzmann method. It puts the processor's floating-point computation capabilities to the test, along with the efficiency of memory subsystems, demonstrating the system's ability to run computationally intensive physical simulations.
- **362.fma3d (3D Structure Response Simulation)**: This benchmark simulates the response of a 3D structure subject to a dynamic load, providing a rigorous assessment of the system's floating-point operations and memory access capabilities. Such simulations are integral to various areas, including civil engineering, mechanical design, and seismic research.

- **363.swim (Shallow-Water Equations Solver)**: This benchmark, part of the NAS Parallel Benchmarks suite, solves a system of shallow-water equations, stressing the system's floating-point computational capabilities. The performance on this benchmark can indicate the system's ability to handle computational fluid dynamics simulations, which is crucial for meteorological modeling and environmental studies.
- **367.imagick (Image Manipulation)**: This benchmark utilizes the ImageMagick software suite to create, edit, and compose bitmap images. It assesses the system's ability to handle memory-intensive operations, testing its overall performance and efficiency in image-processing tasks. This is a proxy for graphic-intensive applications, including digital media processing, machine learning, and computer vision.
- **370.mgrid331 (Multigrid)**: Derived from the NAS Parallel Benchmarks, this benchmark tackles elliptic partial differential equations using a multigrid method. It presents an excellent platform to test the system's capability for floating-point computation and efficient memory access, which is fundamental for mathematical physics, climate modeling, and image processing applications.
- **371.applu331 (Lower Upper Gauss-Seidel Solver)**: This benchmark, part of the NAS Parallel Benchmarks suite, solves partial differential equations using a factorization method. It provides an exhaustive evaluation of the system's capability to perform floating-point operations and handle memory access efficiently. Its performance is crucial for numerical simulations in fluid dynamics, heat transfer, and electromagnetics.
- **372.smithwa (Smith-Waterman Algorithm)**: This benchmark models the Smith-Waterman algorithm, a widely used method for identifying similar regions between two nucleotide or protein sequences. It challenges the system's performance in integer operations and irregular memory access, demonstrating its potential in bioinformatics applications, such as sequence alignment and genomic data analysis.
- **376.kdtree (Nearest-Neighbor Search using KD-Trees)**: This benchmark implements a nearest-neighbor search using KD-Trees, a data structure for organizing points in a k-dimensional space. It is a valuable test for the system's ability to handle memory-intensive operations and efficient integer operations, which are paramount for computer graphics, machine learning, and spatial database systems applications.

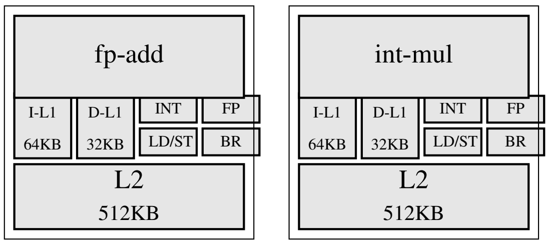
Our study did not include the SPEC benchmarks md, bt331, mgrid331, and applu331. We faced issues in successfully compiling md, which rendered it unusable for this analysis. As for bt331, mgrid331, and applu331, they were excluded because they are the same as the NPB benchmarks BT, MG, and LU, respectively, which we have already analyzed. Our goal was to evaluate a diverse set of benchmarks, so including these would have introduced unnecessary redundancy into our analysis. Additionally, we decided to exclude ilbdc from our analysis. This decision was made due to the excessively long running times observed for ilbdc, even when using small input sizes. The extended execution duration of this benchmark was significantly longer compared to all the other benchmarks used in our study, and it would have disproportionately skewed our performance results and analysis. We intended to perform a balanced and comparative study across various benchmarks, which made excluding ilbdc necessary.

Table 4.3 – Overview of the OpenMP version of the SPEC OMP 2012 Benchmarks.

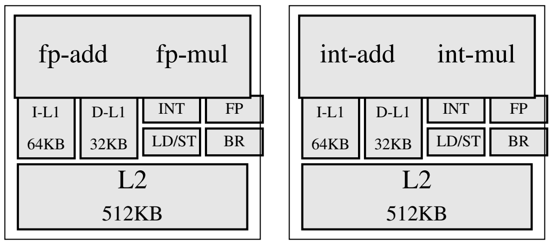
Acronym	Name	Language
350.md	Molecular Dynamics	Fortran
351.bwaves	Blast Wave Simulation	Fortran
352.nab	Nucleic Acid Builder	C
357.bt331	Fluid Dynamics Solver	Fortran
358.botsalgn	Multiple Sequence Alignment (BOTS)	C
359.botsspar	Sparse Matrix Operation (BOTS)	C
360.ilbdc	Incompressible Lattice-Boltzmann Method	C
362.fma3d	3D Structure Response Simulation	Fortran
363.swim	Shallow-Water Equations Solver	Fortran
367.imagick	Image Manipulation	C
370.mgrid331	Elliptic PDE Solver	Fortran
371.applu331	PDE Solver using ADI	Fortran
372.smithwa	Smith-Waterman Algorithm	C
376.kdtree	Nearest-Neighbor Search using KD-Trees	C

4.2 Performance Degradation Analysis

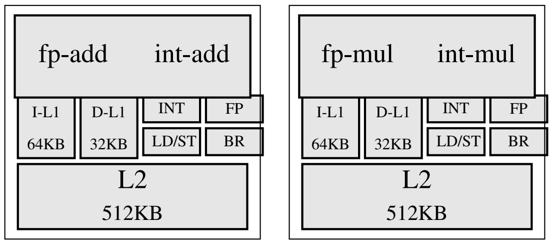
We leverage the SMT-Bench through diverse mapping scenarios to comprehensively understand the performance degradation. The contention of different functional units and their significant influence on application performance can be better illustrated with the three potential mapping scenarios on SMT-based processors like the AMD Zen architecture as represented in Figure 4.2:



(a) Single thread.



(b) Two threads are competing for the same functional units.



(c) Two threads using different functional units.
Figure 4.2 – Different co-running scenarios analyzed.

Scenario A (Isolated Execution): As depicted in Figure 4.2a, in this scenario, a single thread is allocated to run on each core. This case eliminates the potential for interference from co-running threads, rendering all functional units in a core fully accessible for the lone thread. Despite its advantages in unrestricted access to resources, this scenario always operates with half the number of threads compared to the other two scenarios, thus significantly reducing the total system throughput.

Scenario B (Identical Instruction Type Execution): In contrast to Scenario A, two threads are scheduled to run on the same core in Scenario B, as shown in Figure 4.2b. Both threads execute the exact instructions, stressing the same functional units. This leads to direct contention for the same resources, which could influence the performance of both threads.

Scenario C (Diverse Instruction Type Execution): Similar to Scenario B, Scenario C allocates two threads on the same core. However, unlike Scenario B, each thread in Scenario C executes different types of instructions, implying that they stress other functional units as displayed in Figure 4.2c.

In understanding the effects of resource contention on different functional units when SMT is enabled, as discussed previously in the introduction, although Scenario B always lags behind Scenario A, it accomplishes double the computation. Interestingly, despite Scenario C running twice the number of threads as Scenario A, the execution time remains strikingly similar. Ultimately, Scenario C emerged as the most optimal scenario, where different applications stressing different units are allocated on the same core, effectively boosting throughput while maintaining a comparable execution time to a solitary thread executing per core.

Therefore, our research and analysis suggest that an optimal thread to core mapping strategy should be as close as possible to Scenario C. Such a strategy can substantially mitigate functional unit contention, ultimately improving the performance of computing applications.

4.3 Final Remarks

This chapter has comprehensively examined application performance degradation factors and introduced a microbenchmark tool for evaluating different thread-to-core mapping scenarios in the context of SMT-based processors. The importance of understanding the role of functional unit contention and its impact on performance has been thoroughly discussed, emphasizing the critical need to devise optimal mapping strategies to minimize functional unit contention.

The evaluation and analysis demonstrated that performance degradation is significantly affected by the type of operations performed by threads on a core. For instance, the performance degradation was up to 120% when the same kind of instructions was executed in parallel by multiple threads on the same core (Scenario B). In contrast, negligible degradation occurred when different instructions were run in parallel on the same core (Scenario C). The single-threaded per-core execution (Scenario A) presented ideal per-thread performance but at the cost of severely reduced total system throughput.

Scenario C, characterized by the parallel execution of different types of instructions on the same core, was the most efficient thread-to-core mapping strategy. It effectively utilized the entire computational power on offer, demonstrating a near-zero per-thread performance loss while running twice the number of threads compared to Scenario A.

Based on these observations, it is evident that exploiting parallelism through efficient thread-to-core mapping is paramount in SMT-based processors or any multithreaded cores. Therefore, we advocate for an approach similar to Scenario C to mitigate functional unit contention and improve performance.

The insights gleaned from this chapter will significantly contribute to the following stages of this research, which include the development of dynamic mapping strategies to adapt to workload changes during runtime and the designing performance prediction models for thread-to-core mapping. We aim to continue striving towards improved performance, and this study constitutes a substantial step in that direction.

5 IAM - INSTRUCTION-AWARE MAPPING

5.1 Proposed Mechanism

Our proposed tool for online mapping operates in five steps, as depicted in Figure 5.1. The process is initiated with the application's execution. As the application starts running, the tool begins by gathering information about the machine's current topology. This includes the number of available cores, their arrangement, and the level of SMT each core supports, among other architectural details.

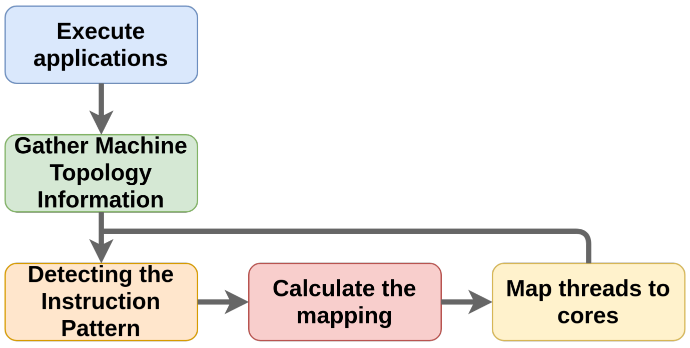

Execute applications
Gather Machine Topology Information
Detecting the Instruction Pattern
Calculate the mapping
Map threads to cores

Figure 5.1 – Steps used to perform the online mapping.

In the third step, the tool leverages the hardware counters provided by the PAPI to detect the instruction patterns of the applications. This detection process enables the tool to discern the types of instructions each thread is issuing, which is crucial in optimizing thread-to-core mapping.

Following the instruction pattern detection, the tool calculates the optimal thread mapping. This calculation considers the machine's topology and the previously detected instruction patterns. The goal here is to minimize functional unit contention by intelligently assigning threads to cores in a manner that accounts for the specific types of instructions that the threads are executing.

As the application continues its execution, the tool implements the calculated thread-to-core mapping. This mapping step ensures that each thread runs on the core that has been determined to be optimal for its specific instruction type.

From the third step onwards, this entire process is repeated online, allowing the tool to adapt to changes in the application's behavior or the system's state. The loop continues until the application's execution completes.

5.2 Implementing the tool

The implementation of our tool is enabled by preloading its library along with the application binary using the `LD_PRELOAD` mechanism (CIESLAK, 2015). This allows our tool to integrate smoothly with the application without requiring any modifications to the application's source code. With this non-intrusive approach, our tool can optimize the performance of a wide range of applications fully automated and transparently. Next, we detail the steps mentioned above.

5.2.1 Gathering Machine Topology Information

After initializing the application's execution, our next objective is to accurately perceive the topology of the machine on which our application runs. This is a nontrivial task that relies on understanding the particularities of the underlying architecture, taking into account its many components, such as the total number of processors, individual cores per processor, and the degree of SMT supported by each core. An essential consideration in this process is identifying and characterizing the system's hierarchical organization, the memory topology, and the shared resources across the system's processing units.

To achieve this, we leveraged the capabilities of the Hardware Locality (hwloc) software project (BROQUEDIS et al., 2010). Hwloc is an effective tool for collecting comprehensive topology-related information about the system. The data it gathers includes, but is not limited to, information on Non-uniform memory access (NUMA) memory nodes, sockets, shared caches, individual processing cores, and SMT threads. The power of hwloc lies in its ability to map the hierarchical architecture of the system and retrieve essential details regarding its structure. This provides a solid foundation for more informed scheduling and resource allocation decisions. This mapping also allows our tool to understand the topology's granularity, helping in the efficient and optimal mapping of threads to cores.

The hwloc tool also offers an additional advantage. It provides an interface for traversing the topology tree, which facilitates our tool's task of mapping threads to the resources in a systematic manner. With the help of this tool, we can identify potential bottlenecks, such as memory node contention, cache misses, or the overuse of a particular functional unit, and optimize our thread scheduling accordingly.

Following the successful detection of the machine's topology, our tool employs this data to efficiently distribute the threads across the system resources, considering the intricacies of the architecture.

5.2.2 Detecting the Instruction Pattern

As established in Chapter 4, within an SMT processor, threads mapped to the same physical core naturally compete for the core's resources. The severity of this competition is particularly accentuated when the threads in contention execute similar types of instructions that place demands on the same functional unit, thereby causing an intensification in resource contention.

To alleviate this issue, our tool prioritizes detecting the types of instructions each thread executes. This insight allows our tool to enhance system performance by actively directing threads likely to stress the same functional unit toward different cores. Therefore, resource contention is reduced via intelligent thread distribution guided by the instruction types detected.

In the third step of our tool, we concentrate on detecting the instruction pattern of each running application. For this purpose, we use the PAPI (JOHNSON et al., 2012). PAPI is a versatile tool that provides an interface to numerous processor hardware counters, including but not limited to the number of executed instructions.

PAPI can segregate instructions into distinct categories: branch, floating-point, integer, and load and store instructions. The specific events measured by PAPI are extensively listed in Table 5.1.

Table 5.1 – PAPI Events measured.

Papi event	Description
`PAPI_BR_INS`	Branch instructions
`PAPI_FP_INS`	Floating point instructions
`PAPI_INT_INS`	Integer instructions
`PAPI_LD_INS`	Load instructions
`PAPI_SR_INS`	Store instructions
`PAPI_TOT_INS`	Total instructions completed

Detecting instruction patterns in our IAM tool is not a singular event but a continuous monitoring activity every second. This frequency was carefully chosen, considering the balance between the precision of instruction detection and the potential for increased computational overhead, as might be seen with more granular intervals facilitated by certain hardware counters or the PAPI. By opting for a one-second frequency, our tool can swiftly and efficiently identify changes in application phases or behavior patterns without unnecessarily increasing CPU utilization for the monitoring process, thus potentially interfering with the applications under observation. Upon detecting a change, the IAM tool immediately responds with a two-step adaptive process, verifying the change and implementing an optimization strategy tailored to the newly identified application behavior. The decision to use a one-second monitoring frequency in our IAM tool was guided by carefully considering the trade-off between accuracy and computational overhead. Hardware counters and the PAPI library offer the capability for exact, granular monitoring. However, invoking these counters at high frequencies can lead to significant computational overhead, as the system needs to spend more CPU cycles on monitoring activity. This can potentially interfere with the observed applications, negatively impacting their performance and the overall system throughput. This monitoring frequency also ensures scalability, accommodating potential future incorporation of more complex features and additional performance counters while maintaining efficiency and responsiveness.

We continuously monitor instruction patterns and ensure our tool attains dynamic workload behavior. This allows it to adjust thread mapping decisions in response to phase changes in the applications' execution, mitigating resource contention and enhancing the overall system performance.

5.2.3 Calculating the Mapping

To evenly distribute the computational load across the cores and functional units, we developed a mapping algorithm that uses the instruction pattern data gathered in step 3. This algorithm is designed to respond to the different types of instructions the threads execute. As highlighted in Chapter 4 and further substantiated by Scenario C (distributing threads that stress distinct types of functional units), our tool to load balancing is rooted in the principles of Gauss sum theory. This theory is utilized to balance the number of the same type of instructions from each thread running on a core, ensuring that the total sum of a given instruction type is almost identical across all cores.

While the experimental data presented in this thesis underscores the relevance of floating-point instructions, it is crucial to assert that our algorithm retains an appreciable degree of versatility, facilitating its adaptability to diverse instruction types. We have observed through our assessments that benchmarks, such as the NPB, primarily consist of floating-point instructions. Our analysis demonstrated that floating-point and integer instructions represent a significant portion of these benchmark instruction sets. Nevertheless, our tool is engineered to retain its adaptability. The tool can dynamically shift its focus to integer instructions in scenarios where floating-point instructions are scant or absent. This inherent adaptability of our algorithm caters to the diverse and dynamic character of computational workloads, thus enhancing the overall efficacy of our Instruction-Aware Mapping (IAM) tool. This essential ability to adapt based on instruction characteristics is critical to achieving optimal performance across various applications.

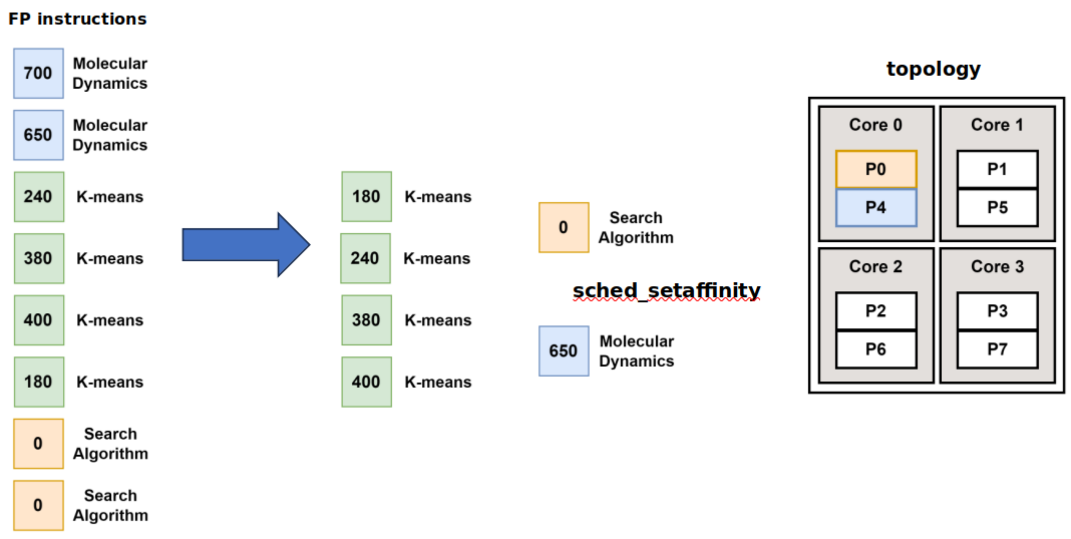


Figure 5.2 – IAM Mapping Calculation

As demonstrated in Figure 5.2, the inputs to our algorithm include a data structure that encapsulates the machine's topology and an array that denotes the quantity of floating-point instructions executed by each thread in every application. The algorithm commences by sorting the threads based on the volume of selected instruction types. It then proceeds to map threads to the same core following a mirrored placement pattern—(0, n), (1, n-1), (2, n-2), etc., until all threads have been assigned to a core.

