Resource Optimized Scheduling For Enhanced Power Efficiency And Throughput On Chip Multi Processor Platforms

Shivam Kundan
Southern Illinois University Carbondale, shivamkundan@hotmail.com

RESOURCE-OPTIMIZED SCHEDULING FOR ENHANCED POWER EFFICIENCY AND THROUGHPUT ON CHIP MULTI-PROCESSOR PLATFORMS

by

Shivam Kundan

M.S., Southern Illinois University Carbondale, 2019

A Thesis
Submitted in Partial Fulfillment of the Requirements for the
Doctor of Philosophy Degree

School of Electrical, Computer, and Biomedical Engineering
in the Graduate School
Southern Illinois University Carbondale
May 2024

DISSERTATION APPROVAL

RESOURCE-OPTIMIZED SCHEDULING FOR ENHANCED POWER EFFICIENCY AND THROUGHPUT ON CHIP MULTI-PROCESSOR PLATFORMS

by

Shivam Kundan

A Thesis Submitted in Partial

Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in the field of Electrical and Computer Engineering

Approved by:

Dr. Iraklis Anagnostopoulos, Chair

Dr. Dimitri Kagaris, Co-Chair

Dr. Spyros Tragoudas

Dr. Chao Lu

Dr. Khaled Ahmed

Graduate School
Southern Illinois University Carbondale
April 9, 2024

AN ABSTRACT OF THE THESIS OF

Shivam Kundan, for the Doctor of Philosophy degree in ELECTRICAL AND COMPUTER ENGINEERING, presented on April 9, 2024, at Southern Illinois University Carbondale.

TITLE: RESOURCE-OPTIMIZED SCHEDULING FOR ENHANCED POWER EFFICIENCY AND THROUGHPUT ON CHIP MULTI-PROCESSOR PLATFORMS

MAJOR PROFESSOR: Dr. I. Anagnostopoulos

The parallel nature of process execution on Chip Multi-Processors (CMPs) has boosted levels of application performance far beyond the capabilities of erstwhile single-core designs. Generally, CMPs offer improved performance by integrating multiple simpler *cores* onto a single die that share certain computing resources among them such as last-level caches, data buses, and main memory. This ensures architectural simplicity while also boosting performance for multi-threaded applications. However, a major trade-off associated with this approach is that concurrently executing applications incur performance degradation if their collective resource requirements exceed the total amount of resources available to the system. If dynamic resource allocation is not carefully considered, the potential performance gain from having multiple cores may be outweighed by the losses due to contention for allocation of shared resources. Additionally, CMPs with inbuilt dynamic voltage-frequency scaling (DVFS) mechanisms may try to compensate for the performance bottleneck by scaling to higher clock frequencies. For performance degradation due to shared-resource contention, this does not necessarily improve performance but does ensure a significant penalty on power consumption due to the quadratic relation of electrical power and voltage ($P_{dynamic} \propto V^2 \cdot f$).

This dissertation presents novel methodologies for balancing the competing requirements of high performance, fairness of execution, and enforcement of priority, while also ensuring overall power efficiency of CMPs. Specifically, we (1) Analyze the problem of resource interference during concurrent process execution and propose two fine-grained scheduling methodologies for improving overall performance and fairness, (2) Develop an approach for enforcement of priority (i.e., minimum performance) for specific processes while avoiding resource starvation for others, and (3) Present a machine-learning approach for maximizing the power efficiency (performance-per-Watt) of CMPs through estimation of a workload's performance and power consumption limits at different clock frequencies.

As modern computing workloads become increasingly dynamic, and computers themselves become increasingly ubiquitous, the problem of finding the ideal balance between performance and power consumption of CMPs is of particular relevance today, especially given the unprecedented proliferation of embedded devices for use in Internet-of-Things, edge computing, smart wearables, and even exotic experiments such as space probes comprised entirely of a CMP, sensors, and an antenna ("space chips"). Additionally, reducing power consumption while maintaining constant performance can contribute to addressing the growing problem of dark silicon.

DEDICATION

To my father, who inspired and enabled my life-long dream of pursuing engineering. To my mother, who has always supported and encouraged my academic pursuits through all the many ups and downs. To my sister, who has always been a source of support and amusing mischief. To my late grandfather, who encouraged me to always dream big and boldly follow my passions. To my grandmother, who has always had unshakeable belief in me. To my aunt, who has been a role-model for reasons both academic and more. Finally, to my cat Data, who has been a constant source of companionship, happiness, and delight throughout my college education.

ACKNOWLEDGMENTS

I would like to thank Dr. Iraklis Anagnostopoulos for his invaluable assistance, guidance, and tremendous patience all throughout my undergraduate and graduate study, and for encouraging me to purse a PhD. I would also like to thank Dr. Spyros Tragoudas who was the one to first suggest pursuing a graduate education. I would like to thank my committee co-chair Dr. Dimitri Kagaris for inspiring my interest in the field, through his excellent teaching in the many undergraduate and graduate courses I took under him. I would also like to thank Dr. Chao Lu and Dr. Khaled Ahmed for taking the time to be part of my dissertation committee. Finally, a sincere thank you to my fellow PhD students Theodoros Marinakis, Ioannis Galanis, Zois Tasoulas, and Ourania Spantidi for their constant help and support in both classwork and research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1
INTRODUCTION

This chapter provides technological and historical background into the development of Chip Multi-Processor (CMP) architectures and discusses the importance of exploring both resource- & power-aware scheduling techniques for current and future CMP architectures.