In the algorithm's initial loop iterations, the primary focus is on the initial scheduling of threads across different cores. This phase effectively mitigates potential interference from co-runners and suppresses the likelihood of resource contention. If any threads remain to be scheduled after this stage, they are addressed in the subsequent while loop. The objective is strategically mapping these residual threads to minimize interference and mitigate contention.

The design of our algorithm promotes a balanced workload distribution across distinct cores. This is achieved by factoring in each thread's specific types of instructions.

As a result, the thread-to-core mapping process becomes dynamic. It ensures an equitable distribution of threads, optimizing overall performance and hardware resource utilization.

The mapping calculations are conducted at a one-second granularity, mirroring the hardware counters' measurement frequency. However, the actual thread migration or remapping is activated only when more than two threads meet the criteria for migration, thereby ensuring efficient use of computational resources and limiting unnecessary thread movements. The primary reason for setting the thread migration criterion at more than two threads is to balance performance optimization and resource efficiency. Thread migration in a multi-core system can be a costly operation. This is due to the overhead associated with moving a thread from one core to another, such as cache invalidation, the creation of the new execution environment, and the time spent re-establishing data locality, among other factors.

5.2.4 Mapping Threads

Following the detection of the instruction pattern, the acquisition of the machine's topology, and the calculation of the optimal thread mapping, our tool proceeds to actualize the computed mapping for the current application. This operation necessitates certain support functionalities from the operating system. In the case of Linux, the operating system provides the `sched_setaffinity` system call, which enables setting a thread's affinity or binding to a specific core (LOVE, 2003).

It is important to emphasize that our tool does not require any modifications to the application's source code. We have developed a tool that employs the `LD_PRELOAD` environment variable, a mechanism the Linux dynamic linker provides. By leveraging this mechanism, our tool can inject a shared library into the application's address space during the application's loading phase. The injected library contains the functionality required to perform all the steps outlined previously, culminating in the execution of the `sched_setaffinity` call.

By invoking the `sched_setaffinity` call, our tool maps each application thread to its designated core per the calculated mapping. This mapping operation considers the machine's topology and the application's instruction pattern to optimize the use of the hardware resources and minimize functional unit contention.

In this way, our tool seamlessly integrates with the application's normal execution process, taking care of the entire process from topology detection, instruction pattern recognition, and optimal thread mapping calculation to the actual thread-to-core mapping. This non-intrusive, intuitive approach aids in ensuring that the application can benefit from optimized thread mapping, resulting in improved performance without necessitating code-level modifications.

5.3 Closing Remarks

This chapter has elucidated the process and inner workings of our proposed online mapping tool designed to mitigate functional unit contention in computing applications running on SMT processors.

We began the chapter by highlighting the inherent problem in SMT processors - threads mapped to the same core compete for functional units, leading to potential performance degradation. The tool we have proposed, IAM, is designed to improve performance by dynamically mapping threads to cores, considering the instruction types each thread executes.

We detailed the IAM tool's five-step process: initiating the application's execution, gathering machine topology, detecting instruction patterns, calculating thread-to-core mapping, and implementing this mapping. These steps are continuously executed in a loop for the application's execution, enabling the tool to adapt to changing instruction patterns or machine states.

Our solution uses tools like hwloc and PAPI to gather information about machine topology and access hardware counters for instruction pattern detection. Furthermore, we developed an intelligent mapping algorithm rooted in the Gauss sum theory to balance the number of similar instruction types across cores.

Importantly, we emphasized that our tool is entirely non-intrusive, requiring no modifications to the application's source code. This is achieved through the `LD_PRELOAD` mechanism, allowing the tool's library to be loaded with the application's binary.

The approach outlined in this chapter has the potential to significantly improve the performance of parallel applications on SMT processors by intelligently mitigating functional unit contention.

In conclusion, the concepts and mechanisms introduced in this chapter demonstrate a significant stride towards optimized utilization of modern computing resources and offer a promising avenue for future research and development.

6 EXPERIMENTAL EVALUATION

This chapter delivers an exhaustive performance analysis of our proposed tool to mitigate functional unit contention. The investigation incorporates a diverse set of benchmarks that include SMT-Bench, NPB Benchmark, and SPEC Benchmark. These benchmarks, covering a wide range of workload scenarios, were chosen to ensure that our experiments encompass the various intricacies within the computing domain.

6.1 Performance Evaluation Using SMT-Bench on AMD Processors

We conducted an exhaustive performance analysis using various application combinations from SMT-Bench. The results derived from these experiments offer substantial insight into the effectiveness of our tool, allowing us to compare it with the ubiquitous Linux Scheduler (Linux scheduler) and the traditional Round-Robin (Round-robin) mapping method. It should be noted that Linux scheduler epitomizes the standard approach to thread-to-core mapping in contemporary operating systems, whereas Round-robin serves as a straightforward yet widely adopted scheduling method, providing a baseline for our comparison.

The accompanying figures illustrate the weighted speedup obtained through a blend of SMT-Bench applications. The y-axis represents the weighted speedup, while the x-axis depicts the various application combinations. Each bar on the graph denotes a distinct thread-to-core mapping technique. On a geometric mean basis, our tool, IAM, registers a weighted speedup of 2.2, surpassing Linux scheduler's 2.0 and marking an average performance improvement of 9.8%. The performance enhancement attributable to IAM ranges from -5.7% to an impressive 37.4% in comparison to Linux scheduler.

The graphical representation convincingly demonstrates that IAM significantly improves performance over the other evaluated techniques. Regarding execution time, our tool outperforms the Linux scheduler scheduler by an average of 9.8%. This increase in performance is attributable to our tool's design, which integrates functional units' usage patterns to deliver optimized thread-to-core mapping, thereby reducing contention and enhancing overall application performance.

When contrasted with the Round-robin mapping, our tool exhibits an even more pronounced average performance improvement of 24.6%. This marked improvement is a testament to our tool's capability to outperform non-contention-aware scheduling techniques like Round-robin significantly. This is primarily due to its careful consideration of instruction patterns and hardware unit requirements for effective mapping.

These final results highlight the promising capabilities of our tool. This comprehensive performance evaluation validates our tool's effectiveness, providing a strong foundation for future work. Our tool has proven worth as a highly efficient resource that can significantly reduce functional unit contention in multi-threaded processing environments.

6.1.1 Performance Enhancement in Branch-Intensive Applications

The `branch` microbenchmark application, known for its heavy reliance on branch instructions that make up the majority of its instruction set, provides a pertinent scenario to evaluate the effectiveness of our tool. In this context, IAM displayed superior efficacy by delivering a performance improvement of 6.6% compared to the conventional Linux scheduler scheduler. These results underscore IAM's unmatched ability to alleviate functional unit contention strategically, thereby enhancing performance strategically.

Figure 6.1 presents the performance outcomes for the branch application operating as a runner. For example, when coupled with `fp-add` and `fp-mul`, IAM significantly augmented performance by 16.1% and 16.0%, respectively. These findings underline IAM's robust capabilities in managing floating-point operations. Conversely, when `branch` co-runs with `int-add` or `int-mul`, IAM maintains an enhancement in performance, although at a marginally lower rate of 3.2%. This consistent performance improvement across a range of instruction types substantiates the comprehensive effectiveness of IAM.

Interestingly, when `load` is employed, IAM trails the OS scheduler by 5.7%, hinting at specific scenarios where cache-critical loads might benefit more from the operating system's default scheduling strategies. These strategies favor thread placement on the same core to leverage shared cache levels. While this case emphasizes the need for further refinement to bolster performance across all instruction types, it does not overshadow IAM's typical advantage over the OS scheduler in most other scenarios.

IAM operates based on an intelligent thread-to-core mapping strategy that meticulously accounts for each thread's characteristic workload. This approach ensures threads with similar functional unit demands are assigned to different cores, effectively mitigating performance bottlenecks that could stem from concurrent access to identical functional

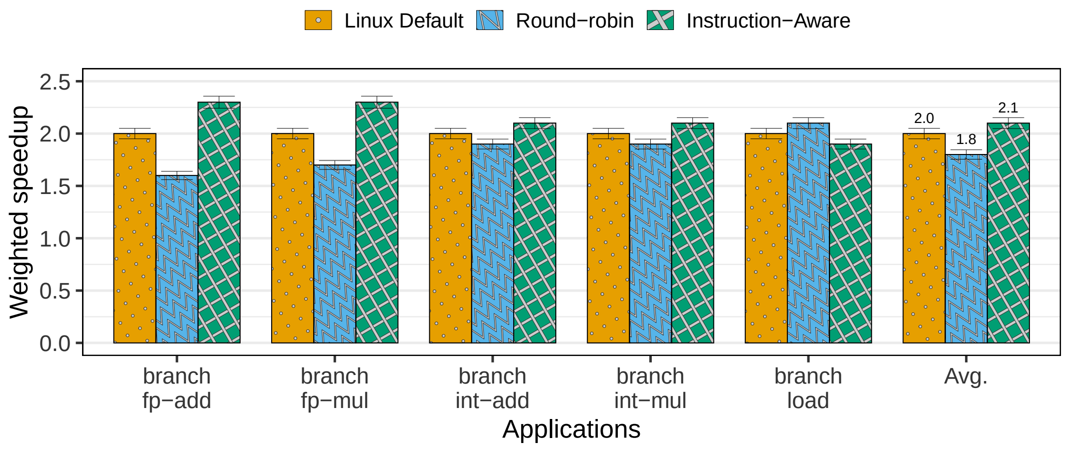


Figure 6.1 – Weighted speedup (higher values are better) for branch applications co-running with diverse applications from SMT-Bench.

units. Such strategic separation, managing parallel resource demand efficiently, is critical to preventing potential performance degradation and amplifying the overall computational throughput.

The results further affirm the value of IAM, showcasing its potency in mitigating resource contention issues, thereby boosting system performance, especially in the realm of SMT-based processors. In conclusion, these performance evaluations affirm that IAM provides an impactful advancement in resource utilization efficiency for computing applications, marking a significant milestone in this field.

6.1.2 Performance Enhancement in Floating-Point-Intensive Applications

Our tool, IAM, recorded substantial improvements when applied to scenarios featuring applications primarily engaging the floating-point functional units. A notable example is the co-execution of `fp-add` and `fp-mul` benchmarks. These applications, designed to utilize the FADD and FMUL units intensively, displayed heightened performance when scheduled with IAM. Specifically, performance surged by an impressive 21.4% in geometric mean compared to Linux scheduler and even more markedly by 53.6% compared to the Round-robin scheduler.

These observations underscore the effectiveness of IAM in contexts where applications necessitate diverse functional units. IAM achieves superior computational throughput by intelligently mapping threads to cores, minimizing resource contention and optimizing available functional units.

Figure 6.2 showcases the performance results of the floating-point applications functioning as runners. When `fp-add` serves as the runner, IAM consistently outperforms the OS scheduler across all co-runners, with percentage gains ranging from 16.1% with `branch` to a striking 37.4% with `fp-mul`. Similarly, when `fp-mul` operates as a runner, IAM continues to outpace the OS scheduler, demonstrating performance increases ranging from 16.0% with `branch` to 37.4% with `fp-add`.

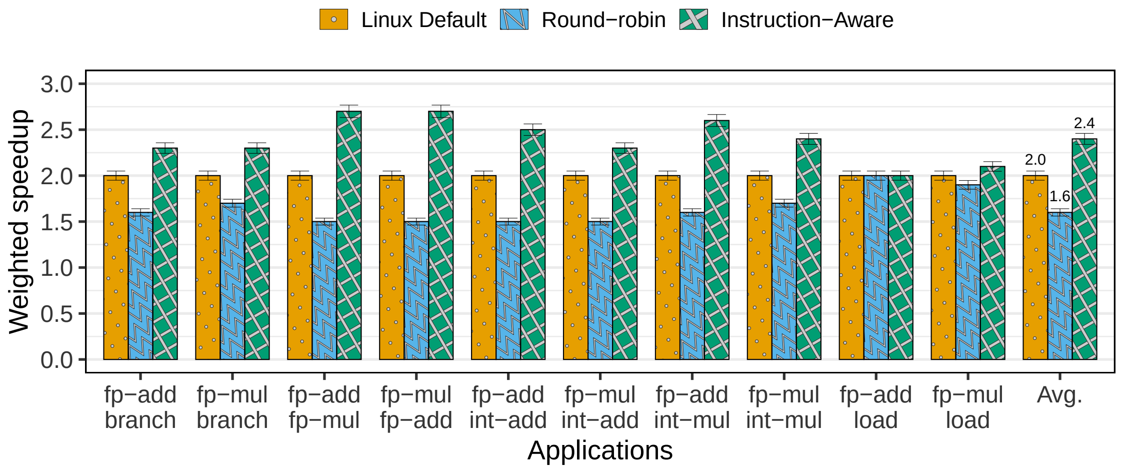


Figure 6.2 – Weighted speedup (higher values are better) for floating-point applications co-running with diverse applications from SMT-Bench.

An intriguing exception to this trend arises when `fp-add` or `fp-mul` co-run with `load`. In these instances, IAM exhibits minor underperformance compared to the OS scheduler, with performance reductions of 2.4% and 3.7%, respectively. This deviation echoes the insights gathered from the branch application evaluations. Such scenarios underscore the importance of ongoing refinement of IAM to ensure consistent performance improvements across the board.

These findings reaffirm IAM's ability to proficiently manage diverse computational workloads, fortifying its reputation as a resilient and adaptable tool to mitigate functional unit contention in SMT-based systems. This, in turn, drives superior system performance and throughput, confirming IAM's value as a practical approach for a broad range of computing applications.

6.1.3 Performance Enhancement in Integer-Intensive Applications

Evaluating IAM's performance with integer-based instructions reveals a consistent trend of IAM surpassing the native operating system scheduler, with only a few exceptions.

Figure 6.3 presents the performance results of integer-intensive applications functioning as runners. With `int-add` as the runner, IAM demonstrates marked superiority over the OS scheduler across most co-runners. Specifically, the percentage performance gains of IAM over the OS scheduler span from 26.8% with `fp-add` to 3.2% with `branch`.

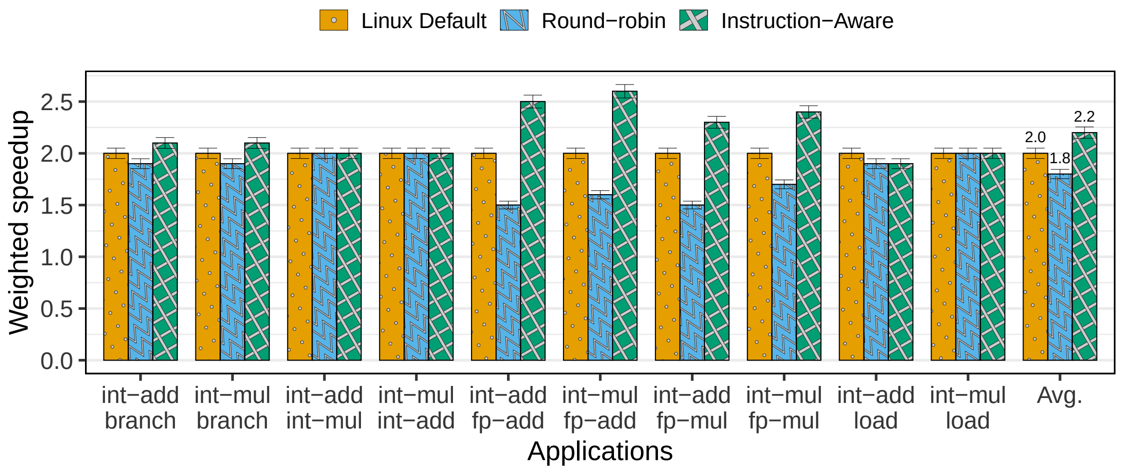


Figure 6.3 – Weighted speedup (higher values are better) for integer applications co-running with diverse applications from SMT-Bench.

Similarly, for `int-mul`, IAM outperforms the OS scheduler for most co-runners. The percentage gains range from 29.2% with `fp-add` to 3.2% with `branch`. However, when `int-add` and `int-mul` co-run with `int-mul` and `int-add`, respectively, IAM's performance mirrors that of the OS scheduler, with a negligible performance gain of 1.9%. This is because these applications are both integer-based and share specific functional units. Moreover, when `int-add` and `int-mul` co-run with `load`, IAM underperforms slightly, with performance reductions of 2.6% and 1.7%, respectively. In summary, for integer-based instructions, IAM predominantly outperforms the native OS scheduler, solidifying its effectiveness.

6.1.4 Performance Enhancement in Load-Intensive Applications

The results portray a slightly different narrative when executing the `load` benchmark in conjunction with other applications. `load` places strain on the load units of the processor, leading to unique performance outcomes.

Figure 6.4 illustrates the performance results of the load application functioning as a runner. With `load` as a runner, IAM's performance closely aligns with that of the OS scheduler across various co-runners. For instance, when co-running with `fp-add`, and `int-mul`, IAM's performance is slightly under the OS scheduler's, with marginal decreases of 1.7%, 2.4%, and 1.7%, respectively. However, IAM exceeds the OS scheduler's performance by 3.7% when paired with `fp-mul`. These percentage differences are relatively low and within the confidence interval of the measurements, implying that the observed variations are not statistically significant.

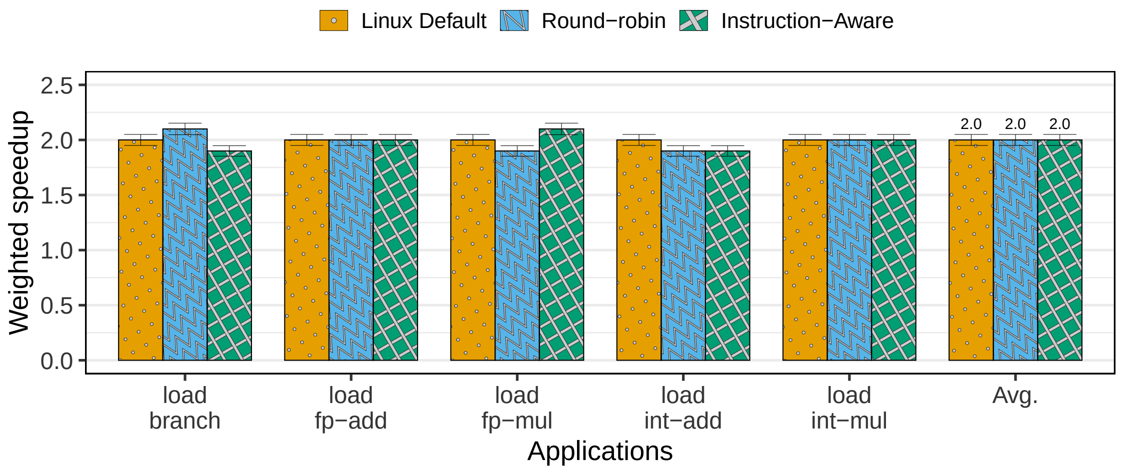


Figure 6.4 – Weighted speedup (higher values are better) for load applications co-running with diverse applications from SMT-Bench.

There are instances when the performance of IAM and the OS scheduler are virtually identical, such as when `load` co-runs with `fp-add` and `int-mul`, where the performance variations are a mere 1.1% and 0.3%, respectively. Moreover, when `load` co-runs with `branch`, IAM's performance decreases by 5.7%

While IAM optimizes performance for integer, floating-point, and branch instruction types, it slightly underperforms when handling load instructions. This deviation is attributed to the inherent characteristics of load instructions and the way IAM manages these workloads.

Load instructions typically involve substantial memory operations, making them latency-sensitive and reliant on data availability fetched from the memory hierarchy. This fetch operation can be time-consuming and introduce delays in the execution pipeline, mainly when the required data is not immediately available in the cache. As a result, load instructions can potentially suffer from stalls, leading to inefficient use of CPU cycles.

In contrast, integer, floating-point, and branch instructions are more compute-intensive and less likely to be affected by memory latency. These characteristics allow IAM to predict better and optimize the scheduling of such instructions across threads and cores.

IAM functions by mapping threads onto the processor based on the instruction type primarily executed by each thread. This strategy works well for compute-intensive instructions, as it mitigates contention for the functional units that these instructions rely on. However, the main performance bottleneck for memory-bound load instructions is usually the memory system rather than the functional units. Therefore, IAM's thread mapping strategy might need to be adjusted to handle this bottleneck and avoid suboptimal performance effectively.

The latency induced by the memory system during the execution of load instructions poses a substantial challenge that the current IAM design cannot entirely address. This suggests a more sophisticated scheduling mechanism considering memory behavior and instruction types.

6.2 Performance Evaluation Using NPB and SPEC Benchmarks

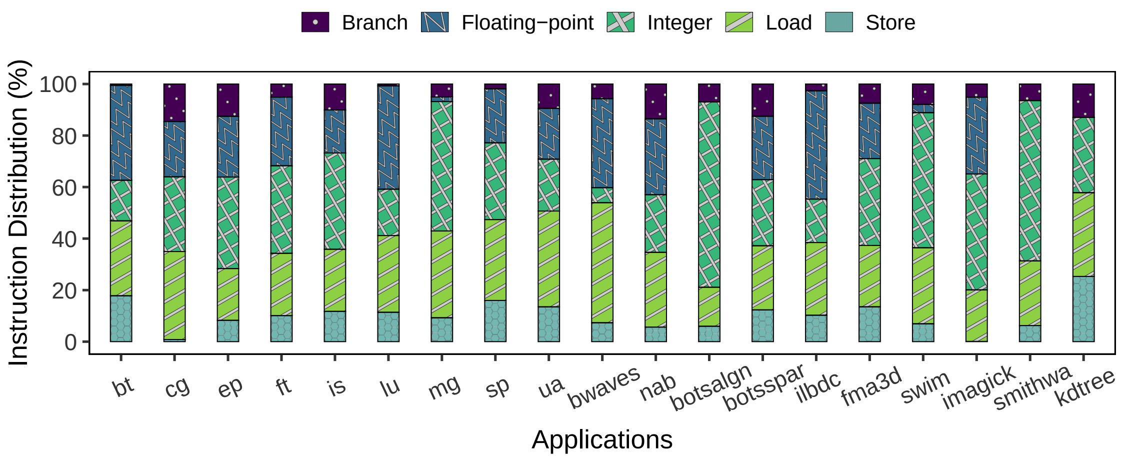


Figure 6.5 – NAS and SPEC Benchmarks instruction distribution.

In Chapter 5, we introduced our tool's third step - determining the instruction patterns of each application. This critical information guides our decision-making process regarding the optimal placement of threads. The instruction distribution for all NAS and SPEC Benchmark applications is illustrated in Figure 6.5, with the y-axis indicating the instruction distribution percentage and the x-axis displaying the applications. The bars represent different instruction types.

Examining the BT application, we observe most instructions as floating-point and load-based. Given their shared focus on floating-point instructions, co-running applications such as LU with BT could negatively impact BT's performance. However, pairing BT with applications like IS and MG, which primarily deal with integer operations, could enhance overall system performance.

In the case of CG and UA, most instructions involve loads. UA also features random data access. Running these applications could hinder performance for both. Applications like EP and FT might be better-suited co-runners for CG and UA, as they primarily concentrate on integer and floating-point operations and execute fewer load instructions.

We extend our analysis to the instruction pattern of SPEC benchmarks, which include: bwaves, nab, botsalgn, botsspar, ilbdc, fma3d, swim, imagick, smithwa, and kdtree. bwaves primarily executes floating-point operations and load instructions. If co-run with nab, which is also heavy on floating-point operations, contention for the floating-point unit may arise. However, pairing it with botsalgn, integer-operation dominant, could reduce contention and improve overall performance.

Nab showcases a high volume of branch and floating-point instructions but negligible store operations. As such, they are paired with an application like ilbdc, which demonstrates fewer branch operations and a balanced mix of other instruction types. In contrast, botsalgn mainly executes integer instructions and has fewer load instructions than most other applications. It performs minimal floating-point operations, making it a suitable candidate for pairing with floating-point-intensive applications such as bwaves or nab.

Both botsspar and kdtree present balanced instruction distributions. They might pair well with applications with heavily skewed instruction types, utilizing computational resources that the other application does not monopolize. ilbdc, much like bwaves, executes many floating-point and load instructions. Pairing it with nab, another floating-point-intensive application, might result in performance degradation due to contention for the floating-point unit.

Applications like fma3d and swim also display balanced instruction distributions. Co-running them with applications with a strong preference for a particular instruction type may yield good performance, as these applications can tap into underutilized computational resources. imagick primarily employs integer and load operations. A pairing with smithwa, focusing on branch and store operations, may help reduce resource contention.

Finally, smithwa is primarily oriented toward branch and integer instructions. Co-running it with an application like bwaves, which relies heavily on loads and floating-point operations, may enhance performance. Such considerations are crucial when designing and implementing thread scheduling algorithms, enabling optimal hardware utilization and overall system performance.

Chapter 4 demonstrated the impact of resource sharing using synthetic microbenchmarks under varying scenarios. We established that mapping applications that execute the same instruction on different cores enhances application and overall system performance. This section illustrated that real-world benchmarks also present opportunities for performance improvement through intelligent mapping. The following section will showcase the performance enhancements facilitated by our instruction-aware mapping.

6.2.1 Performance Improvements on Intel Processors

When executed on Intel processors, our extensive performance investigation of three specific applications - swim, botsalgn, and bwaves has yielded valuable insights into the system's performance dynamics. Using weighted speedup as our performance metric, we have precisely quantified the speedup of application performance about a reference scenario. The results from this analysis can be seen in Figures 6.6, 6.7, and 6.8 for the swim, botsalgn, and bwaves applications, respectively, illustrating their performance when co-running with a variety of applications.

In particular, Figure 6.6 demonstrates that the swim application experiences notable performance enhancements when co-run with specific applications like botsalgn and botsspar. The weighted speedup in these instances is impressively high, reaching 4.1 and 6.7, respectively. This indicates beneficial synergies between swim and these applications, resulting in the Swim application's efficient usage of the available resources. However, the speedup decreases below the baseline value of 2 when the swim is co-run with ilbdc and kdtree, indicating potential performance bottlenecks.

Turning our attention to the botsalgn application, Figure 6.7 displays optimal performance when it is co-run with bwaves and swim, resulting in speedups of 4.0 and 4.0,

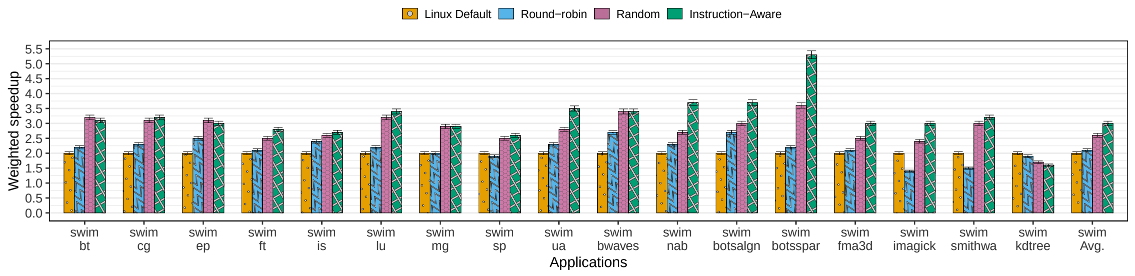


Figure 6.6 – Weighted speedup (higher values are better) for **swim** application co-running with diverse applications.

respectively. This substantial speedup implies that botsalgn can leverage the system resources efficiently when paired with these applications. Conversely, when botsalgn is co-run with ep and ilbdc, the speedup falls below the baseline, suggesting possible performance issues requiring optimization.

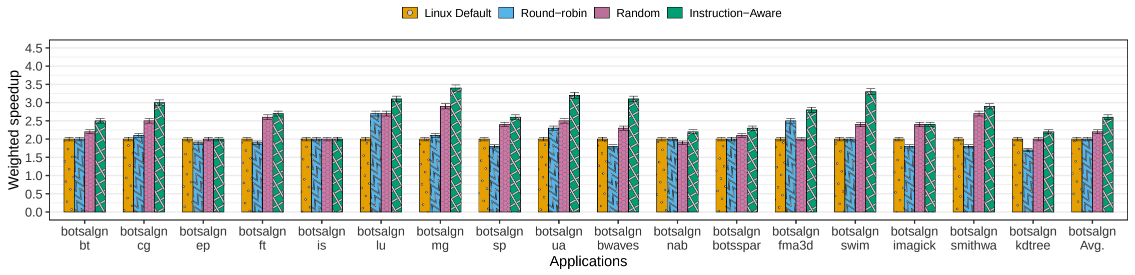


Figure 6.7 – Weighted speedup (higher values are better) for **botsalgn** application co-running with diverse applications.

Lastly, as seen in Figure 6.8, the bwaves application exhibits substantial speedups when co-run with botsalgn and imagick, achieving scores of 4.0 and 4.3, respectively. This performance boost suggests efficient sharing of system resources between bwaves and these applications. However, like the other applications, bwaves also shows decreased speedup values (below 2.0) when co-running with specific applications such as kdtree and lu, signifying potential resource contention or bottlenecks.

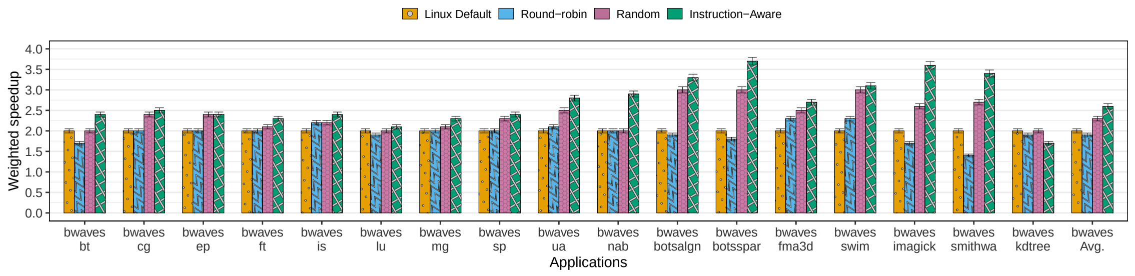


Figure 6.8 – Weighted speedup (higher values are better) for **bwaves** application co-running with diverse applications.

It is essential to discuss further the crucial role that functional units in each processor core play in these results. Each core contains several functional units, ALUs, FPUs, and others. These units can perform tasks in parallel, enhancing the core's overall performance. However, using the same functional units by multiple applications can lead to resource sharing and potentially degrade performance, as seen in some instances mentioned above. Therefore, devising strategies for efficient resource sharing and scheduling of these functional units is critical to optimizing application performance when co-running multiple tasks.

6.2.2 Performance Improvements on AMD Processors

Table 6.1 outlines the performance results of Instruction-Aware mapping on the NPB. As an illustrative example, consider MG as an application and EP as a co-runner; MG's performance was elevated by 29.8%. We derived these results from the average of 30 randomly executed tests under exclusive machine access. Our findings contrast the performance of our solution with those achieved through Linux scheduler and Round-robin mapping. The execution time results were normalized to the Linux scheduler; thus, shorter bars in the figures denote superior performance. We computed the standard deviation using the t-Student distribution with a 95% confidence interval. The input size C was selected to provide a feasible execution time.

Table 6.1 – Performance Improvement of the Instruction-Aware Mapping on the NPB.

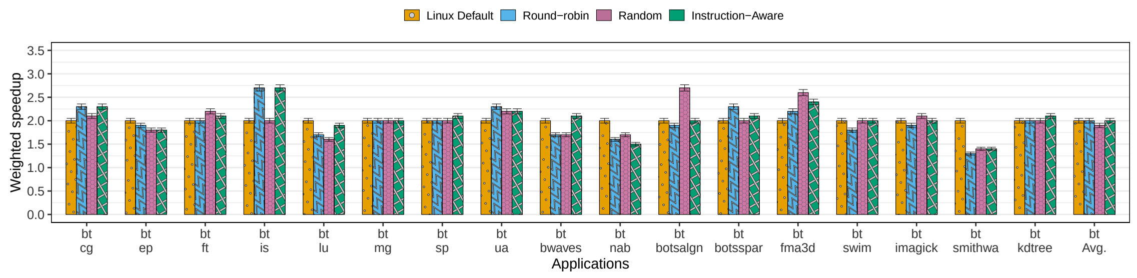
	Co-Runner								
	BT	CG	EP	FT	IS	LU	MG	SP	UA
BT		23.7%	8.8%	14.9%	13.5%	10.9%	23.9%	21.3%	20.8%
CG	24.4%		17.2%	11.8%	23.7%	23.3%	19.9%	26.8%	25.6%
EP	11.2%	16.4%		3.7%	2.2%	13.1%	29.8%	11.8%	1.9%
FT	14.7%	9.8%	6.1%		13.8%	16.2%	27.4%	17.5%	17.7%
IS	14.6%	19.4%	3.8%	14.2%		10.9%	19.2%	17.4%	14.7%
LU	11.5%	23.1%	5.9%	11.4%	10.3%		17.1%	15.3%	11.9%
MG	20.1%	22.4%	29.8%	24.9%	13.6%	18.7%		23.5%	23.2%
SP	22.3%	26.3%	10.8%	17.9%	14.8%	13.3%	21.2%		21.7%
UA	21.1%	26.2%	11.9%	17.3%	15.2%	11.2%	19.5%	22.2%	

The following section presents results classified into four groups based on application similarity. The K-means clustering algorithm was used for this classification, where the inputs were the applications and the quantity of each instruction type. The Elbow method (KASSAMBARA, 2017) was employed to identify the ideal number of clusters for our application data. We only highlight the most significant results, as some are omitted due to their similar behavior to the ones discussed.

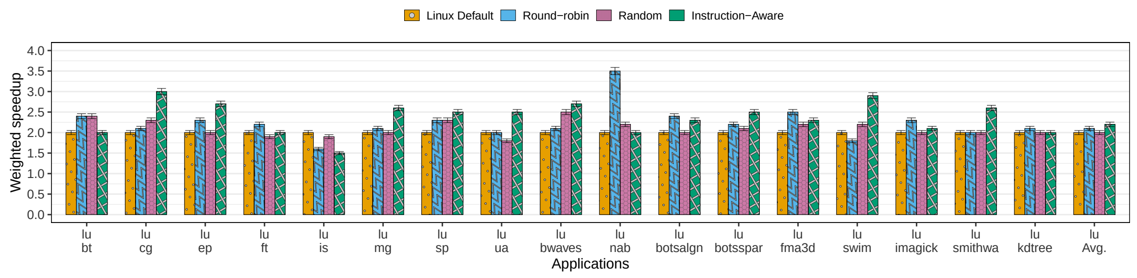
The first group comprises BT and LU, applications that execute many floating-point instructions. The second group contains MG, an application with the most integer instructions. The third group includes CG, SP, and UA, applications dominated by load instructions. Finally, the fourth group contains EP, FT, and IS, which display a balanced distribution of floating-point, integer, and load instructions.

The weighted speedup obtained with each application, running with different co-runners, is shown. The geometric mean of the weighted speedup for all co-runners is 2.4, denoting a 20% performance improvement. When the co-runner is LU, an application that performs many floating-point and load operations, the performance improves by 10% over Round-robin. Since the applications share the same functional units, mapping them on the same or different cores results in almost identical performance improvement. However, when the co-runner is MG, an application with most integer operations, the performance is 30% better than the Linux scheduler. It demonstrates that the operation type of each thread is a crucial factor when mapping multiple parallel applications.