1.1 DECLINE OF SINGLE-CORE ARCHITECTURES

From the late 1990's, chip manufacturers began development of multi-processor architectures to address the growing shortcomings of existing single-core designs. Among the major challenges to improving single-thread performance were the diminishing returns on application performance at higher clock frequencies and the growing difficulty of heat dissipation due to unprecedented device densities on the die [2]. In the preceding decades, chip manufacturers, consumers, and enthusiasts alike had become accustomed to large increases in clock speed with each successive generation of processors. However, at the turn of the century, the industry inevitably experienced a slowdown in the growth of both clock frequency and single-thread performance, despite steady advances still continuing at the device (transistor) level. This led many observers to speculate an imminent end to the performance predictions associated with Moore's Law. Figure 1.1 shows a compilation of nearly 50 years of data depicting raw transistor counts and the corresponding clock frequencies and single-thread performance (measured by SpecINT benchmarks [3]). The key detail in the graph is the discrepancy between the growth of raw transistor count compared to the other metrics such as single-thread performance and clock frequency.

In addition to heat dissipation challenges, single-thread performance was also increasingly bottlenecked by the growing divide between main memory and CPU speeds (i.e., the Memory Wall). For example, from 1986 to 2000 the average CPU clock speed increased by 55% annually while the speed of memory accesses increased at only 10% [4].

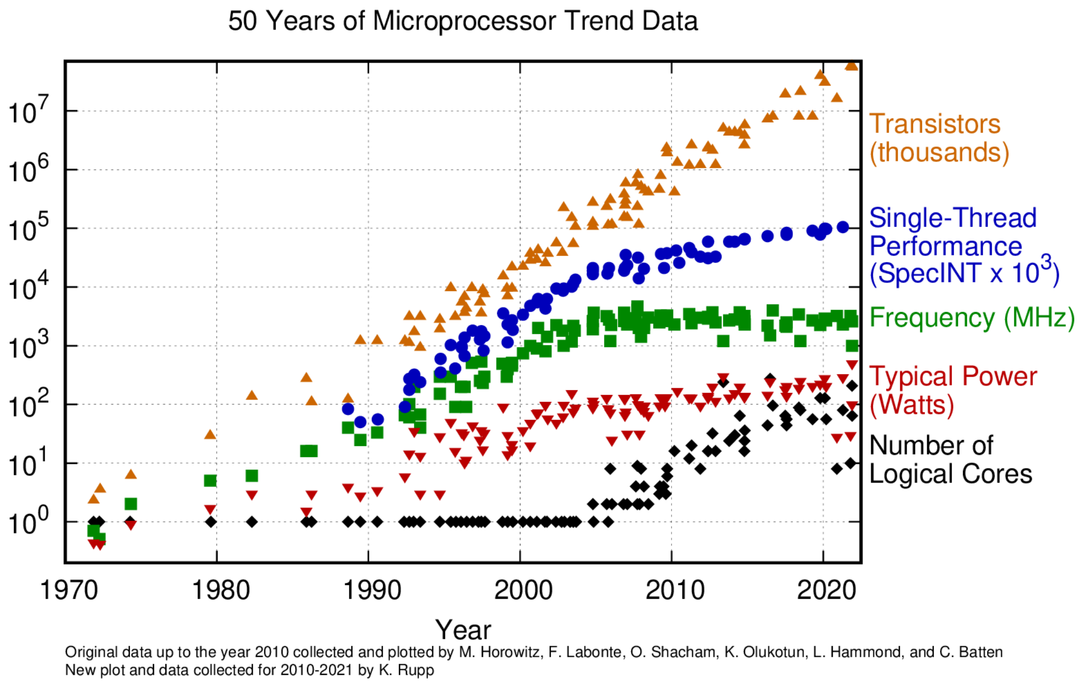


Figure 1.1: Processor trends related to Moore's Law [1]

In order to "squeeze-out" additional performance gains from single-core architectures, chip designers experimented with various techniques to increase parallelism at the instruction level. Among the developments were deep execution pipelines, superscalar architectures, Very Long Instruction Word (VLIW) architectures, and proprietary systems such as Explicitly Parallel Instruction Computing (by Intel) [5]. While these approaches improved single-thread performance, they came at the cost of greater hardware complexity and a corresponding increase in power consumption. One example of this approach was the introduction of a 31-stage pipeline in the final variations of Intel's *Pentium* line of processors [6].

1.2 CHIP MULTI PROCESSORS

One approach to improving performance without increasing raw clock frequency (and subsequent power consumption and heat) was to distribute computational workloads among several different processors. Each process could then be executed on its own distinct *core* and could communicate with concurrently executing processes through message-passing or accessing shared memory locations. Although the concept of multiprocessing itself had been known since 1843 [7], manufacturing an entire multiprocessor system on a single chip was both an unprecedented challenge as well as a significant breakthrough in the history of computing. The first commercially available Chip Multi-Processor (designed for servers and workstations), IBM's *Power4* was released in 2001 [8]. It comprised two 64-bit cores embedded onto the same die, sharing a high-speed L2 cache and buses for inter-core communication at rates of over 35GB/s [8]. Realizing the obvious advantages, other chip manufacturers such as Intel and Motorola also adopted the CMP architectural paradigm.

The multi-processor approach of using a number of simpler and less powerful cores offered considerable practical benefits to the end user. Novel CMP architectures were developed to address the primary obstacles to single-thread performance improvement, namely the fast-approaching Memory and ILP Walls, and reduction of dynamic power consumption to lower the amount of heat generated by the chip (i.e., power wall). CMPs also addressed the discrepancy between processor speed and rate of memory accesses to a large extent. Combining multiple CPUs onto a single die allowed for superior cache coherency, while simply having multiple simpler cores increased the overall rate of memory accesses.

Combined with the changing landscape of consumer computing in the mid-2000's, CMP architectures facilitated several breakthroughs in the field of embedded systems. The era of mobile computing, Internet-of-Things (IoT), and bio-medical electronics were made possible in part by the development of ultra low-power multiprocessor architectures. Despite requiring a more complex manufacturing process, the performance benefits of CMPs were substantial enough to quickly make them a staple of modern consumer electronics. Additionally, CMPs made multitasking the norm rather than a rare exception for embedded systems.