BT and LU instructions are mostly floating-point. Figure 6.9 presents the performance results for this group. We expected more considerable improvements when the applications that stress different functional units are mapped on the same core. When MG is the LU co-runner, our framework's performance is 20% and 30% better than the Linux scheduler and Round-robin. This happens because MG executes integer instructions most



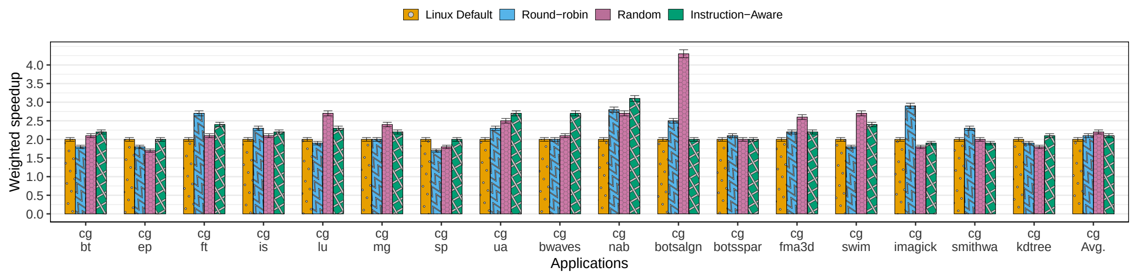
(a) **BT**



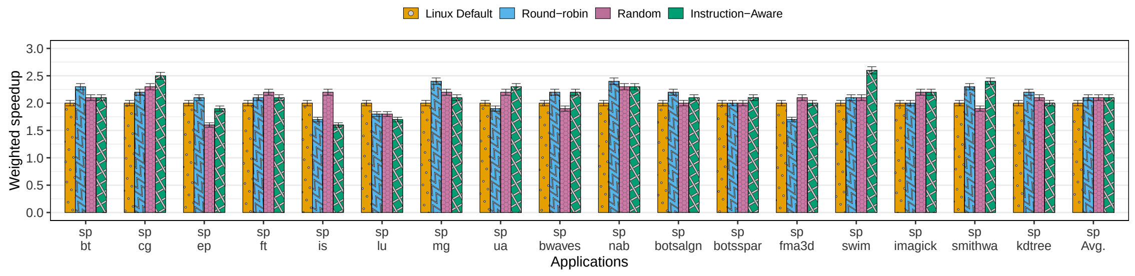
(b) **LU**

Figure 6.9 – Weighted speedup (higher values are better) running NAS applications focusing on floating-point instructions.

of the time. The opposite occurs when a floating-heavy application is the co-runner of LU. The contention is not reduced because both applications stress the same units.



(a) **CG**



(b) **SP**

Figure 6.10 – Weighted speedup (higher values are better) running NAS applications focusing on load instructions.

MG forms the second group and executes integer operations in 50.2% of its instructions. The geometric mean speedup for MG is 30% compared with the Linux scheduler and 40% compared with Round-robin. The gain is modest for co-runners with the same units, such as IS. This is because MG and IS execute many integer instructions, so any mapping still taxes the integer units.

CG, SP, and UA form the third group, where most instructions are loaded. Figure 6.10 presents the performance results for this group. The geometric mean speedup is 20% and 30% compared with Linux scheduler and Round-robin, respectively. The speedup is higher for co-runners that do not share the same type of instruction.

The final group contains EP, FT, and IS, which have a balanced distribution of floating-point, integer, and load instructions. The geometric mean speedup for EP, FT, and IS is around 15% when compared with both Linux scheduler and Round-robin. The speedup is higher when the co-runner has a different dominant instruction type.

Overall, our solution outperforms both Linux scheduler and Round-robin across all groups. The solution is particularly effective when applications have different dominant instruction types. This suggests that our approach provides an efficient and flexible way to map multiple applications in multicore processors, improving overall performance.

6.3 Discussion and Guidelines

Our study reveals crucial aspects of SMT processors' performance when executing varying instruction types. It paves the way for further potential enhancements. The IAM tool we introduced provides a fresh and pragmatic thread scheduling approach that substantially improves compute-intensive workloads' efficiency. The tool's performance, however, fluctuates across diverse instruction types, highlighting the need for instruction-specific considerations when developing and implementing SMT schedulers.

Our experimental outcomes illustrate the value of mapping algorithms that account for the instruction types executed by each thread in augmenting overall performance. Our research underscored that performance hinges on the kind of instructions executed by each concurrently running thread. We categorized the applications into four groups based on the predominance of a particular instruction type.

Compute-Intensive Instructions: The IAM tool delivers excellent performance for compute-intensive instruction types such as integer, floating-point, and branch operations. It employs an innovative method of mapping threads to cores based on instruction types, thereby mitigating contention for functional units and leading to substantial performance gains. Consequently, for workloads that primarily use these instructions, IAM emerges as an up-and-coming alternative to conventional operating system schedulers.

Memory-Bound Instructions: For memory-bound instruction types, especially load and store operations, the performance benefits of IAM are less evident. This variation underscores the influence of factors beyond functional unit contention, notably memory latency, in shaping SMT performance. While IAM may not be optimal for these workloads in its current version, its instruction-aware scheduling approach offers opportunities for refinement. Future iterations of IAM could incorporate strategies to anticipate and alleviate memory latencies, such as dynamic adjustments to the scheduling strategy based on real-time memory utilization patterns.

Guidelines for SMT Optimization: Our study emphasizes the value of instruction-specific considerations in optimizing SMT performance. The generalized strategy may only partially exploit the capabilities of SMT processors. Instead, a more refined approach accounting for the specific characteristics of the workload could be adopted. Strategies like IAM reduce functional unit contention and can provide substantial performance benefits for compute-intensive instructions. Factors related to memory latency and cache behavior become critical for memory-bound instructions. Continued exploration of instruction-specific SMT optimization strategies is warranted, and the potential for tools like IAM to dynamically adapt to varying workloads should be examined.

Our findings advance our understanding of SMT performance behavior and the influence of instruction-level characteristics in shaping this behavior. This knowledge is vital in developing innovative solutions to enhance computational efficiency, especially as processors continue to evolve and applications become increasingly diverse and complex.

Moreover, our tool is adaptable to any computing environment, enhancing resource allocation and facilitating thread mapping based on specific workload characteristics. It can be beneficial for both infrastructure providers and users. Providers could implement this mechanism across their computing infrastructure to take advantage of improved mapping. At the same time, users could leverage this mechanism to optimize their workloads, thus boosting computational efficiency and performance.

6.4 Summary

In this chapter, we presented and examined our experimental results, highlighting the significant potential of our tool to reduce functional unit contention in SMT-based systems. We outlined the performance enhancements achieved when different application combinations from the SMT-Bench, NPB Benchmark, and SPEC Benchmark were executed under various scheduling policies. The experiments showcased our tool's remarkable ability to surpass both Linux scheduler and Round-robin schedulers, mainly when applications that stress different functional units are executed simultaneously. These findings reinforce our research's primary hypothesis: mapping threads to cores according to their functional unit demands can significantly improve performance in SMT processors.

7 CONCLUSIONS AND FUTURE DIRECTIONS

In the face of continually evolving and complex multicore processors, the implications of functional unit contention are expected to amplify. This trend is driven by the prospective architectures where increasing cores share functional units, exacerbating the challenge of mitigating resource contention. Consequently, as discussed in this thesis, devising efficient mapping strategies will be instrumental in optimizing thread allocation across cores. This will allow for the full utilization of computational capabilities in forthcoming multicore architectures, despite the complexities associated with resource sharing.

Our research has pioneered a unique microbenchmark to evaluate the impact of resource sharing methodically. This tool, which deviates from the traditional focus on memory contention, considers contention across various functional units, offering a more nuanced understanding of resource-sharing dynamics in multicore environments.

We have developed an innovative strategy for mapping threads from multiple parallel applications onto SMT-based multicore architectures based on different functional unit contention. This method leverages a granular understanding of functional unit contention to guide thread allocation decisions, optimizing resource utilization and striking a balance in resource usage.

Our results show considerable performance enhancements, with a geometric mean increase of 9.8% compared to Linux scheduler. These improvements are primarily credited to mitigating functional unit contention through strategic thread placement, which assigns threads stressing the same functional units to distinct cores. Furthermore, we have introduced a dissimilarity metric indicating a correlation between application similarity and performance degradation when mapped to the same core.

7.1 Future Directions

In terms of future work, we aspire to expand our investigation into the influence of functional unit contention across a more comprehensive array of architectures. Our focus includes state-of-the-art architectures such as Intel Sapphire Rapids and AMD Zen 5 as we strive to deepen our understanding of resource management in multicore processors. In the process, our objective is to formulate a comprehensive set of guidelines that will enable efficient utilization of these and future multicore architectures, allowing us to harness their vast computational capacities without succumbing to resource contention.

Additionally, we envisage the integration of our tool into the Linux kernel, a logical progression given the kernel's role in managing thread placement on processor cores. Incorporating our tool within the Linux kernel's context switches will permit real-time monitoring and management of resource contention. This arrangement could handle resource allocation based on the tool's insights into functional unit contention, thereby diminishing performance impacts. This could yield more refined control and enhanced resource management, culminating in optimized system performance.

By this approach, our tool could continually adapt to the behavior of each application executing on the system, adjusting resources as needed to maximize performance. This would enable the implementation of dynamic, self-adjusting resource management strategies where the kernel continuously learns from and adapts to the system's workload.

In summary, these progressive endeavors will lay the foundation for improved resource management strategies, ultimately driving more efficient and effective utilization of multicore processors. These advances could have far-reaching implications across various domains, including data centers, consumer electronics, and mobile technology.

7.2 Publications

During the development of this thesis, several academic papers were produced, embodying the depth and breadth of the ongoing research. We first list the ones as the first author:

- **SERPA, M. S.**; CRUZ, E. H. M.; DIENER, M.; LORENZON, A. F.; BECK, A. C. S.; NAVAUX, P. O. A. Mitigating functional unit Contention in Parallel Applications using Instruction-Aware Mapping. In: Concurrency and Computation: Practice and Experience, 2022, **Qualis A2**.
- **SERPA, M. S.**; PAVAN, P. J.; CRUZ, E. H. M.; MACHADO, R. L.; PANETTA, J.; AZAMBUJA, A.; CARISSIMI, A. S.; NAVAUX, P. O. A. Energy Efficiency and Portability of Oil and Gas Simulations on Multicore and GPU Architectures. In: Concurrency and Computation: Practice and Experience, 2021, **Qualis A2**.

- **SERPA, M. S.**; CRUZ, E. H. M.; DIENER, M.; BECK, A. C. S.; NAVAUX, P. O. A. Atenuando a Contenção nas Unidades de Execução com Mapeamento Instruction-Aware. In: Symposium on Computer Systems, 2020.
- **SERPA, M. S.**; CRUZ, E. H. M.; DIENER, M.; KRAUSE, A. M.; NAVAUX, P. O. A.; PANETTA, J.; FARRÉS, A.; ROSAS, C.; HANZICH, M. Optimization strategies for geophysics models on manycore systems. In: International Journal of High Performance Computing Applications, 2019, **Qualis B1**.
- **SERPA, M. S.**; MOREIRA, F. B.; NAVAUX, P. O. A.; CRUZ, E. H. M.; DIENER, M.; GRIEBLER, D. J.; FERNANDES, L. G. Memory Performance and Bottlenecks in Multicore and GPU Architectures. In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2019, **Qualis A2**.
- **SERPA, M. S.**; PAVAN, P. J.; PANETTA, J.; AZAMBUJA, A.; CARISSIMI, A. S.; NAVAUX, P. O. A. Portabilidade e Eficiência do Método Fletcher de Aplicações Sísmicas em Arquiteturas Multicore e GPU. In: Symposium on Computer Systems, 2019.
- **SERPA, M. S.**; CRUZ, E. H. M.; PANETTA, J.; AZAMBUJA, A.; CARISSIMI, A. S.; NAVAUX, P. O. A. Improving Oil and Gas Simulation Performance Using Thread and Data Mapping. In: Selected Papers - Symposium on Computer Systems, 2018.
- **SERPA, M. S.**; CRUZ, E. H. M.; PANETTA, J.; AZAMBUJA, A.; NAVAUX, P. O. A. Otimizando uma Aplicação de Geofísica com Mapeamento de Threads e Dados. In: Symposium on Computer Systems, 2018.

In addition to the papers mentioned above, a series of other publications have been produced throughout the development of this thesis, further contributing to the breadth of our research in the high-performance computing area.

- NAVAUX, P. O. A.; LORENZON, A. F.; **SERPA, M. S.** Challenges in High-Performance Computing. In: Journal of the Brazilian Computer Society, 2023, **Qualis B1**.
- SCHUSSLER, B. S.; RIGON, P. H.; **SERPA, M. S.**; KUNAS, C. A.; CARISSIMI, A.; PANETTA, J.; NAVAUX, P. O. A. Comparando o Desempenho entre Computaçao em Nuvem e Servidor Local na Execuçao do Método Fletcher. In: Escola Regional de Alto Desempenho da Região Sul, 2023.

- MARQUES, S. M. V. N.; **SERPA, M. S.**; MUÑOZ, A. N.; ROSSI, F. D.; LUIZELLI, M. C.; NAVAUX, P. O. A.; BECK, A. C. S.; LORENZON, A. F.; Optimizing the EDP of OpenMP applications via concurrency throttling and frequency boosting. In: Journal of Systems Architecture, 2022, **Qualis B1**.
- KUNAS, C. A.; PINTO, D. R.; GRANVILLE, L. Z.; **SERPA, M. S.**; PADOIN, E. L.; NAVAUX, P. O. A. Edge Computing versus Cloud Computing: Impact on Retinal Image Pre-processing. In: International Symposium on Computer Architecture and High Performance Computing, 2022, **Qualis B1**.
- CARNEIRO, A. R.; **SERPA, M. S.**; NAVAUX, P. O. A. Lightweight Deep Learning Applications on AVX-512. In: IEEE Symposium on Computers and Communications, 2021, **Qualis A2**.
- GIRELLI, V. S.; MOREIRA, F. B.; **SERPA, M. S.**; CARASTAN-SANTOS, D.; NAVAUX, P. O. A. Investigating memory prefetcher performance over parallel applications: From real to simulated. In: Journal of Parallel and Distributed Computing, 2021, **Qualis A2**.
- FREYTAG, G.; **SERPA, M. S.**; LIMA, J. V. F.; RECH, P.; NAVAUX, P. O. A. Collaborative execution of fluid flow simulation using non-uniform decomposition on heterogeneous architectures. In: Concurrency and Computation: Practice and Experience, 2021, **Qualis A2**.
- MARQUES, S. M. V. N.; MEDEIROS, T. S.; **SERPA, M. S.**; ROSSI, F. D.; LUIZELLI, M. C.; NAVAUX, P. O. A.; BECK, A. C. S.; LORENZON, A. F.; Optimizing Parallel Applications via Dynamic Concurrency Throttling and Turbo Boosting. In: Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2021, **Qualis A2**.
- BERNED, G. P.; MEDEIROS, T. S.; **SERPA, M. S.**; ROSSI, F. D.; LUIZELLI, M. C.; NAVAUX, P. O. A.; BECK, A. C. S.; LORENZON, A. F. Combining Thread Throttling and Mapping to Optimize the EDP of Parallel Applications. In: Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2021, **Qualis A2**.

  KUNAS, C. A.; **SERPA, M. S.**; HECK, L. P.; PADOIN, E. L.; NAVAUX, P. O. A. Improving Performance of Long Short-Term Memory Networks for Sentiment Analysis Using Multicore and GPU Architectures. In: Latin America High Performance Computing Conference, 2021.

- NAVAUX, P. O. A.; **SERPA, M. S.** Desafios do Processamento de Alto Desempenho. In: Seminário Integrado de Software e Hardware, 2021.
- KUNAS, C. A.; **SERPA, M. S.**; BEZ, J. L.; PADOIN, E. L.; NAVAUX, P. O. A. Offloading the Training of an I/O Access Pattern Detector to the Cloud. In: Workshop on Cloud Computing, 2021.
- LORENZON, A. F.; VARGAS, L.; LIMA, L.; **SERPA, M. S.**; OLIVEIRA, A.; LUIZELLI, M. C.; ROSSI, F.; NAVAUX, P. O. A. Otimizando a correspondência de patches para o inpainting de imagens com diferentes modelos de programação paralela. In: Symposium on Computer Systems, 2020.
- MICHELS, F. D. P.; **SERPA, M. S.**; SANTOS, D. C.; SCHNORR, L. M.; NAVAUX, P. O. A. Otimização de Aplicações Paralelas em Aceleradores Vetoriais NEC SX-Aurora. In: Symposium on Computer Systems, 2020.
- NESI, L. L.; **SERPA, M. S.**; SCHNORR, L. M.; NAVAUX, P. O. A. Task-Based Parallel Strategies for CFD Application in Heterogeneous CPU/GPU Resources. In: Concurrency and Computation: Practice and Experience, 2020, **Qualis A2**.
- GARCIA, A. M.; **SERPA, M. S.**; GRIEBLER, D. J.; SCHEPKE, C.; FERNANDES, L. G.; NAVAUX, P. O. A. The Impact of CPU Frequency Scaling on Computing Infrastructures. In: International Conference on Computational Science and Applications, 2020, **Qualis B1**.
- MOREIRA, F. B.; SCHAAN, B. D.; SCHNEIDERS, J.; REIS, M. A.; **SERPA, M. S.**; NAVAUX, P. O. A. Impacto da Resolução na Detecção de Retinopatia Diabética com uso de Deep Learning. In: Brazilian Symposium on Computing Applied to Health, 2020.
  CAMARGO, M. W.; **SERPA, M. S.**; SANTOS, D. C.; CARISSIMI, A.; NAVAUX, P. O. A. Accelerating Machine Learning Algorithms with TensorFlow Using Thread Mapping Policies. In: Latin America High-Performance Computing Conference, 2020.
- PAVAN, P. J.; BEZ, J. L.; **SERPA, M. S.**; BOITO, F. Z.; NAVAUX, P. O. A. An Unsupervised Learning Approach for I/O Behavior Characterization. In: International Symposium on Computer Architecture and High-Performance Computing, 2019, **Qualis B1**.
- FREYTAG, G.; **SERPA, M. S.**; LIMA, J. V. F.; RECH, P.; NAVAUX, P. O. A. Non-uniform Partitioning for Collaborative Execution on Heterogeneous Architectures.

In: International Symposium on Computer Architecture and High-Performance Computing, 2019, **Qualis B1**.

- GIRELLI, V. S.; MOREIRA, F. B.; **SERPA, M. S.**; NAVAUX, P. O. A. Impacto do Prefetcher na Precisão de Simulações de Arquiteturas Paralelas. In: Symposium on Computer Systems, 2019.
- PADOIN, E. L.; DIEFENTHALER, A. T.; **SERPA, M. S.**; PAVAN, P. J.; CARREÑO, E. D.; NAVAUX, P. O. A.; MÉHAUT, J.;: Optimizing Water Cooling Applications on Shared Memory Systems. In: Latin American Conference on High-Performance Computing, 2019.

REFERENCES

ACEITUNO, J. M. et al. Hardware resources contention-aware scheduling of hard real-time multiprocessor systems. **Journal of Systems Architecture**, Elsevier, v. 118, p. 102223, 2021.

AKTURK, I.; OZTURK, O. Adaptive thread scheduling in chip multiprocessors. **International Journal of Parallel Programming**, Springer, v. 47, n. 1, p. 1–31, 2019.

ALVES, M. A. Z. et al. Sinuca: A validated micro-architecture simulator. In: IEEE. **2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems**. [S.l.], 2015. p. 605–610.

ANDERSON, T. E.; LAZOWSKA, D.; LEVY, H. M. The performance implications of thread management alternatives for shared-memory multiprocessors. **ACM SIGMETRICS Performance Evaluation Review**, ACM New York, NY, USA, v. 17, n. 1, p. 49–60, 1989.

ASANOVIC, K. et al. The landscape of parallel computing research: A view from berkeley. **Technical Report UCB/EECS-2018-2**, 2018.

BAILEY, D. H. Nas parallel benchmarks. **Encyclopedia of Parallel Computing**, Springer, v. 1, n. 1, 2011.

BAKITA, J. et al. Simultaneous multithreading in mixed-criticality real-time systems. In: IEEE. **2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)**. [S.l.], 2021. p. 278–291.

BAUMANN, A. et al. The multi-kernel: a new os architecture for scalable multi-core systems. In: **Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles**. [S.l.: s.n.], 2009. p. 29–44.

BLAGODUROV, S. et al. A case for numa-aware contention management on multi-core systems. In: **Proceedings of the 19th international conference on Parallel Architectures and compilation techniques**. [S.l.: s.n.], 2010. p. 557–558.

BROQUEDIS, F. et al. hwloc: A generic framework for managing hardware affinities in hpc applications. In: IEEE. **2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing**. Pisa, Italy, 2010. p. 180–186.

BROWNE, S. et al. A portable programming interface for performance evaluation on modern processors. **The international journal of high-performance Computing Applications**, Sage Publications Sage CA: Thousand Oaks, CA, v. 14, n. 3, p. 189–204, 2000.

BULPIN, J. R.; PRATT, I. Hyper-threading aware process scheduling heuristics. In: **USENIX Annual Technical Conference, General Track**. [S.l.: s.n.], 2005. p. 399–402.

CAZORLA, F. et al. Dynamically controlled resource allocation in smt processors. In: **37th International Symposium on Microarchitecture (MICRO-37'04)**. [S.l.]: IEEE, 2004.

CHALL, S.; PAUL, K. Cache contention characterization and analysis on smt processors. **Journal of Parallel and Distributed Computing**, Elsevier, v. 147, p. 64–76, 2021.

CHEN, C.-H.; TSAY, R.-S. Analytical process scheduling optimization for heterogeneous multi-core systems. **arXiv preprint arXiv:2109.04605**, 2021.

CIESLAK, R. Dynamic linker tricks: Using ld_preload to cheat, inject features and investigate programs. **Retrieved March**, v. 12, p. 2015, 2015.

CLARK, M. A new x86 core architecture for the next generation of computing. In: **2016 IEEE Hot Chips 28 Symposium (HCS)**. Cupertino, CA, USA: IEEE, 2016. p. 1–19.

CRUZ, E. H. et al. Optimizing memory locality using a locality-aware page table. In: IEEE. **2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing**. [S.l.], 2014. p. 198–205.

CRUZ, E. H. et al. A sharing-aware memory management unit for online mapping in multi-core architectures. In: SPRINGER. **European Conference on Parallel Processing**. [S.l.], 2016. p. 490–501.

DALLY, W. J. Domain-specific architectures for computer vision. In: IEEE. **Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.], 2019. p. 1–12.

DELIMITROU, C.; KOZYRAKIS, C. Paragon: Qos-aware scheduling for heterogeneous data centers. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 48, n. 4, p. 77–88, 2013.

DIAVASTOS, A.; CARLSON, T. E. Efficient instruction scheduling using real-time load delay tracking. **ACM Transactions on Computer Systems**, ACM New York, NY, v. 40, n. 1-4, p. 1–21, 2022.

DONGARRA, J. et al. High-performance computing: clusters, constellations, mpps, and future directions. **Computing in Science & Engineering**, IEEE, v. 7, n. 2, p. 51–59, 2005.

EGGERS, S. J. et al. Simultaneous multithreading: A platform for next-generation processors. **IEEE micro**, IEEE, v. 17, n. 5, p. 12–19, 1997.

EL-MOURSY, A. et al. Compatible phase co-scheduling on a cmp of multi-threaded processors. In: **Proceedings 20th IEEE International Parallel & Distributed Processing Symposium**. [S.l.]: IEEE, 2006.

EYERMAN, S.; EECKHOUT, L. Probabilistic job symbiosis modeling for SMT processor scheduling. In: **Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems - ASPLOS '10**. [S.l.]: ACM Press, 2010.

FEDOROVA, A.; SELTZER, M.; SMITH, M. D. Improving performance isolation on chip multiprocessors via an operating system scheduler. In: **16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)**. [S.l.]: IEEE, 2007.

FELIU, J. et al. Speculative inter-thread store-to-load forwarding in smt architectures. **Journal of Parallel and Distributed Computing**, Elsevier, v. 173, p. 94–106, 2023.

FELIU, J. et al. Understanding cache hierarchy contention in cmps to improve job scheduling. In: **2012 IEEE 26th International Parallel and Distributed Processing Symposium**. Shanghai, China: IEEE, 2012. p. 508–519.

FELIU, J. et al. Bandwidth-aware on-line scheduling in SMT multicores. **IEEE Transactions on Computers**, v. 65, n. 2, 2016.

FELIU, J. et al. Thread isolation to improve symbiotic scheduling on smt multi-core processors. **IEEE Transactions on Parallel and Distributed Systems**, Institute of Electrical and Electronics Engineers (IEEE), v. 31, n. 2, p. 359–373, feb 2020.

FLYNN, M. J.; AKENINE-MöLLER, T.; STRID, O. **Microprocessor systems: an introduction**. [S.l.]: PWS Publishing Company, 1995. ISBN 9780534948221.

GAO, Y. et al. Analyzing thread interaction impact on l3 cache contention in smt processors. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, 2023.

GOMEZ, I. et al. Communication and cache contention on smt processors: Characterization, impact, and mitigation. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM, v. 17, n. 4, p. 1–26, 2020.

GUPTA, P.; PATRA, M. K. Architectural exploration of heterogeneous quantum-classical systems for near-term quantum computing. In: ACM. **Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.], 2021. p. 285–296.

HENNESSY, J. L.; PATTERSON, D. A. Computer architecture: a quantitative approach. **Elsevier**, v. 5, 2011.

HILL, M. D.; MARTY, M. R. Amdahl's law in the multi-core era. **Computer**, IEEE, v. 41, n. 7, p. 33–38, 2008.

JIANG, Y. et al. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In: **Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08**. [S.l.]: ACM Press, 2008.

JOHNSON, M. et al. Papi-v: Performance monitoring for virtual machines. In: ACM. **International Conference on Parallel Processing Workshops**. [S.l.], 2012.

KALLA, R. et al. Power7: Ibm's next-generation server processor. **IEEE micro**, IEEE, v. 30, n. 2, p. 7–15, 2010.

KAMBADUR, M. et al. Measuring interference between live data center applications. In: IEEE. **SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2012. p. 1–12.

KASNECI, E. et al. Chatgpt for good? on opportunities and challenges of large language models for education. **Learning and Individual Differences**, Elsevier, v. 103, p. 102274, 2023.

KASSAMBARA, A. **Practical guide to cluster analysis in R: Unsupervised machine learning**. [S.l.]: Sthda, 2017.

KUNDAN, S.; ANAGNOSTOPOULOS, I. Priority-aware scheduling under shared-resource contention on chip multicore processors. In: IEEE. **2021 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.], 2021. p. 1–5.

KUNDAN, S. et al. A pressure-aware policy for contention minimization on multicore systems. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM New York, NY, v. 19, n. 3, p. 1–26, 2022.

LI, T. et al. Operating system support for overlapping-isa heterogeneous multi-core architectures. In: IEEE. **HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture**. [S.l.], 2010. p. 1–12.

LIN, Y.-T. et al. Characterizing and mitigating instruction cache contention on smt processors. **IEEE Transactions on Computers**, IEEE, 2022.

LIU, J.; CHEN, W. Scalable task scheduling for multi-core systems. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 29, n. 8, p. 1767–1781, 2018.

LIU, J. et al. Efficient cache sharing mechanism for mitigating communication bottleneck on smt processors. In: ACM. **Proceedings of the 2023 ACM International Conference on Supercomputing (ICS)**. [S.l.], 2023.

LORENZON, A. F.; FILHO, A. C. S. B. **Parallel computing hits the power wall: principles, challenges, and a Survey of Solutions**. [S.l.]: Springer Nature, 2019.

LOVE, R. Kernel korner: Cpu affinity. **Linux Journal**, Belltown Media, v. 2003, n. 111, p. 8, 2003.

MUCCI, P. J. et al. Papi: A portable interface to hardware performance counters. In: **Proceedings of the Department of Defense HPCMP Users Group conference**. [S.l.: s.n.], 1999. v. 710.

MÜLLER, M. S. et al. Spec omp2012—an application benchmark suite for parallel systems using openmp. In: SPRINGER. **International Workshop on OpenMP**. [S.l.], 2012. p. 223–236.

NARAYANAN, D. et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.: s.n.], 2021. p. 1–15.

PABLA, C. S. Completely fair scheduler. **Linux Journal**, Belltown Media, v. 2009, n. 184, p. 4, 2009.

PADUA, D. **Encyclopedia of parallel computing**. [S.l.]: Springer Science & Business Media, 2011.

PAN, Z.; ZHAI, W. Exploring cache contention on smt processors in cloud environments. In: IEEE. **Proceedings of the 2021 IEEE International Conference on Cloud Computing (CLOUD)**. [S.l.], 2021. p. 90–97.

PATTERSON, D. Quantum computing: Progress and prospects. **Communications of the ACM**, ACM, v. 61, n. 11, p. 17–19, 2018.

PI, A.; ZHOU, X.; XU, C. Holmes: Smt interference diagnosis and cpu scheduling for job co-location. In: **Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing**. [S.l.: s.n.], 2022. p. 110–121.

RODRIGUEZ, V.; ABELLA, J.; CANAL, R. A comprehensive study of instruction and data cache contention in simultaneous multithreading processors. **Journal of Systems Architecture**, Elsevier, v. 123, p. 102293, 2022.

ROLOFF, E. et al. Exploring instance heterogeneity in public cloud providers for hpc applications. In: SCITEPRESS. **9th International Conference on Cloud Computing and Services Science, CLOSER 2019**. [S.l.], 2019. p. 210–222.

SAEZ, J. C. et al. Leveraging core specialization via os scheduling to improve performance on asymmetric multi-core systems. **ACM Transactions on Computer Systems (TOCS)**, ACM New York, NY, USA, v. 30, n. 2, p. 1–38, 2012.

SERPA, M. S. et al. Mitigating execution unit contention in parallel applications using instruction-aware mapping. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, p. e6819, 2022.

SERPA, M. S. et al. Memory performance and bottlenecks in multi-core and gpu architectures. In: IEEE. **2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)**. [S.l.], 2019. p. 233–236.

SETTLE, A. et al. Architectural support for enhanced smt job scheduling. In: IEEE COMPUTER SOCIETY. **Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques**. Juan-les-pins, France, 2004. p. 63–73.

SHAMEEM, A.; JASON, R. Multi-core programming-increasing performance through software multithreading. **Intel, Hillsboro, OR**, 2005.

SHEN, K. et al. Hardware counter driven on-the-fly request signatures. **ACM SIGARCH Computer Architecture News**, ACM New York, NY, USA, v. 36, n. 1, p. 189–200, 2008.

SHI, T. et al. Alioth: A machine learning based interference-aware performance monitor for multi-tenancy applications in public cloud. **arXiv preprint arXiv:2307.08949**, 2023.

SINGH, T. et al. 3.2 zen: A next-generation high-performance$\times$ 86 core. In: **2017 IEEE International Solid-State Circuits Conference (ISSCC)**. San Francisco, CA, USA: IEEE, 2017. p. 52–53.

SNAVELY, A.; TULLSEN, D. M. Symbiotic job scheduling for a simultaneous multithreaded processor. In: **Proceedings of the ninth international conference on Architectural Support for programming languages and operating systems - ASPLOS-IX**. [S.l.]: ACM Press, 2000.

SNAVELY, A.; WOLTER, N.; CARRINGTON, L. Modeling application performance by convolving machine signatures with application profiles. In: IEEE. **Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)**. [S.l.], 2001. p. 149–156.

TAM, D.; AZIMI, R.; STUMM, M. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. **ACM SIGOPS Operating Systems Review**, ACM New York, NY, USA, v. 41, n. 3, p. 47–58, 2007.

TAM, D.; AZIMI, R.; STUMM, M. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In: **Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 - EuroSys**. [S.l.]: ACM Press, 2007.

TAM, D. K. et al. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 44, n. 3, p. 121–132, 2009.

TAN, C.; NADEAU, T. P.; GAO, G. R. Multi-core and many-core processors: Theory, architectures, algorithms, and applications. In: ACM. **Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing**. [S.l.], 2019. p. 197–199.

TERPSTRA, D. et al. Collecting performance data with papi-c. In: **Tools for High Performance Computing 2009**. [S.l.]: Springer, 2010.

TIAN, K.; JIANG, Y.; SHEN, X. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In: **Proceedings of the 6th ACM conference on Computing frontiers - CF '09**. [S.l.]: ACM Press, 2009.

TSAI, P.-C.; HSU, C.-H.; LIN, K.-C. Many-core computing: Algorithms, architectures, and applications. **IEEE Transactions on Computers**, IEEE, v. 69, n. 2, p. 152–157, 2020.

TULLSEN, D. M. et al. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. **ACM SIGARCH Computer Architecture News**, ACM, v. 24, n. 2, p. 191–202, 1996.

TULLSEN, D. M.; EGGERS, S. J.; LEVY, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In: ACM. **Proceedings of the 22nd Annual International Symposium on Computer Architecture**. [S.l.], 1995. (ISCA ’95).

VENKATESH, H. K.; PATRA, M. K. Power-efficient cache design for heterogeneous quantum-classical computing systems. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 30, n. 1, p. 113–127, 2022.

WANG, L. et al. Scalability challenges in cloud computing. In: SPRINGER. **Proceedings of the International Conference on Cloud Computing**. [S.l.], 2019. p. 71–82.

WANG, L.; YIN, C.; LI, W. Characterizing and mitigating l2 cache contention in smt processors. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, 2021.

WANG, Y. et al. An in-depth analysis of system-level techniques for simultaneous multithreaded processors in clouds. In: **Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications**. [S.l.: s.n.], 2020. p. 145–149.

WEAVER, V. M. et al. Measuring energy and power with papi. In: IEEE. **2012 41st international conference on parallel processing workshops**. Pittsburgh, PA, USA, 2012. p. 262–268.

XU, Y. et al. Artificial intelligence: A powerful paradigm for scientific research. **The Innovation**, Elsevier, v. 2, n. 4, p. 100179, 2021.

YIN, C.; LI, W. Mitigating data cache contention on smt processors using instruction scheduling techniques. In: IEEE. **Proceedings of the 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.], 2023.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing resource management: A survey. **IEEE Transactions on Services Computing**, IEEE, v. 11, n. 1, p. 42–59, 2018.

ZHAO, S. et al. Cache-aware allocation of parallel jobs on multi-cores based on learned recency. In: **Proceedings of the 31st International Conference on Real-Time Networks and Systems**. [S.l.: s.n.], 2023. p. 177–187.

ZHOU, X.; HU, W.; XIONG, J. Characterizing and mitigating communication bottleneck in parallel graph processing on smt processors. In: IEEE. **Proceedings of the 2020 IEEE International Symposium on Workload Characterization (IISWC)**. [S.l.], 2020. p. 58–68.

APPENDIX A — ADDITIONAL RESULTS

This appendix presents additional results.

A.1 SPEC and NPB Benchmark Results on AMD Processors

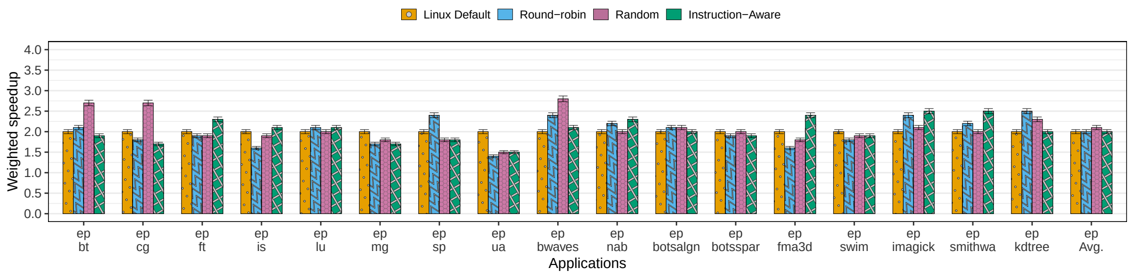


Figure A.1 – Weighted speedup (higher values are better) for **ep** application co-running with diverse applications on the AMD Processor.

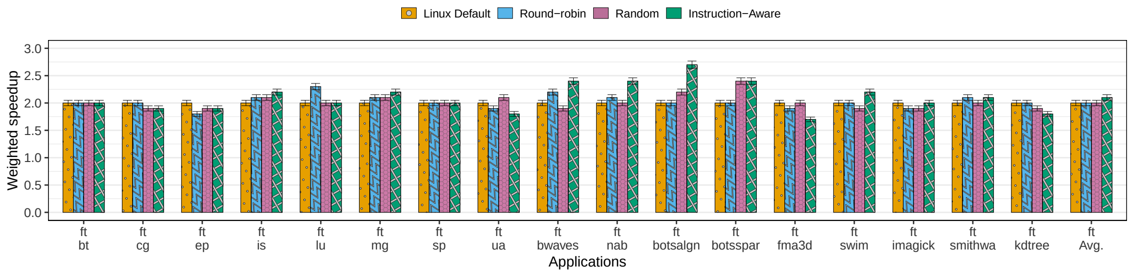


Figure A.2 – Weighted speedup (higher values are better) for **ft** application co-running with diverse applications on the AMD Processor.