1.3 CHIP MULTI PROCESSOR CHALLENGES

Despite their significant advantages, the rise of CMP architectures came at the cost of higher hardware and software complexity. As CMPs enabled multiple processes to be executed in parallel, one of the major software challenges was to determine how concurrently running applications might affect the performance and power consumption of one another. Despite each process being able to run on its own core, certain resources like caches, internal buses, and main memory must still be shared among concurrently executing processes to ensure design simplicity and enable multi-threaded applications to boost performance by sharing a common cache [9]. This means that individual processes may have to 'contend' for control of the limited resources, resulting in a potential drop in overall performance. In some cases, the threads of multi-threaded applications may interfere with each other if fine-grained resource management is not considered. Contention for allocation of shared resources would also lead to novel power/thermal management challenges such as the problem of reducing *dark silicon* areas on the chip. Another problem arising due to shared-resource contention was the increased difficulty of enforcing priority and fairness among concurrent processes due to increased unpredictability (i.e., non-determinism). This would especially affect the functioning of the rapidly growing cloud services industry which relies on specific Service-Level Agreements (SLAs) for offering different tiers of performance.

Although resource allocation was also a requirement in single-core processors, the dynamic and parallel nature of CMP process execution added a significant level of complexity to the process. As modern processors follow a non-deterministic flow of execution, the number and type of processes being executed on a CMP can vary greatly between successive time periods, leading to cases where applications interfere with each other. In contrast, the amount of computing resources available to a process in single-core execution is usually well known, with the details varying only slightly between architectures.

1.4 CONCLUSION

Shared resource contention is the primary bottleneck to concurrent application performance in CMPs. Sophisticated management of shared resources can yield diverse benefits such as improved performance, higher throughput, better enforcement of fairness and priority, lower impact of non-determinism on dynamic execution, lower power consumption, lower percentage of dark silicon, and generally higher performance-per-Watt for the overall system. Additionally, fine-grained tracking and allocation of shared resource usage can aid in the development of dynamic frequency-scaling approaches. When used effectively, Dynamic Voltage & Frequency Scaling (DVFS) capability can be used to lower power consumption without incurring any penalty in performance.

Given the diverse benefits, it is imperative to develop sophisticated scheduling algorithms that account for both the efficient allocation of shared resources as well as the judicious use of higher clock frequencies.

1.5 RESEARCH QUESTIONS

1. Regarding the relationship between application performance and shared-resource contention on CMPs:
    (a) To what extent, if any, do concurrently executing applications affect the performance of one another? How can the level of interference be quantified?
    (b) What role do shared resources play in facilitating (or hindering) concurrent application execution?
    (c) What is the performance benefit of parallel execution on CMPs compared to time-multiplexed single-core multitasking?
2. What is the impact of shared-resource contention on the overall power consumption of CMPs when executing concurrent workloads?
3. Regarding improvement in application performance per unit of power consumed:

   (a) Can management of shared-resource contention help guide dynamic scaling of clock frequency (using DVFS)?

   (b) Can power consumption be reduced without affecting concurrent application performance?

   (c) Is it practical to scale the operating frequency at runtime?

   (d) What is the trade-off between overhead vs performance improvement when using power-aware scheduling guided by resource pressure management?

4. Using resource-aware scheduling, what is the optimum combination of performance and power consumption that can be obtained?

1.6 CONTRIBUTIONS

1. A holistic analysis of the impact of shared-resource contention on concurrent application performance in CMPs. Identifying and quantifying a given process's contentiousness and sensitivity to resource pressure. Based on this analysis, development of two fine-grained pressure-aware scheduling methodologies to maximize overall throughput and fairness.

2. A methodology for enforcing minimum performance thresholds in an execution framework comprising two priority levels for processes (high and low). Our approach ensures that high-priority applications meet their performance requirements while simultaneously preventing low-priority applications from experiencing resource starvation. This approach also results in a net improvement in overall throughput.

3. A run-time algorithm for proactively scaling operating frequency based on Performance Monitoring Counter (PMC) values to improve the overall power efficiency (IPS-per-Watt) of a clustered RISC CMP architecture.

4. (a) A mechanism to predict the power consumption and performance of concurrent

workloads at all available operating frequencies of a CMP, using PMC counts as inputs.

(b) A scheduling methodology which selectively switches frequency to satisfy both performance and power constraints

CHAPTER 2
LITERATURE REVIEW

Several works have sought to address the multifaceted problem of how to best manage shared resources on chip-multiprocessors in order to achieve a broad variety of goals such as improving tail latency, maximizing IPC, enforcing fairness, regulating temperature, and avoiding hot-spots. This chapter analyzes these approaches from two broad perspectives - improving purely performance-related metrics such as IPC and fairness, and improving power-related metrics such as the total power consumption of the chip, or the power consumption of a specific component on the chip. There is significant overlap between some works, which may fit into both sections.

2.1 CONTENTION-AWARE SCHEDULING

Most works that address shared-resource contention in CMPs use two main strategies - resource utilization monitoring and application classification.

In [9, 10], the authors aim to improve the fairness of resource allocation among concurrent applications by means of a sampling-based approach. For each time quantum, concurrent applications are profiled and re-scheduled if needed to minimize the amount of resource-based interference between them. The methodology in [11] presents a distributed resource allocation technique for multi-threaded applications using pool-based clustering of the available cores. Clusters are created based on system's characteristics and the allocation of cores is performed in a manner so as to increase resource utilization and reduce fragmentation. The authors of [12] use an IPC-per-core based approach, along with application classification for determining the ideal number of threads as well as the best thread-to-core mapping for a given set of workloads.