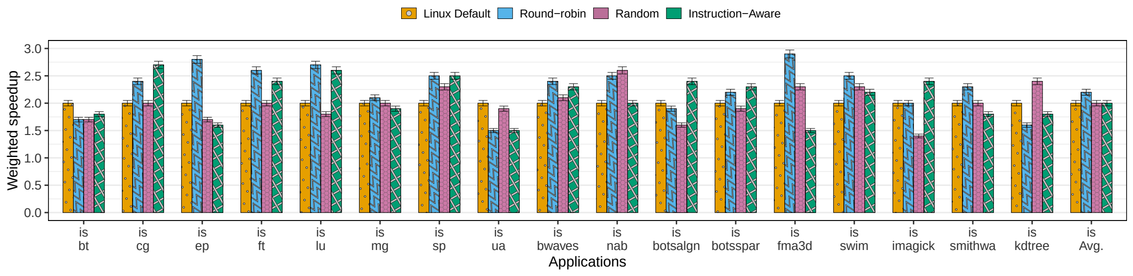


Figure A.3 – Weighted speedup (higher values are better) for **is** application co-running with diverse applications on the AMD Processor.

A.2 SPEC and NPB Benchmark Results on Intel Processors

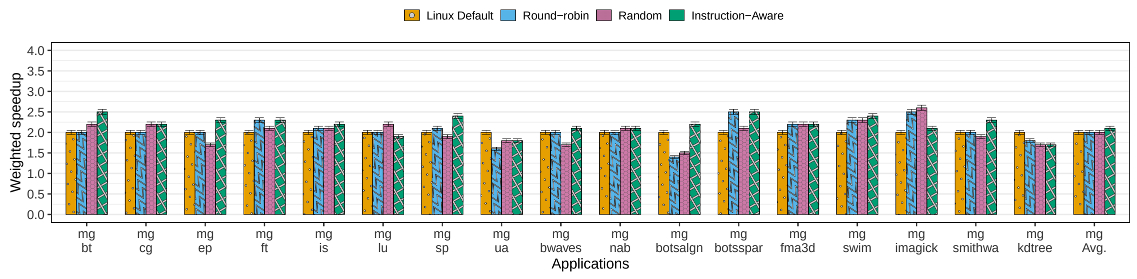


Figure A.4 – Weighted speedup (higher values are better) for **mg** application co-running with diverse applications on the AMD Processor.

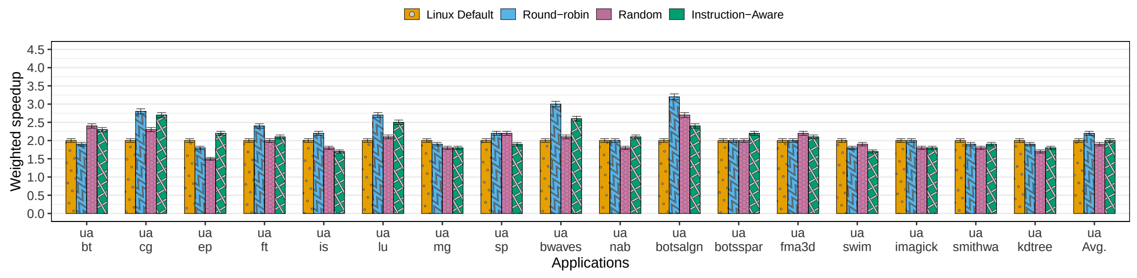


Figure A.5 – Weighted speedup (higher values are better) for **ua** application co-running with diverse applications on the AMD Processor.

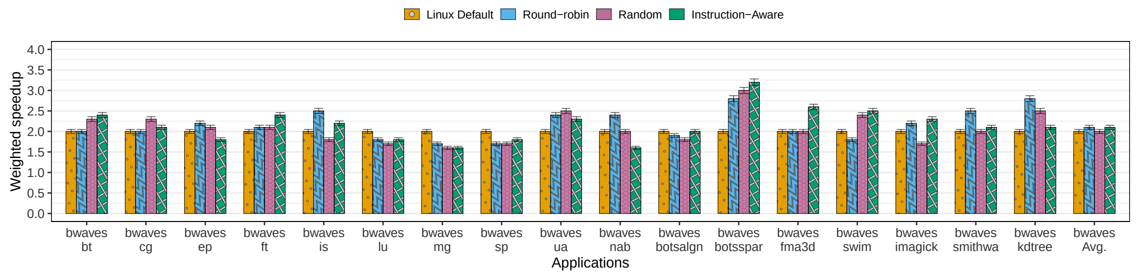


Figure A.6 – Weighted speedup (higher values are better) for **bwaves** application co-running with diverse applications on the AMD Processor.

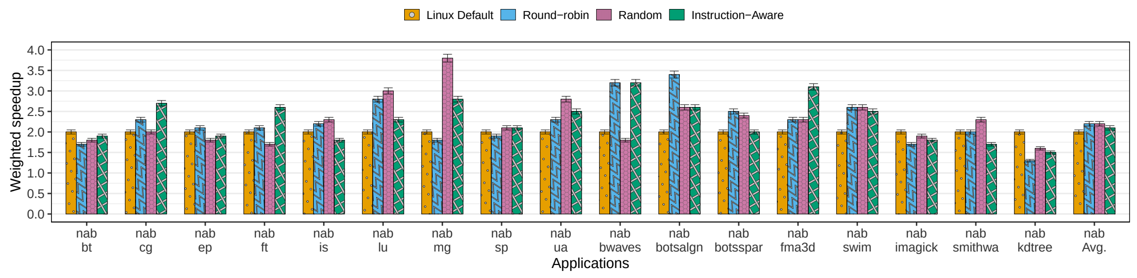


Figure A.7 – Weighted speedup (higher values are better) for **nab** application co-running with diverse applications on the AMD Processor.

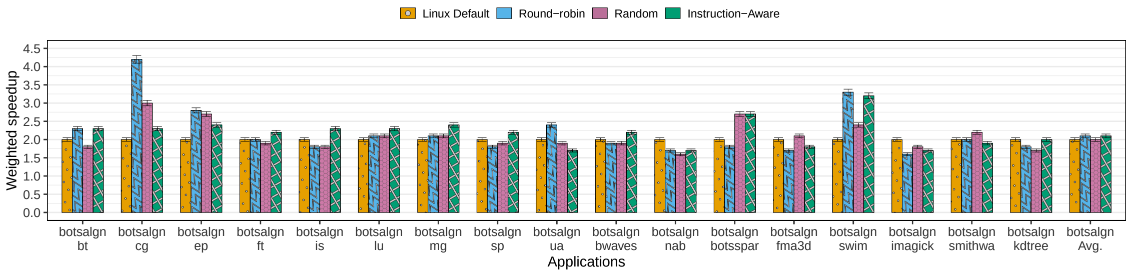


Figure A.8 – Weighted speedup (higher values are better) for **botsalgn** application co-running with diverse applications on the AMD Processor.

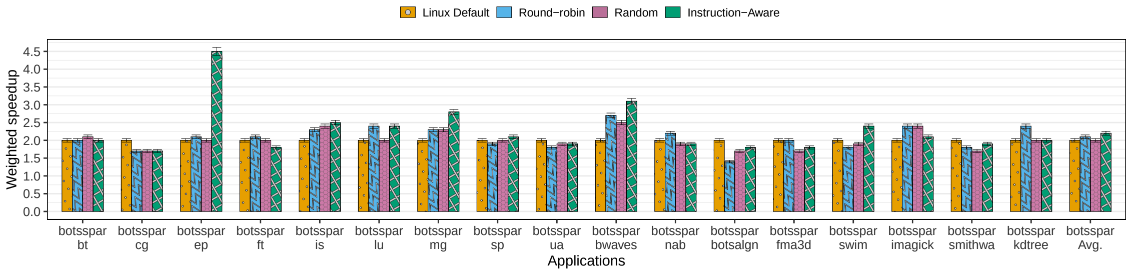


Figure A.9 – Weighted speedup (higher values are better) for **botsspar** application co-running with diverse applications on the AMD Processor.

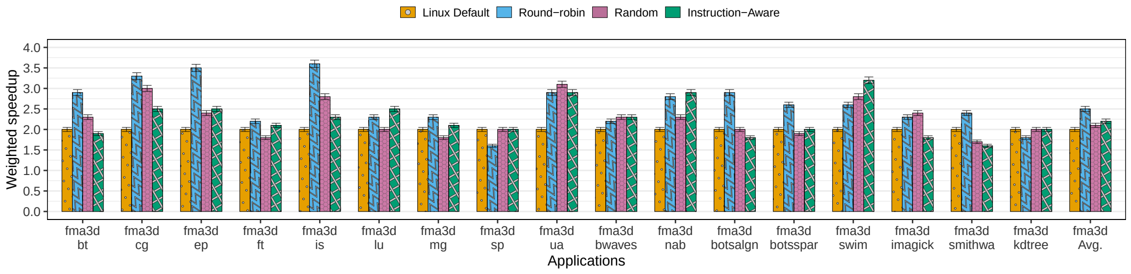


Figure A.10 – Weighted speedup (higher values are better) for **fma3d** application co-running with diverse applications on the AMD Processor.

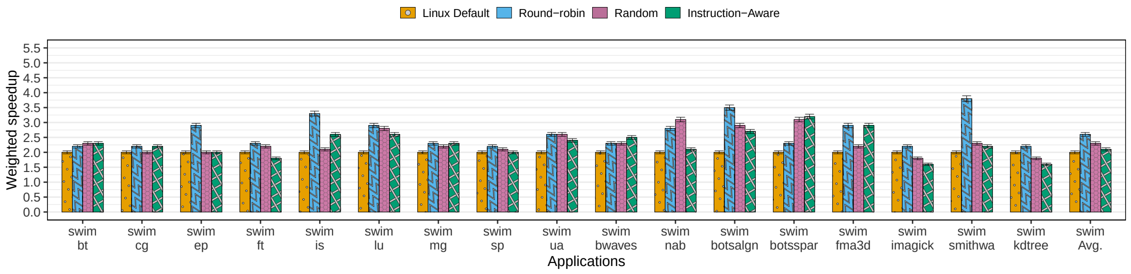


Figure A.11 – Weighted speedup (higher values are better) for **swim** application co-running with diverse applications on the AMD Processor.

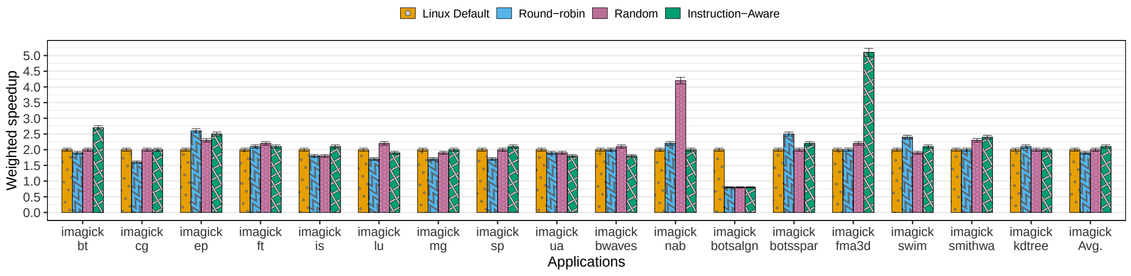


Figure A.12 – Weighted speedup (higher values are better) for **imagick** application co-running with diverse applications on the AMD Processor.

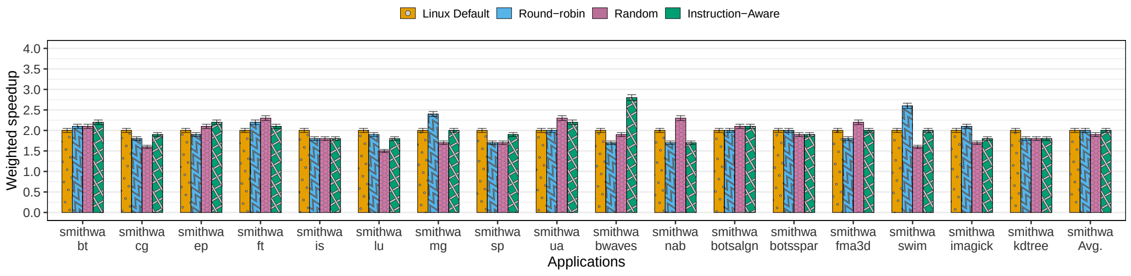


Figure A.13 – Weighted speedup (higher values are better) for **smithwa** application co-running with diverse applications on the AMD Processor.

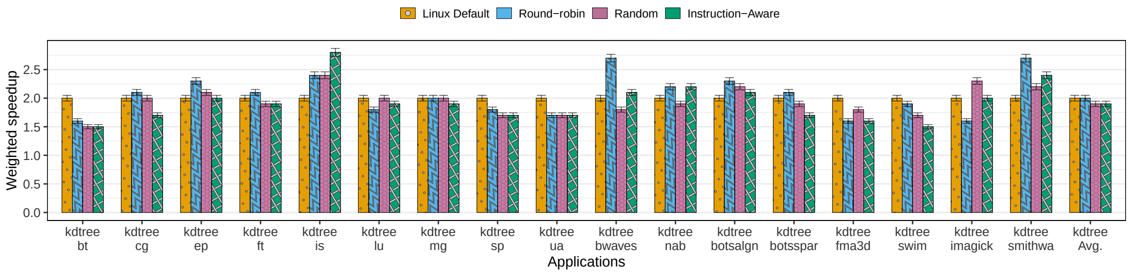


Figure A.14 – Weighted speedup (higher values are better) for **kdtree** application co-running with diverse applications on the AMD Processor.

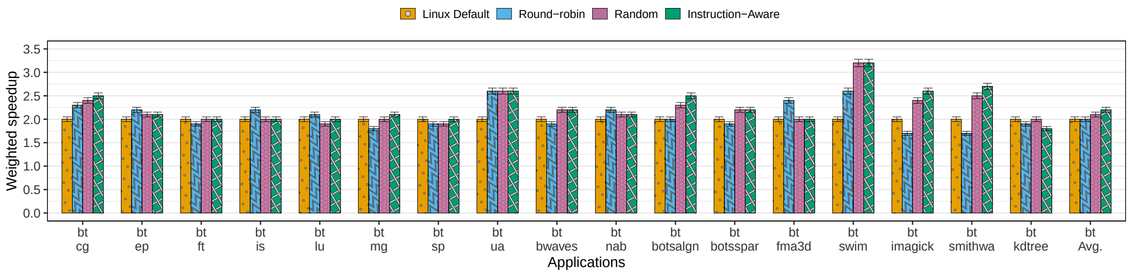


Figure A.15 – Weighted speedup (higher values are better) for **bt** application co-running with diverse applications on the Intel Processor.

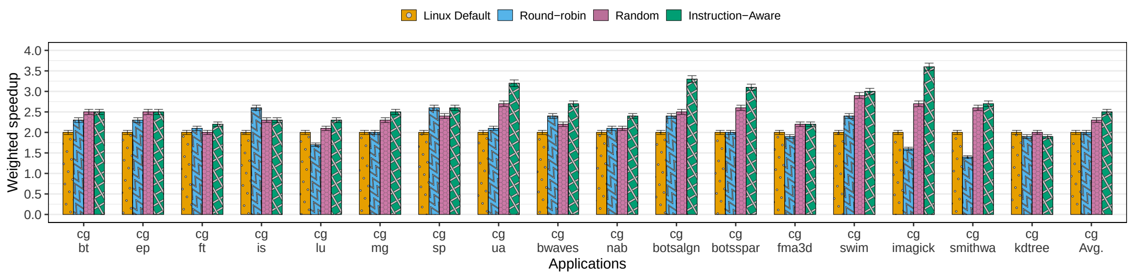


Figure A.16 – Weighted speedup (higher values are better) for **cg** application co-running with diverse applications on the Intel Processor.

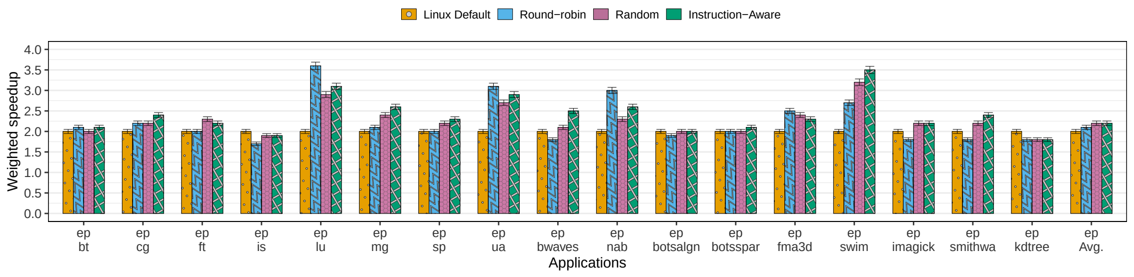


Figure A.17 – Weighted speedup (higher values are better) for **ep** application co-running with diverse applications on the Intel Processor.

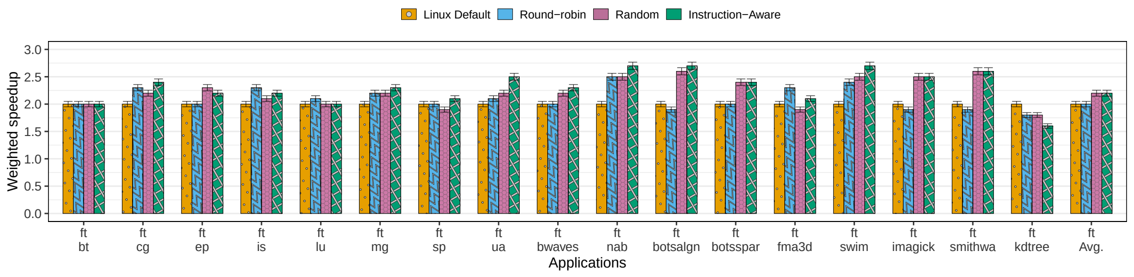


Figure A.18 – Weighted speedup (higher values are better) for **ft** application co-running with diverse applications on the Intel Processor.

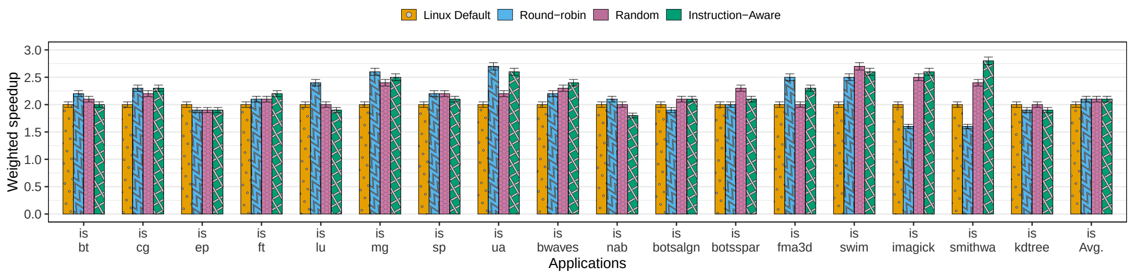


Figure A.19 – Weighted speedup (higher values are better) for **is** application co-running with diverse applications on the Intel Processor.

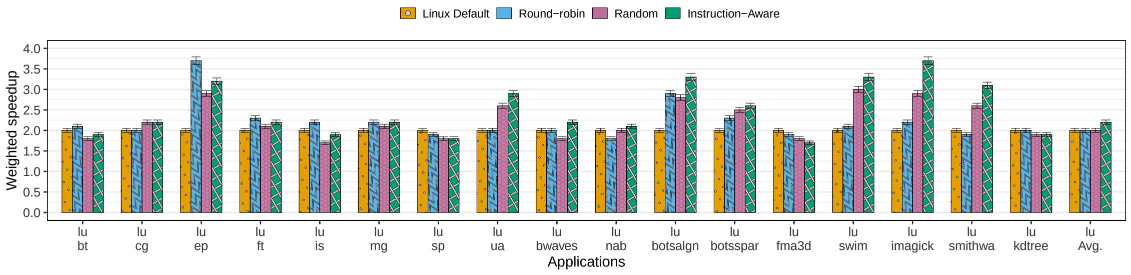


Figure A.20 – Weighted speedup (higher values are better) for **lu** application co-running with diverse applications on the Intel Processor.

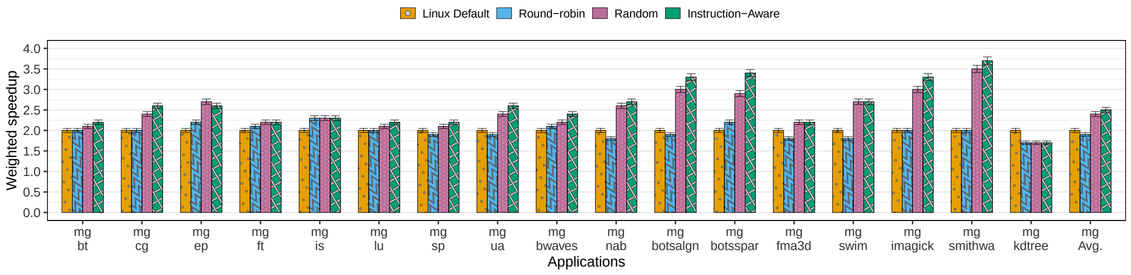


Figure A.21 – Weighted speedup (higher values are better) for **mg** application co-running with diverse applications on the Intel Processor.

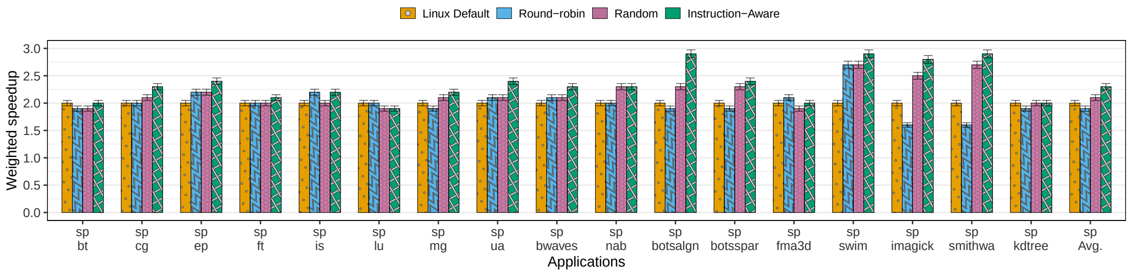


Figure A.22 – Weighted speedup (higher values are better) for **sp** application co-running with diverse applications on the Intel Processor.

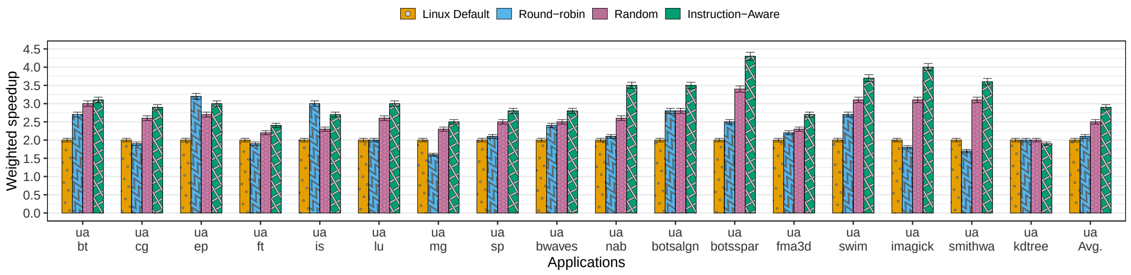


Figure A.23 – Weighted speedup (higher values are better) for **ua** application co-running with diverse applications on the Intel Processor.

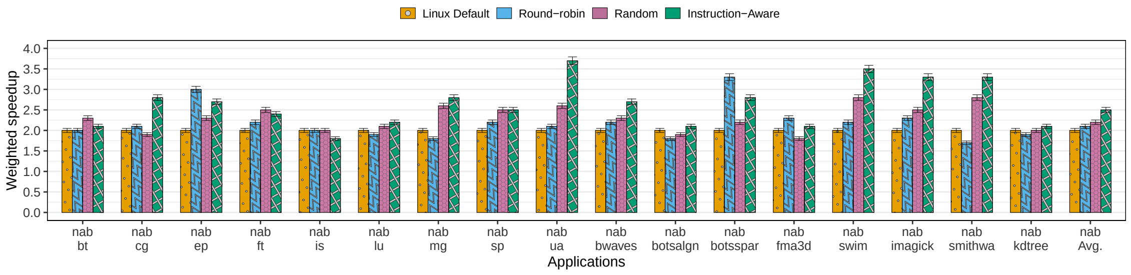


Figure A.24 – Weighted speedup (higher values are better) for **nab** application co-running with diverse applications on the Intel Processor.

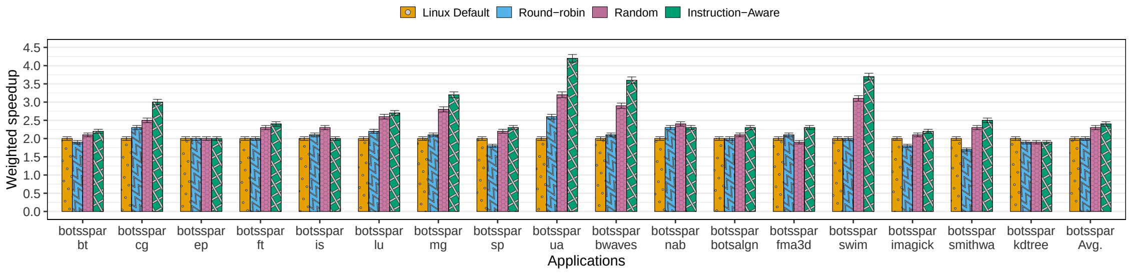


Figure A.25 – Weighted speedup (higher values are better) for **botsspar** application co-running with diverse applications on the Intel Processor.

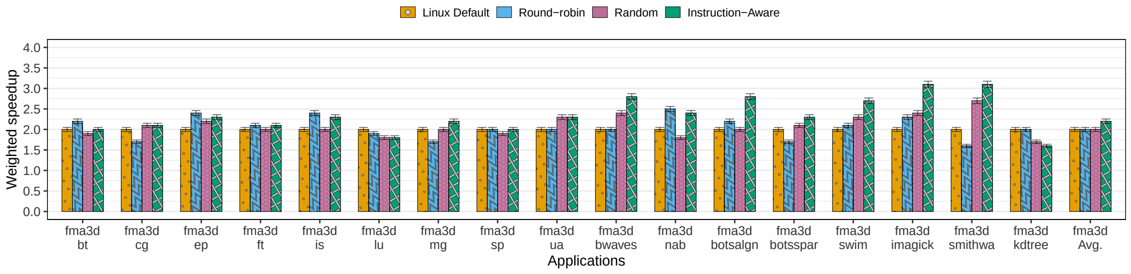


Figure A.26 – Weighted speedup (higher values are better) for **fma3d** application co-running with diverse applications on the Intel Processor.

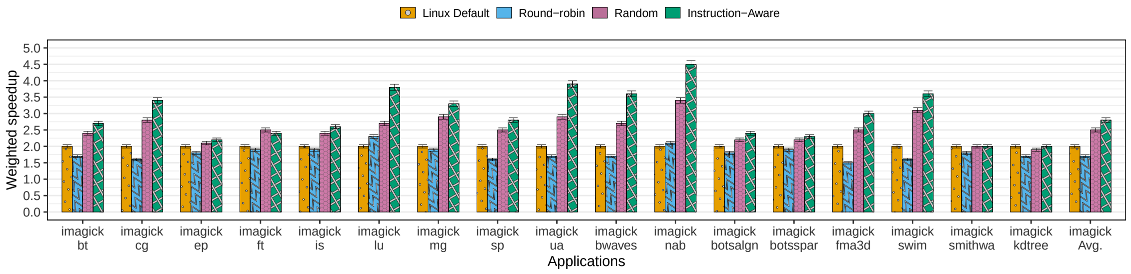


Figure A.27 – Weighted speedup (higher values are better) for **imagick** application co-running with diverse applications on the Intel Processor.

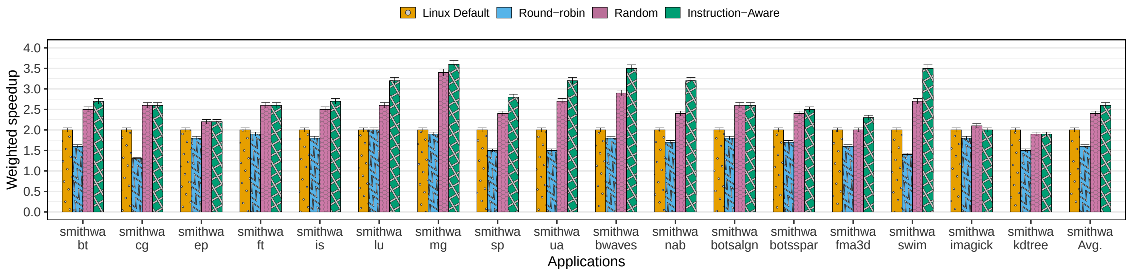


Figure A.28 – Weighted speedup (higher values are better) for **smithwa** application co-running with diverse applications on the Intel Processor.

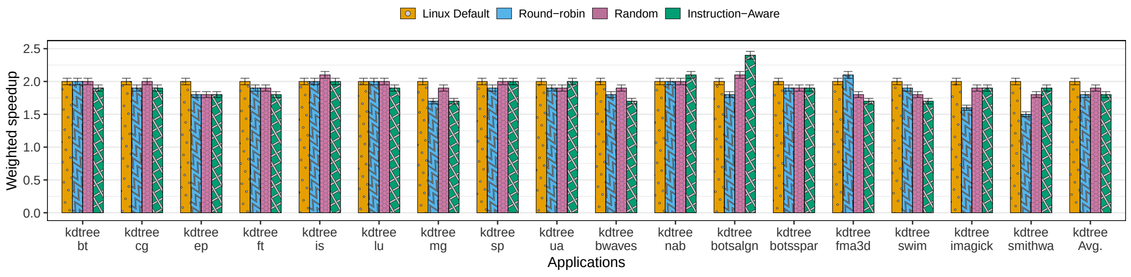


Figure A.29 – Weighted speedup (higher values are better) for **kdtree** application co-running with diverse applications on the Intel Processor.

APPENDIX B — RESUMO EM PORTUGUÊS

This chapter presents a thesis summary in Portuguese, as required by the PPGC Graduate Program in Computing.

Neste capítulo, é apresentado um resumo desta tese na língua portuguesa, como requerido pelo Programa de Pós-Graduação em Computação.

B.1 Introdução

A proliferação de sistemas multicore e SMT tem sido fundamental para moldar os avanços em vários campos. A evolução das arquiteturas de computadores aumentou significativamente o poder computacional, facilitando a resolução de problemas cada vez mais complexos em domínios de aplicação modernos, como inteligência artificial, ciência de dados, big data, bioinformática, computação quântica, segurança cibernética e grandes modelos de linguagem (BAKITA et al., 2021; XU et al., 2021; NARAYANAN et al., 2021; KASNECI et al., 2023).

No ambiente tecnológico atual, muitas aplicações operam em plataformas de computação compartilhadas. A escalabilidade das arquiteturas subjacentes, portanto, se torna primordial. O advento dos sistemas multicore promove a execução paralela de várias aplicações em um ambiente computacional compartilhado, reforçando a escalabilidade, a utilização de recursos e a eficiência de custos (LIU; CHEN, 2018; TAN; NADEAU; GAO, 2019; ROLOFF et al., 2019; TSAI; HSU; LIN, 2020). No entanto, o compartilhamento de recursos computacionais introduz desafios adicionais, exigindo uma gestão eficaz dos recursos para evitar a degradação do desempenho à medida que o número de aplicações em execução simultânea aumenta (ZHANG; CHENG; BOUTABA, 2018; WANG et al., 2019).

Essa gestão de escalabilidade está intrinsecamente ligada ao conceito de utilização. A transição para as arquiteturas multicore e SMT visa otimizar a utilização dos recursos, melhorando o desempenho geral do sistema. No entanto, esses sistemas podem enfrentar um uso subótimo sem uma gestão cuidadosa e uma alocação inteligente de tarefas, o que pode levar a um subaproveitamento significativo dos recursos e gargalos de desempenho (PATTERSON, 2018; ASANOVIC et al., 2018; DALLY, 2019; GUPTA; PATRA, 2021; VENKATESH; PATRA, 2022).

Embora uma abordagem holística para a gestão de recursos em arquiteturas multicore e SMT seja essencial, este trabalho concentra-se no mapeamento de threads em relação às unidades funcionais. Consequentemente, não aprofundaremos técnicas de mapeamento de dados, memória, cache e recursos similares, focando em compreender e resolver problemas de contenção ligados às unidades funcionais.

Além disso, tais ambientes frequentemente lidam com problemas de contenção de unidades funcionais. O SMT facilita a emissão simultânea de instruções de várias threads independentes para várias unidades funcionais, amplificando significativamente a utilização de recursos e o desempenho geral (KALLA et al., 2010; LORENZON; FILHO, 2019; TULLSEN; EGGERS; LEVY, 1995; WANG et al., 2020; FELIU et al., 2023). O principal objetivo do SMT de melhorar a utilização dos recursos de hardware pode, paradoxalmente, levar à degradação do desempenho devido à contenção por recursos compartilhados. Este problema é especialmente pronunciado no contexto das unidades funcionais que lidam com operações de dados. Essas unidades podem se tornar gargalos quando várias threads disputam seu uso simultaneamente, levando à competição por recursos.

Para enfrentar esses desafios, é necessário implementar estratégias eficazes de mapeamento de threads para núcleos. Idealmente, as threads que utilizam fortemente as mesmas unidades funcionais devem ser distribuídas para diferentes núcleos, minimizando assim a contenção e otimizando o uso dos recursos do núcleo. No entanto, determinar o mapeamento ótimo pode ser intrincado e computacionalmente exigente, necessitando de uma compreensão matizada do comportamento da aplicação e da arquitetura subjacente. Essa complexidade ressalta a urgência do desenvolvimento de técnicas automatizadas de mapeamento de threads.

Pesquisas anteriores identificaram a comunicação e a contenção da memória cache em processadores SMT como gargalos de desempenho significativos (CRUZ et al., 2014; FELIU et al., 2016; AKTURK; OZTURK, 2019; SERPA et al., 2019; GOMEZ et al., 2020; ZHOU; HU; XIONG, 2020; CHALL; PAUL, 2021; PAN; ZHAI, 2021; WANG; YIN; LI, 2021; RODRIGUEZ; ABELLA; CANAL, 2022; LIN et al., 2022; GAO et al., 2023; YIN; LI, 2023; LIU et al., 2023). Estratégias para aliviar tais gargalos foram propostas e validadas por pesquisadores, focando principalmente em várias cargas de trabalho multiprograma de thread única, alcançando assim melhorias de desempenho mensuráveis. Esses estudos enfatizam a necessidade de lidar com a contenção de unidades funcionais, especialmente ao considerar aplicações de processamento paralelo. A contenção de unidades funcionais ocorre quando threads da mesma ou de diferentes aplicações emitem tipos semelhantes de instruções (como operações aritméticas, acesso à memória ou operações de ponto flutuante) que utilizam as mesmas unidades funcionais.

- Estudar o comportamento de execução de threads e a mistura de instruções associada a um núcleo pode fornecer insights sobre possíveis contendas para unidades funcionais.
- Ao mapear threads para núcleos, considerando a mistura de instruções, é possível reduzir a contenção de unidades funcionais e melhorar o desempenho geral do sistema.

Dadas essas hipóteses, o objetivo principal desta tese é desenvolver mecanismos para mitigar a contenção de unidades funcionais em plataformas de computação. Pretendemos atingir esse objetivo através dos seguintes passos:

- Propomos uma ferramenta de Mapeamento Consciente de Instrução (IAM) que pode mapear eficientemente threads para núcleos considerando suas misturas de instruções. Esta ferramenta minimiza a contenção e maximiza o desempenho do sistema em ambientes multicore e SMT.
- Para avaliar a eficácia da ferramenta IAM, utilizamos diferentes benchmarks e sistemas de computação em nossos estudos experimentais.
- Também comparamos o IAM com outras estratégias existentes para mapeamento de thread para núcleo, demonstrando suas forças únicas e potencial para melhorar o desempenho do sistema em vários cenários de computação.

Por favor, note que, enquanto o campo mais amplo de otimização de SMT muitas vezes envolve considerações como memória ou cache, esta tese visa especificamente aliviar a contenção ao nível das unidades funcionais. Consequentemente, nossos objetivos e hipóteses estão confinados a este escopo mais estreito.

Em conclusão, o objetivo principal desta tese é abordar a questão da contenção de unidades funcionais e projetar estratégias que melhoram o desempenho geral e levam à utilização eficiente de recursos. O overhead do nosso mecanismo proposto também será avaliado para entender seus benefícios e potenciais limitações. Os estudos pretendem contribuir significativamente para a compreensão da contenção de unidades funcionais e para o desenvolvimento de estratégias eficazes de mitigação.