Several methodologies classify the memory, cache, and performance usage patterns of applications in order to schedule them into contention-minimizing groups. The authors

of [13] propose a memory-aware scheduling methodology which monitors realtime memory usage and cache miss rates without the need for application profiling. A drawback of this approach is that it only works for cache architectures with LRU policies. Another work which utilizes resource monitoring is [14]. This methodology relies upon the L1 cache bandwidth to identify and minimize shared resource contention. However, the authors do not consider other shared resources such as memory and L2 cache accesses. The authors of [15] proposes a two-tier a methodology to address both shared-resource contention and power consumption of many-core processors. They exploit usage patterns of shared LLC's to schedule the given workloads into groups. The drawback of this approach is that it relies upon per-core voltage monitors, which is not feasible for many CMP architectures [16]. The authors of [17] present a scalable methodology for dynamic resource allocation on CMPs. Although they increase weighted speedup and scheduling fairness, they do not consider power consumption and frequency scaling. The authors of [18] use a machine learning based approach for anticipating the level of shared-resource contention based on past resource allocation statistics. Their use of a runtime global monitoring and scheduling methodology increases the performance overhead considerably. [19] uses a novel MIMO controller methodology to maximize resource allocation efficiency for concurrently executing applications. Although this method improves performance over heuristic approaches, the added hardware required decreases the practicality of this approach.
In [20], the authors propose a methodology to improve fairness among concurrently executing applications by using a novel method to quantize execution progress by using instructions-per-second. The applications are then assigned to cores based on progress. The authors demonstrate an 86% improvement in fairness over evaluated schedulers. In [21], the authors develop the progress measurement metric further to create application-to-core mappings that balance both bandwidth and LLC requirements. This method improves average performance gain by 6.3% to 16% while enforcing fairness of execution.

Other works utilize resource monitoring to achieve domain-specific goals such as the enforcement of priority and maximization of fairness in dynamically scheduled systems. In [22] and [23], the authors aim to reduce the performance degradation of applications due to contention for main-memory bandwidth. The authors of [22] present a scheduling methodology that balances memory bandwidth across all executing applications. In [23], an improvement in performance is achieved by means of a per-core memory regulator and reclaim manager.

Numerous works, such as [24], [25], and [26] aim to improve performance of prioritized applications through hardware-assisted cache partitioning techniques. The authors of [24] present an application categorization methodology based on cache utilization. The applications are scheduled according to their class co-execution characteristics. In both [25] and [26], the authors present a scheduling methodology that uses Intel's Cache Allocation Technology (CAT) to create partitions that reduce contention for the LLC. Furthermore, the authors of [26] demonstrate how their methodology can be used to improve performance for prioritized applications. However, a major drawback of these methods is their need for specialized hardware support. The work presented in [27] proposes three approaches to resource-management for hard real-time embedded systems, with the goal of reducing Worst-Case Execution Time (WCET) and Worst Case Reaction Time (WCRT). It uses the 3-phase task execution model presented in [28], [29], and [30], to perform a contention analysis for the memory bus, a contention analysis for the cache bus, and a formulation for fixed task-priority approach to memory-centric scheduling. The bus contention analysis considers two different memory access models, i.e., dedicated and fair memory access models, built on top of the first-come-first-served (FCFS) bus arbitration policy to achieve its ultimate goal of determining the maximum contention that can be suffered by tasks. The contention analysis for the cache focuses on analyzing cache persistence to determine how LLC misses affect contention on the memory bus, to supplement the earlier analysis of memory bus contention and further improve the estimation of the impact of resource contention on WCRT. The memory-centric scheduling approach makes use of these analyses to reduce WCRT. [31] presents an approach to enforcing QoS for devices in the cloud-edge continuum. It aims to eliminate or minimize QoS violations due to shared-resource contention on edge devices that execute cloud micro-services. The authors utilize a mapping approach based on reinforcement learning to capture the complex contention behaviors of edge devices that execute co-located and dynamically allocated cloud micro-services. The authors demonstrate a reduction in computational resource usage by 23.9% while maintaining QoS and also improving tail latency. In [32] the authors propose a resource-management system to achieve desired video inference latency and accuracy trade-offs under changing runtime conditions in internet-connected mobile-devices. This work includes a contention-aware scheduling methodology that uses DNN models along with extensive offline profiling and considers resource contention on the CPU, GPU,and memory bus. The authors of [33] present a contention minimization methodology as part of a work to reduce QoS violations while executing processes for cloud micro-services. Their "resource-manager" determines optimal resource allocation based on reinforcement learning, which the authors claim can capture complex contention behaviors. In [34], the authors outline an efficient resource-management mechanism to predict contention-induced performance degradation as part of a larger method to meet specific Service Level Agreements (SLAs) for server utilization when providing Network Function (NF) Virtualization service. Their contention minimization methodology relies on precisely characterizing (1) the pressure each NF applies on the server's shared hardware resources (contentiousness) and (2) how susceptible each NF is to performance drop due to competing contentiousness (sensitivity). They demonstrate a 6% to 14% improvement in server utilization efficiency.

2.2 POWER-AWARE SCHEDULING

A number of works address the reduction of power consumption in CMPs while also maintaining a minimum level of performance and/or throughput.

First, regarding the power consumption of clustered multi-processors, the authors of [35] present a methodology that utilizes a PI controller for monitoring and reducing power consumption. However, the individual characteristics of the workloads are not taken into consideration, effectively treating the processor as a ‘black box’ with two inputs (operating frequency and the predefined control parameters) and one output (power). The authors of [36, 37] present a methodology which uses Memory-Reads per Instruction (MRPI) to meet application-specific performance requirements (IPS) and reduce energy consumption by predicting the operating frequency of the processor. Similarly, the authors in [38, 39] adjust operating frequency based on the memory accesses and the required response time. However, these approaches allow for significant energy and power reduction only when memory-intensive applications are part of the application mix. Additionally, the authors in [40] present an agent-based power system for frequency selection based on inter-core communication. However, the frequency selection does not consider contention effects. Furthermore, the authors in [18] present a machine learning approach which tracks the usage of multiple shared resources of the processor in order to improve performance of multiprogrammed workloads. However, the authors do not analyze the selection of the inputs to their neural network, making the degree of multicollinearity in the training labels uncertain. Moreover, while the latter tries to improve performance of workloads, it does not consider the trade-off between the increase in performance and power consumption, leading to lowered reduced efficiencies.