Esta tese propõe uma solução inovadora para os problemas mencionados anteriormente - a ferramenta de Mapeamento Consciente de Instrução (IAM). O IAM é uma ferramenta online que aproveita as informações a nível de instrução para otimizar o mapeamento de múltiplas aplicações paralelas em núcleos. O núcleo da ferramenta IAM reside em sua capacidade de entender as características das unidades funcionais da carga de trabalho em tempo real. Ele lê dinamicamente contadores de desempenho de hardware para avaliar os padrões de uso de instruções, como o número de operações de ponto flutuante, inteiro, ramificações, cargas e armazenamentos. Essa informação permite que o IAM mapeie de forma inteligente threads que sobrecarregam unidades funcionais idênticas em núcleos diferentes.

A ferramenta IAM visa otimizar o mapeamento de thread para núcleo de forma que as unidades funcionais sejam utilizadas ao seu máximo potencial sem causar contenção. Isso é conseguido minimizando o número de threads que emitem simultaneamente instruções semelhantes à mesma unidade funcional. Ao fazer isso, o IAM melhora o desempenho geral dos processadores SMT e garante o uso eficiente dos recursos computacionais.

As contribuições desta pesquisa são as seguintes:

- Desenvolvemos e introduzimos o SMT-Bench, um microbenchmark projetado para sobrecarregar unidades funcionais específicas. Este benchmark nos permite avaliar empiricamente o impacto do compartilhamento de recursos. Ele fornece insights cruciais sobre o desempenho e comportamento dos processadores SMT sob várias cargas de trabalho. Complementamos isso com avaliações usando dois benchmarks amplamente reconhecidos, o NPB e o SPEC, oferecendo uma análise de desempenho mais abrangente em diversas cargas de trabalho.
- Propomos a IAM (SERPA et al., 2022), uma ferramenta dinâmica, em tempo real, consciente de instruções, para mapear threads de várias aplicações paralelas em núcleos. A ferramenta aproveita os padrões de instrução distintos dessas aplicações, permitindo uma estratégia de mapeamento adaptativo que responde às características da carga de trabalho em mudança à medida que se desdobram.
- Nossos resultados experimentais, derivados de testes realizados em processadores AMD e Intel, demonstram os aprimoramentos de desempenho alcançáveis através da nossa ferramenta de Mapeamento Consciente de Instrução. A ferramenta IAM superou consistentemente o agendador nativo, uma implementação round-robin e uma abordagem de mapeamento aleatório em todos os testes. Ele rendeu uma melhoria média geométrica de desempenho de 9,8% em relação ao agendador nativo do sistema operacional. Esses resultados sublinham a capacidade da nossa fer-

ramenta de aumentar significativamente a eficiência e o desempenho dos processadores SMT.

B.2 Referencial Teórico

Um impulso constante para aprimorar o desempenho e a eficiência impulsiona a computação moderna. Isso catalisou o desenvolvimento de arquiteturas e tecnologias de computação complexas. Este capítulo apresenta esses conceitos-chave e lança luz sobre suas complexidades.

Nos concentramos nas arquiteturas multicore, uma faceta essencial da computação contemporânea. Esta discussão esclarece o efeito transformador de ter vários núcleos dentro de uma única unidade de processamento, possibilitando processamento paralelo e aumentando significativamente a velocidade e a potência de computação.

A conversa passa para SMT, que melhora a eficiência do processador. Esta abordagem permite que múltiplas threads de execução independentes usem mais efetivamente os recursos fornecidos pelas arquiteturas de processadores de hoje. SMT introduz desafios e complexidades, notavelmente em termos de unidades funcionais e conflito de recursos, tópicos nos quais nos aprofundamos em detalhes.

Posteriormente, dissecamos o conceito de unidades funcionais, destacando seu papel e o persistente problema de conflito de recursos. Esta exploração sublinha como estas unidades podem se tornar gargalos, principalmente quando várias threads competem pelos mesmos recursos.

A parte posterior deste capítulo desloca o foco para estratégias de mapeamento de thread para núcleo. Esta discussão enfatiza a importância de um mapeamento efetivo ao aproveitar todo o potencial de arquiteturas multicore e multithread, sublinhando o papel destas estratégias na otimização da utilização do núcleo e no gerenciamento eficiente das threads.

Para concluir, destacamos a criticidade dos contadores de desempenho de hardware e ferramentas essenciais na medição e diagnóstico de desempenho. Mergulhamos em seu papel ao oferecer insights em tempo real das operações do processador, identificando gargalos de desempenho e auxiliando no desenvolvimento e otimização de estratégias de gerenciamento de recursos.

Ao lançar luz sobre esses conceitos fundamentais, este capítulo estabelece as bases para discussões futuras sobre as intrincadas questões de conflito de unidade funcional e exploração de nossa solução proposta. Esta jornada é projetada para dar ao leitor uma compreensão profunda da paisagem de computação moderna e soluções potenciais para o conflito de recursos.

B.3 Trabalhos Relacionados

O conflito de recursos em processadores SMT há muito é reconhecido como um contribuinte significativo para gargalos de desempenho. A literatura neste campo é rica em estratégias e metodologias variadas para superar esses desafios. Este capítulo tem como objetivo fazer um levantamento abrangente desses trabalhos, englobando diferentes aspectos do problema e das soluções propostas.

Consideramos uma classificação multifacetada para esses estudos a fim de realizar uma exploração significativa do assunto. Mergulhamos nas nuances de como essas soluções operam, se elas se concentram em aplicações multithread ou single-thread, pois essa distinção pode influenciar significativamente sua eficácia. Reconhecendo que o contexto de operação desempenha um papel crítico, também examinamos se esses métodos são projetados para ambientes reais ou simulados.

Uma consideração notável é a necessidade de modificações de hardware em várias dessas soluções. Embora algumas abordagens baseadas em hardware ofereçam resultados promissores, a necessidade de mudanças apresenta desafios em termos de praticidade e viabilidade. Torna-se ainda mais complexo quando essas alterações são específicas para certas arquiteturas, limitando a universalidade da solução.

Além disso, avaliamos as soluções com base em sua dependência da arquitetura. Embora eficazes em ambientes específicos, soluções dependentes de arquitetura podem necessitar de mais versatilidade em paisagens arquitetônicas diversas. Assim, soluções independentes de arquitetura que mantêm a eficácia em várias plataformas têm uma vantagem distinta em aplicação ampla.

Além disso, o momento de aplicação dessas soluções é outro fator crítico. Escrutinamos se esses métodos são projetados para execução online, utilizados em tempo real durante a execução do processo, ou para execução offline, implementados quando o sistema não está executando tarefas ativamente.

Organizamos nossa discussão em duas seções principais para fornecer um entendimento abrangente. A primeira seção se concentra em estratégias de soluções baseadas em software que não requerem modificações de hardware e operam principalmente por meio de melhorias de software ou algoritmos. A segunda seção é dedicada a soluções baseadas em hardware que exigem alterações ou melhorias no próprio hardware. Essa dicotomia nos permite apresentar uma visão equilibrada do campo, demonstrando as forças e fraquezas inerentes a cada abordagem.

Por meio desta exploração abrangente, pretendemos fornecer aos leitores uma compreensão completa do cenário atual de soluções para conflitos de recursos em processadores SMT. Este entendimento pode ser a base para o desenvolvimento de novas estratégias inovadoras para melhorar o desempenho e a eficiência dos processadores SMT.

B.4 SMT-Bench

Este capítulo aprofunda-se nas implicações de desempenho do compartilhamento de diversos recursos em processadores SMT. O objetivo principal é elucidar como o compartilhamento de recursos e a disputa podem impactar significativamente o desempenho geral e apresentar razões convincentes para o mapeamento estratégico de threads para mitigar tais problemas.

Após a análise da disputa da unidade funcional, focamos em um microbenchmark especializado que desenvolvemos. Este microbenchmark é especificamente projetado para estressar unidades funcionais distintas do processador. Ao usar esta ferramenta, podemos demonstrar que o tipo de operações realizadas por cada thread tem uma influência profunda tanto no desempenho quanto nos níveis de disputa. Ele oferece uma visão sobre a utilização da unidade funcional e a dinâmica da disputa em um contexto SMT.

Posteriormente, apresentamos evidências convincentes para ilustrar que mapear threads que utilizam os mesmos recursos em um único núcleo pode levar a uma disputa significativa. Tal disputa pode efetivamente diminuir o desempenho, potencialmente negando os benefícios oferecidos pelo SMT. No entanto, argumentamos ainda que estratégias inteligentes de mapeamento podem mitigar esses problemas de desempenho.

Este capítulo enfatiza a importância do mapeamento de threads consciente de recursos em processadores SMT, revelando a conexão intrínseca entre o tipo de operações executadas por threads, o compartilhamento de recursos e a disputa resultante. Além disso, demonstra os ganhos de desempenho potenciais que podem ser alcançados gerenciando efetivamente a disputa da unidade funcional.

B.5 IAM - Mapeamento Consciente de Instrução

Nossa ferramenta proposta para mapeamento online opera em cinco etapas, conforme ilustrado na Figura 5.1. O processo é iniciado com a execução do aplicativo. À medida que o aplicativo começa a rodar, a ferramenta inicia coletando informações sobre a topologia atual da máquina. Isso inclui o número de núcleos disponíveis, sua disposição e o nível de SMT que cada núcleo suporta, entre outros detalhes arquitetônicos.

Na terceira etapa, a ferramenta aproveita os contadores de hardware fornecidos pelo PAPI para detectar os padrões de instrução dos aplicativos. Este processo de detecção permite que a ferramenta distinga os tipos de instruções que cada thread está emitindo, o que é crucial para otimizar o mapeamento de thread para núcleo.

Após a detecção do padrão de instrução, a ferramenta calcula o mapeamento de threads ideal. Este cálculo leva em consideração a topologia da máquina e os padrões de instrução detectados anteriormente. O objetivo aqui é minimizar a disputa da unidade funcional ao atribuir de maneira inteligente threads aos núcleos de uma forma que leve em conta os tipos específicos de instruções que as threads estão executando.

À medida que o aplicativo continua sua execução, a ferramenta implementa o mapeamento de thread para núcleo calculado. Esta etapa de mapeamento garante que cada thread seja executada no núcleo que foi determinado como ótimo para seu tipo específico de instrução.

A partir da terceira etapa, todo esse processo é repetido online, permitindo que a ferramenta se adapte a mudanças no comportamento do aplicativo ou no estado do sistema. O loop continua até a execução do aplicativo ser concluída.

B.6 Resultados Experimentais

Este capítulo apresenta uma análise de desempenho exaustiva de nossa ferramenta proposta para mitigar a competição de unidades funcionais. A investigação incorpora um conjunto diversificado de benchmarks que incluem SMT-Bench, NPB Benchmark e SPEC Benchmark. Esses benchmarks, cobrindo uma ampla gama de cenários de carga de trabalho, foram escolhidos para garantir que nossos experimentos abrangem as várias complexidades dentro do domínio da computação.

Nosso estudo revela aspectos cruciais do desempenho de processadores SMT ao executar diferentes tipos de instruções. Ele abre o caminho para possíveis melhorias futuras. A ferramenta IAM que introduzimos fornece uma abordagem de agendamento de threads nova e pragmática que melhora substancialmente a eficiência de cargas de trabalho intensivas em computação. No entanto, o desempenho da ferramenta flutua entre diferentes tipos de instrução, destacando a necessidade de considerações específicas de instrução ao desenvolver e implementar escalonadores SMT.

Nossos resultados experimentais ilustram o valor dos algoritmos de mapeamento que levam em conta os tipos de instrução executados por cada thread para aumentar o desempenho geral. A ferramenta IAM, projetada para aplicação em tempo real, aumenta o desempenho do NAS Benchmark em uma média de 9,8% em comparação com os método de mapeamento Linux scheduler. Nossa pesquisa destacou que o desempenho depende do tipo de instruções executadas por cada thread rodando simultaneamente. Classificamos os aplicativos em quatro grupos com base na predominância de um tipo particular de instrução.

Instruções Intensivas de Cálculo: A ferramenta IAM apresenta excelente desempenho para tipos de instrução intensivas em cálculo, como operações de inteiros, ponto flutuante e branch. Ela emprega um método inovador de mapeamento de threads para núcleos com base em tipos de instrução, mitigando assim a disputa por unidades funcionais e levando a ganhos substanciais de desempenho. Portanto, para cargas de trabalho que usam principalmente essas instruções, o IAM surge como uma alternativa promissora aos escalonadores de sistema operacional convencionais.

Instruções Limitadas pela Memória: Para tipos de instrução limitados pela memória, especialmente operações de carga e armazenamento, os benefícios de desempenho do IAM são menos evidentes. Essa variação destaca a influência de fatores além da disputa de unidade funcional, notadamente a latência da memória, na formação do desempenho do SMT. Embora o IAM possa não ser ideal para essas cargas de trabalho em sua versão atual, sua abordagem de agendamento consciente de instruções oferece oportunidades para refinamento. As futuras iterações do IAM poderiam incorporar estratégias para antecipar e aliviar latências de memória, como ajustes dinâmicos na estratégia de agendamento com base em padrões de utilização de memória em tempo real.

Diretrizes para Otimização SMT: Nosso estudo enfatiza o valor das considerações específicas de instrução na otimização do desempenho SMT. A estratégia generalizada pode apenas explorar parcialmente as capacidades dos processadores SMT. Em vez disso, uma abordagem mais refinada que leve em conta as características específicas da carga de trabalho pode ser adotada. Estratégias como o IAM reduzem a disputa de unidade funcional e podem fornecer benefícios de desempenho substanciais para instruções intensivas em computação. Fatores relacionados à latência da memória e ao comportamento do cache se tornam críticos para instruções limitadas pela memória. A exploração contínua de estratégias de otimização de SMT específicas para instrução é justificada, e o potencial de ferramentas como o IAM para se adaptar dinamicamente a cargas de trabalho variáveis deve ser examinado.

Nossas descobertas avançam nosso entendimento do comportamento de desempenho do SMT e da influência das características de nível de instrução na formação deste comportamento. Este conhecimento é vital para desenvolver soluções inovadoras para melhorar a eficiência computacional, especialmente à medida que os processadores continuam a evoluir e os aplicativos se tornam cada vez mais diversos e complexos.

Além disso, nossa ferramenta é adaptável a qualquer ambiente de computação, aprimorando a alocação de recursos e facilitando o mapeamento de threads com base em características específicas de carga de trabalho. Ela pode ser benéfica tanto para provedores de infraestrutura quanto para usuários. Os provedores podem implementar este mecanismo em toda a sua infraestrutura de computação para aproveitar o aprimoramento do mapeamento. Ao mesmo tempo, os usuários podem aproveitar este mecanismo para otimizar suas cargas de trabalho, aumentando assim a eficiência computacional e o desempenho.

B.7 Conclusões e Direções Futuras

Diante dos processadores multicore continuamente evoluindo e se tornando mais complexos, espera-se que as implicações da competição de unidades funcionais se amplifiquem. Essa tendência é impulsionada pelas arquiteturas prospectivas onde um número crescente de núcleos compartilha unidades funcionais, exacerbando o desafio de mitigar a competição de recursos. Consequentemente, como discutido nesta tese, a elaboração de estratégias de mapeamento eficientes será fundamental na otimização da alocação de threads entre os núcleos. Isso permitirá a utilização total das capacidades computacionais nas futuras arquiteturas multicore, apesar das complexidades associadas ao compartilhamento de recursos.

Nossa pesquisa foi pioneira em um microbenchmark único para avaliar metodicamente o impacto do compartilhamento de recursos. Esta ferramenta, que se desvia do foco tradicional na competição de memória, considera a competição entre várias unidades funcionais, oferecendo uma compreensão mais matizada da dinâmica do compartilhamento de recursos em ambientes multicore.

Desenvolvemos uma estratégia inovadora para mapear threads de várias aplicações paralelas em arquiteturas multicore baseadas em SMT com base na competição de diferentes unidades funcionais. Este método aproveita um entendimento granular da competição de unidades funcionais para guiar decisões de alocação de threads, otimizando a utilização de recursos e alcançando um equilíbrio no uso de recursos.

Nossos resultados mostram melhorias de desempenho consideráveis, com um aumento médio geométrico de 9.8% em comparação com Linux scheduler. Essas melhorias são atribuídas principalmente à mitigação da competição de unidades funcionais através do posicionamento estratégico de threads, que atribui threads que sobrecarregam as mesmas unidades funcionais a núcleos distintos. Além disso, introduzimos uma métrica de dissimilaridade que indica uma correlação entre a semelhança de aplicativos e a degradação de desempenho quando mapeados para o mesmo núcleo.

Em termos de trabalho futuro, aspiramos expandir nossa investigação sobre a influência da competição de unidades funcionais em uma variedade mais abrangente de arquiteturas. Nosso foco inclui arquiteturas de ponta como Intel Sapphire Rapids e AMD Zen 5, enquanto nos esforçamos para aprofundar nosso entendimento sobre a gestão de recursos em processadores multicore. No processo, nosso objetivo é formular um conjunto abrangente de diretrizes que permitirá a utilização eficiente dessas e futuras arquiteturas multicore, permitindo-nos aproveitar suas vastas capacidades computacionais sem sucumbir à competição de recursos.

Adicionalmente, vislumbramos a integração de nossa ferramenta no kernel do Linux, uma progressão lógica dada a função do kernel na gestão do posicionamento de threads nos núcleos do processador. A incorporação de nossa ferramenta nas trocas de contexto do kernel do Linux permitirá o monitoramento e a gestão em tempo real da competição de recursos. Este arranjo poderia gerir a alocação de recursos com base nas percepções da ferramenta sobre a competição de unidades funcionais, diminuindo assim os impactos no desempenho. Isso poderia resultar em um controle mais refinado e uma gestão de recursos aprimorada, culminando em desempenho de sistema otimizado.

Por essa abordagem, nossa ferramenta poderia se adaptar continuamente ao comportamento de cada aplicação em execução no sistema, ajustando recursos conforme necessário para maximizar o desempenho. Isso permitiria a implementação de estratégias de gestão de recursos dinâmicas e autoajustáveis onde o kernel aprende continuamente e se adapta à carga de trabalho do sistema.

Em resumo, nosso trabalho futuro visa expandir nossa compreensão da competição de recursos em arquiteturas multicore e investigar a integração de nossa ferramenta no kernel do Linux. Esses esforços progressivos lançarão as bases para estratégias de gestão de recursos aprimoradas, impulsionando uma utilização mais eficiente e eficaz dos processadores multicore. Esses avanços poderiam ter implicações de longo alcance em vários domínios, incluindo data centers, eletrônicos de consumo e tecnologia móvel.