The authors of [26] present a contention-aware scheduling policy which leverages Intel’s Top Down Microarchitecture Analysis Method (TMAM) for application profiling. The TMAM method allows for obtaining an accurate representation of application’s resource utilization which is in turn used to develop their application-to-core placement technique. At run-time, the application groupings are determined by a cost function that seeks to minimize performance (IPC) loss due to contention for the shared LLC. A drawback of this method is that it requires Intel’s Cache Allocation Technology (CAT) support for the processor, available only in specific models. In [41], the authors aim to avoid contention on the LLC by combining detailed resource-utilization information with a resource-aware application-to-core placement policy and use of Intel's Cache Allocation Technology (CAT). Their methodology utilizes a two-step approach which first calculates the ideal cache partition (using hierarchical clustering) and then calculates the ideal application-to-core placement (using a heuristic based on cache-miss curves). The key contributor to overhead is the quadratic-time Look-ahead algorithm used at run-time to determine the best cache partition. When there are more than 8 available applications (and 12-way cache), the scheduling overhead increases significantly, rendering this approach impractical in real-work applications. In addition, this method also requires specialized hardware support (Intel's CAT). Additionally, several other studies have focused on quantifying the impact of cross-core interference and deciding application groupings to to boost application throughput [42, 43, 44, 45, 46]. These works combine varied techniques to reduce the performance impact of shared-resource contention. However, they differ from our proposed approach as they do not holistically monitor contention for all levels of the memory hierarchy. In [47], the authors present a contention-aware scheduler which utilizes cache misses per million instructions as a heuristic for quantifying contention. While this method works successfully for highly memory intensive groups of applications, it yields lower performance for other workload types. In [48], the bandwidth between links of the memory hierarchy is tracked to capture activity of the applications. This method identifies cache-intensive applications and avoids scheduling them with applications that heavily thrash the LLC. However, unlike our proposed static scheduler, this method is less *fine-grained* and cannot detect applications which rely on cache but their cache access pattern makes them less susceptible to LLC interference (cache non-sensitive). In [49] a combination of application profiling and neural networks are used to maximize the performance (Instructions per Second (IPS)) of varied resource-intensive workloads. Although this work demonstrates an increase in performance over Linux scheduler, their coarse-grained characterization scheme leaves room for further improvements in both individual and overall application performance. A fine-grained application characterization is utilized in [50] to improve the performance (IPC) of prioritized applications executing under shared-resource contention. Even though this method uses fine-grained statistics, its primary goal is to improve performance for specific high-priority applications in a workload, in contrast to our approach which considers workload performance. In [51], the authors present a methodology to improve performance for convolutional neural network workloads on heterogeneous multicore processors. This method explores the operating frequencies of the device, prunes the design space and decides the optimal device configuration according to system objectives (minimize power/maximize performance). Experimental results demonstrate 42.8% to 61.5% reduction in power consumption. The authors of [42] aim to improve the efficiency of shared resource utilization while also preserving the required Quality-of-Service (QoS) (measured by tail latency) of each scheduled application. They accomplish this by extracting the resource-interference profiles of each application using specially developed micro-benchmarks and performing a series of stochastic gradient descent operations to decipher the ideal application-to-core placement. A shortcoming of this approach is that it requires profiling for each application using specially designed benchmarks. This renders the approach impractical in cases where additional CPU time cannot be allocated for separate profiling of applications. The authors of [43, 45] utilize approximate computing techniques to improve resource utilization for shared servers based on CMP architectures. Both works aim to keep performance degradation of each application within specified tolerance limits. They take the performance tolerance threshold (measured in % slack in tail latency) as input and output the ideal application-to-core placement. However, both techniques need to dynamically recompile the applications which increases the run-time overhead and precludes their usage in cases where the source code is restricted. In [46], the authors utilize deep learning techniques in order to anticipate spatial and temporal patterns in application execution that may result in quality of service degradation in CMP servers. Their method involves obtaining resource utilization information from remote procedure call (RPC) level traces of each scheduled application to proactively detect any upcoming violations in tail latency requirements. In this manner, the authors aim to improve predictability of cloud microservices and prevent any single misbehavior from causing a cascade of QoS violations. The main drawback of this approach is the need for an extremely large dataset for training the Deep Neural Networks (DNNs). Each DNN predictor requires up to 1 week's worth of execution data for training. The authors of [44] propose a method that uses a combination of containers, thread pinning, cache partitioning, frequency scaling, and memory capacity partitioning to allow latency-critical apps and microservices to be co-executed without QoS (tail latency) violations. They begin by performing a comprehensive analysis of resource requirements, contention effects, and sensitivity for different application types. Next, they identify 'fungible' computing resources that can be re-allocated at run-time to provide the necessary resources for applications. The Perf&Fair scheduler [52] uses an online average of performance and fairness as part of a fitness function to schedule applications for each time quantum. However, due to its increased focus on fairness, it generally results in lower overall performance. The BAOS scheduler [22] calculates overall main memory requests and the average main memory bandwidth utilization for each application per quantum. The application with the best fit in terms of bandwidth is selected to run during the following quantum. However, this method cannot prevent memory-intensive applications executing alongside cache-intensive applications, which leads to a drop in performance.

Finally, certain works employ resource-utilization strategies as part of a larger approach to meet domain-specific goals. In [53], the authors present an approach that aims to enforce varying QoS requirements for applications with different levels of execution priority. Their proposed method utilizes a distributed set of *'lightweight performance meters'* on each available core of the system. The performance meters can account for differences in the cores' architecture and help to coordinate overall resource allocation. The authors were able to demonstrate up to 24% improvement in memory bandwidth utilization while also enforcing QoS (measured in frames per second) for target applications. In [54], the authors express run-time constraints and mine multiple parameters in order to create optimized system- and application-aware operating points. This approach results in a balance between temperature, power consumption, and performance (instructions-per-second), resulting i 15.4% to 35.3% reduction in energy consumption. In [55], the authors propose a technique for improving the energy efficiency of heterogeneous multi core architectures while also maintaining a minimum user-defined fairness of execution. This is achieved through a combination of offline profiling/ application characterization and dynamic frequency scaling. The authors of [56] also present a methodology for improving the power efficiency of heterogeneous multi core processors used for edge-computing purposes. It uses a hardware-tailored multi-task programming model which yields an average of 22% increase in energy efficiency compared to a reference system.

CHAPTER 3
CONTENTION-AWARE SCHEDULING

Modern CMPs are have been steadily integrating an increasing number of cores into a single socket to address the continually growing demand for higher application performance. Generally, the cores of a CMP share several components of the memory hierarchy such as a Last-Level Cache (LLC) and main memory bandwidth. This allows for considerable gains in multi-threaded application performance, while also helping to maintain overall architectural simplicity. However, a consequence of sharing resources is the inevitable performance bottleneck caused by contention for shared resources among concurrently executing applications. In this chapter, we formulate a fine-grained application characterization methodology that leverages the Performance Monitoring Counters (PMCs) and Cache Monitoring Technology (CMT) available in Intel processors. We utilize this characterization methodology to develop two contention-aware scheduling policies, one *static* and one *dynamic*, that co-schedule applications based on their resource-interference profiles. Our approach focuses on minimizing contention on both the main-memory bandwidth and the LLC, by monitoring the pressure each application inflicts on these resources. We achieve performance benefits for diverse workloads, outperforming Linux and three state-of-the-art contention-aware schedulers in terms of system throughput and fairness for both single and multi-threaded workloads. Compared to Linux, our policy achieves up to 16% greater throughput for single-threaded and up to 40% greater throughput for multi-threaded applications. Additionally, the policies increase fairness by up to 65% for single-threaded and up to 130% for multi-threaded ones.

3.1 SHARED-RESOURCE CONTENTION

Chip Multi-core Processors (CMPs) have become the dominant architectural choice in the server and desktop processing domains due to their scalable computational capabilities at steadily decreasing costs. From the late 90's, CMPs began to successfully overcome the performance and heat dissipation bottlenecks faced by single-core designs through the use of sophisticated Thread-Level Parallelism (TLP) and aggressive scaling of cores. Integrating multiple simpler and more energy-efficient cores paved the way for higher performance yields, as greater throughput could be achieved by executing multiple applications in parallel. Performance boosts, energy-efficiency, and cost-effectiveness were some of the advantages that ultimately led to CMPs becoming the dominant design choice in the server and desktop domains.

However, the benefits offered by the CMP architectural paradigm also introduced a new kind of performance bottleneck: *shared-resource contention*. Since the cores of a CMP are not completely independent processors but are instead clustered together to share several components of the memory-hierarchy, contention among applications for access to these resources can result in significant performance degradation. Shared-resource contention can affect the performance of 1. applications executing concurrently on neighboring cores (noisy-neighbor problem), and 2. threads belonging to the same application if the application does not take advantage of fine-grained resource sharing [57] .

The first major point of resource contention is the Last Level Cache (LLC) [58, 59]. Cache replacement policies attempt to take advantage of temporal and spatial locality by bringing data to the LLC independently from the application level [57]. A thread may evict the data of a neighboring thread and its own data may be evicted by another [58]. The effect of concurrently executing threads competing for space in the LLC has been studied by many different researchers [59, 60, 61]. Their results demonstrate that the cache miss rate of a thread can vary significantly depending upon its co-runners, resulting in a performance (Instructions Per Cycle (IPC)) penalty which can be as high as 63% in extreme cases [62]. The other major source of resource contention is the main memory bandwidth [22, 48, 63, 64]. Concurrently executing applications on a chip multi-core processor with a shared memory system can suffer performance degradation due to interference in memory accesses. Authors in [65] demonstrate that the average memory latency of an application can increase up to 7× (translated to 60% IPC loss) when running concurrently with another memory-intensive application. Similarly, performance degradation is observed in experiments where bandwidth contention is injected by memory-intensive co-runners (up to 65% IPC drop [22], 2.2× slowdown [63]).

Resource contention can harm the performance of the system in various ways. Some contention-aware approaches target improvements in *resource efficiency and utilization* [66, 67], while others focus on *maximizing throughput* [47, 58, 68] and *balancing fairness* [20, 64, 69]. Previous approaches generally try to address the challenges posed by resource contention in two steps. First, they extract an *interference profile* for each application executed on the CMP platform. These profiles quantify the application's performance when competing with co-runners for allocation of shared resources [70, 71]. For each application, it is important to identify (1) its *sensitivity* to resource contention, and (2) its *contentiousness* [70] . After application characterization, the second step is the development of contention-aware strategies that optimize system performance by determining contention-minimizing application groupings [22, 47, 72] .

In this chapter, we present two pressure-aware scheduling policies, one *static* and one *dynamic*, for contention minimization on chip multiprocessor systems. Both policies utilize the inbuilt performance monitoring unit for fine-grained characterization and scheduling of applications based upon the pressure exerted on the processor's shared resources. Specifically, the innovations of our approach are:

- A fine-grained application characterization methodology that leverages hardware Performance Monitoring Counters (PMCs) and Intel's Cache Monitoring Technology (CMT) [73] to differentiate between applications with similar behavior. This approach analyzes the individual pressure exerted on shared resources by each application at run-time, in turn helping to guide the subsequent resource-aware placement of applications.

- Holistic exploration of the performance impact of resource contention among different application classes, instead of only pairs of classes as in previous works. Our approach focuses on monitoring the pressure exerted on *all available shared resources* of the system simultaneously. Particularly, our method accounts for the following factors: (1) resource pressure is generated by applications in different ways, and (2) applications are sensitive to different levels of pressure applied to various parts of the memory hierarchy.

- A *static* contention-aware scheduling policy that co-schedules applications based on their obtained resource-interference profiles. The key idea lies in our observation that applications of certain characteristics can be placed together without loss in performance if their collective resource pressure does not saturate the LLC and/or main memory bandwidth of the CMP platform.

- A *dynamic* contention-aware policy that co-schedules applications at run-time based on their ongoing resource-interference profiles. This policy uses the same characterization and placement principle as the static scheduling policy except that it does not require any previous information about the workload. It boosts the performance of applications that vary their patterns of resource usage frequently. In both scheduling policies, applications time-share the CPU in a way that evenly balances the pressure on the CMP's shared resources.

- Experimental evaluation of various single and multi-threaded application mixes on two different Intel servers with Cache Monitoring Technology (CMT), and comparison of results with the Linux scheduler (CFS [57]) and four state-of-the-art contention-aware schedulers [22, 47, 48, 52] .

Bench.	**MPMI used by DI [47]**	**LCA [72] classification**	**Sensitivity classification [70]**
stream	7,614.064	L	Contentious/Non-sensitive
ocean	2,936.885	L	Contentious/Non-sensitive
chase	9.040	C	Non-contentious/sensitive
gemm	0.272	C	Non-contentious/Non-sensitive
trmm	0.211	C	Non-contentious/sensitive
atax	0.167	C	Non-contentious/Non-sensitive
3mm	0.143	N	Non-contentious/Non-sensitive
2mm	0.137	N	Non-contentious/Non-sensitive

Table 3.1: Application classification (L = memory intensive, C = cache intensive and N = compute intensive).

3.2 MOTIVATION: THE EFFECT OF FINER-GRAIN SCHEDULING.

This section contains a motivational example that shows the necessity of fine-grain classification. Table 3.1 depicts a workload of eight applications to be scheduled on a 4-core Intel Core i7 processor with shared LLC and memory controller. The second column indicates the Misses Per Million Instructions (MPMI), which are used as a metric for the Distributed Intensity (DI) contention-aware scheduler [47]. The third column shows the application classification based on the Link and Cache-Aware (LCA) scheduler [72]. The last column depicts the application classification based on contentiousness and sensitivity as presented in [70]. We executed all the 35 possible unique co-scheduling combinations, each one of them consisting of two groups of four applications.

Figure 3.1 depicts the performance of each combination normalized to the optimal. We have also annotated the DI [47] and LCA [72] schedulers based on their decisions. With the DI being in the 22^{th} percentile, we observe that splitting the applications with "high miss rate" to avoid accumulation of memory intensive applications results in a sub-optimal decision. This happens because there are applications with "low miss rate" that show significant reliance on the LLC (C). Consequently, splitting memory intensive applications (2 in our case) and co-execute them with the cache-reliant applications leads to significant interference for the later. Overall, using LLC miss rate as a metric fails to distinguish applications with high LLC utilization from applications that restrict their activity in the lower parts of cache hierarchy.

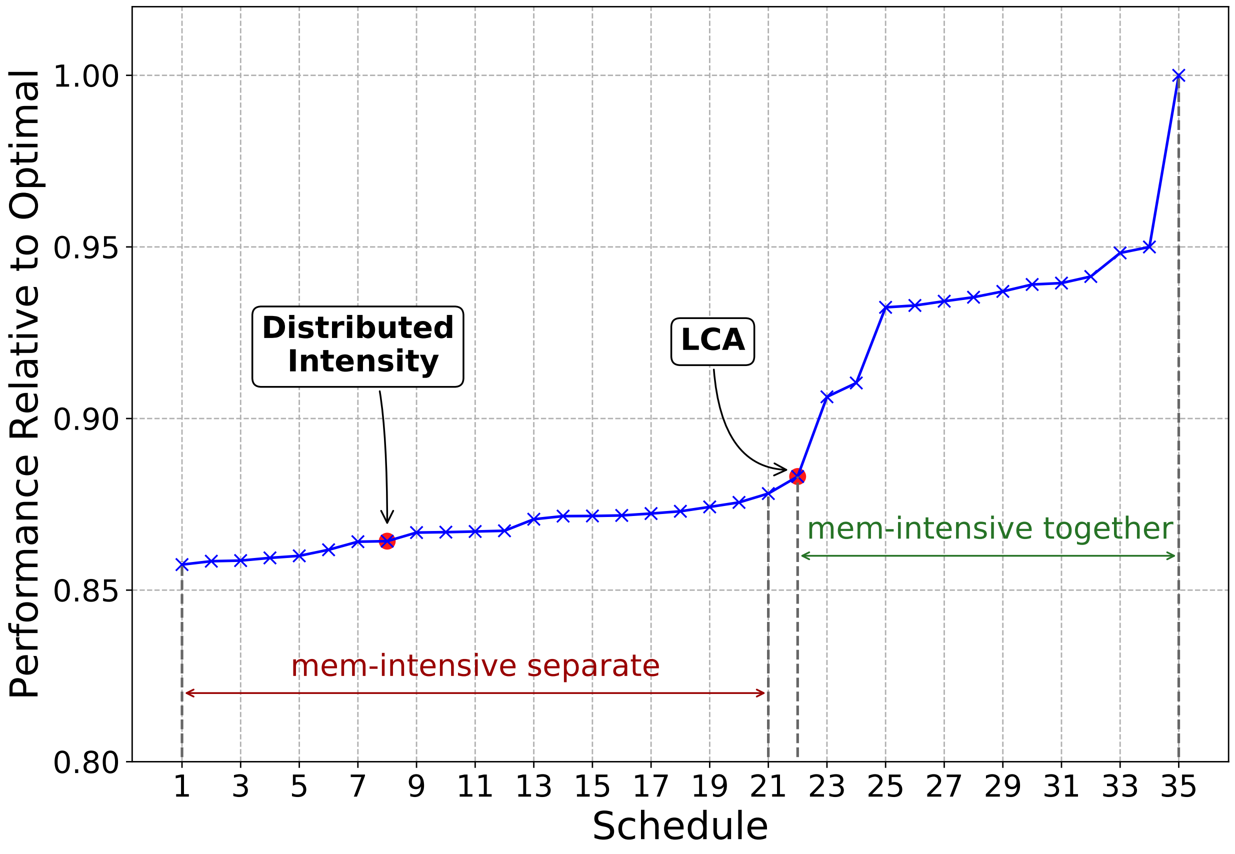


Figure 3.1: Performance of the 35 co-scheduling scenarios compared to the optimal case.

This problem is addressed by the LCA scheduler, which identifies the cache-intensive applications (C) and avoids co-execution with applications that heavily thrash the LLC (L). However, packing all the C applications together also results in performance losses. This happens because the C applications inflict a lot of pressure on the LLC significantly affecting their performance. Although, this policy is better than DI, as it is located on the 62^{th} percentile, there is still enough room for improvement. We see that solutions that group together the memory intensive applications comprise the top 38% of all the combinations. This is reasonable because the high miss rate applications, which are also classified as contentiousness, are not spread among groups to harm the sensitive cache-intensive ones. Also, packing them together does not hurt their performance, as the aggregate pressure they put on memory bandwidth can be tolerated and their memory requests can be satisfied.

Considering all the aforementioned observations, we conclude that none of the above classification methodologies alone provide sufficient information for finding the optimal scheduling policy. Thus, it is necessary to find *a more fine-grain application classification scheme* and examine the performance degradation of an application taking into consideration the following parameters: (1) the pressure the specific application imposes on the shared resources, (2) the total pressure inflicted on the shared resources by the co-runners, and (3) the sensitivity of the application.

3.3 PROPOSED METHODOLOGY

In this section, we propose two pressure-aware scheduling approaches, a static and a dynamic one. Both schedulers utilize Performance Monitoring Counters (PMCs) and Cache Monitoring Technology (CMT) to capture the pressure that applications exert on the shared resources and categorize the accordingly. The primary difference between them is that the static scheduler requires an offline profiling stage to determine the interference profile for each application and since it works offline, it can afford to perform complex computations to find the optimum grouping by using Mixed Integer Linear Programming (MILP) (Section 3.3.2). In contrast, the dynamic scheduler does not require any prior information about the workload but determines it at run time. Thus, it can only afford to run a first fit heuristic for scheduling the applications. This generally makes the performance of the static scheduler better than that of the dynamic one. However, the dynamic scheduler yields better performance when executing applications that exhibit frequent phase changes (i.e., changes in resource utilization patterns). Therefore, we split our scheduling methodology into two separate versions for use in the appropriate situation. At their core, both methods use the same underlying principle of fine-grained pressure-aware application placement.

Moreover, both proposed scheduling policies are platform independent. The only prerequisite is the ability to record per-core shared-memory bus bandwidth and last-level cache behavior, which currently is supported by most of systems. Particularly, for Intel x86 processors, this feature is available either through Intel Resource Director Technology (server class architectures) or Intel Performance Counter Monitor Technology (desktop class architectures).

The first part of this section (Section 3.3.1) presents a fine-grained application characterization methodology and analyzes the performance response of co-executing applications to varying levels of shared-resource contention. The second part (Section 3.3.2) presents the proposed pressure-aware static scheduler that focuses on minimizing contention for shared resources. Finally, the third part (Section 3.3.3) presents a modification of the static pressure-aware scheduler that allows for dynamic workload execution (i.e., no prior information about the workload is required at run-time). We utilize the processor's inbuilt Performance Monitoring Unit (PMU) to gather statistics for application characterization, interference analysis, and run-time decision making.

3.3.1 Application Characterization & Interference Analysis

Applications competing for the same shared resources can be executed concurrently under certain conditions, leading to an increase in system performance. In order to find which applications are good candidates for each application in a workload, we need to identify 1. the shared resource it utilizes, 2. the amount of pressure it puts on this resource, and 3. its sensitivity when competing with other co-runners for this resource.

Applying Pressure

The first step in our approach is to generate pressure with tunable intensity on the available shared resources of the system in order to test the behavior of applications with widely varying execution characteristics. We record the performance achieved throughout execution of each application while applying varying levels of pressure to the memory hierarchy. As a result, we can identify the source of interference and the amount of pressure at which the applications become vulnerable to a drop in performance. We developed a micro-benchmark resembling the streaming access pattern of the `STREAM` [74] benchmark. It helps us in profiling the sensitivity of applications under the *whole spectrum of interference* while also being able to *gradually increase the intensity of pressure* applied on the different shared resources. Furthermore, it has *aggressive behavior*, meaning that the intensity of resource pressure is maintained when competing with other applications. These characteristics help to accurately identify the level of pressure at which applications become susceptible to performance degradation. We note that application characterization can be affected by the number of threads a given application utilizes, and also by the specific input to the application. Our methodology considers this by treating benchmarks with different numbers of threads and input sizes as separate applications for the purpose of their subsequent scheduling (Sections 3.3.2 and 3.3.3) .

Application Characterization

We consider four classes of applications with regard to their contentiousness and sensitivity to hardware resources (Memory Bandwidth (BW) and Last Level Cache (LLC)): Memory Bandwidth Sensitive (BW): Applications with a memory usage pattern that resembles a streaming behavior and with a working set larger than LLC size. Consequently, they thrash the LLC (*LLC occupancy* = *LLC size*). LLC Sensitive (CS): Applications with low memory bandwidth and a working set size that fits entirely within the LLC. They experience severe performance degradation when their cache capacity requirements are not met. LLC Non-Sensitive (CNS): Applications that have low memory bandwidth and benefit from occupying a certain portion of the LLC, but their performance degradation is not as destructive as for their more sensitive twins (CS). Their cache access pattern makes

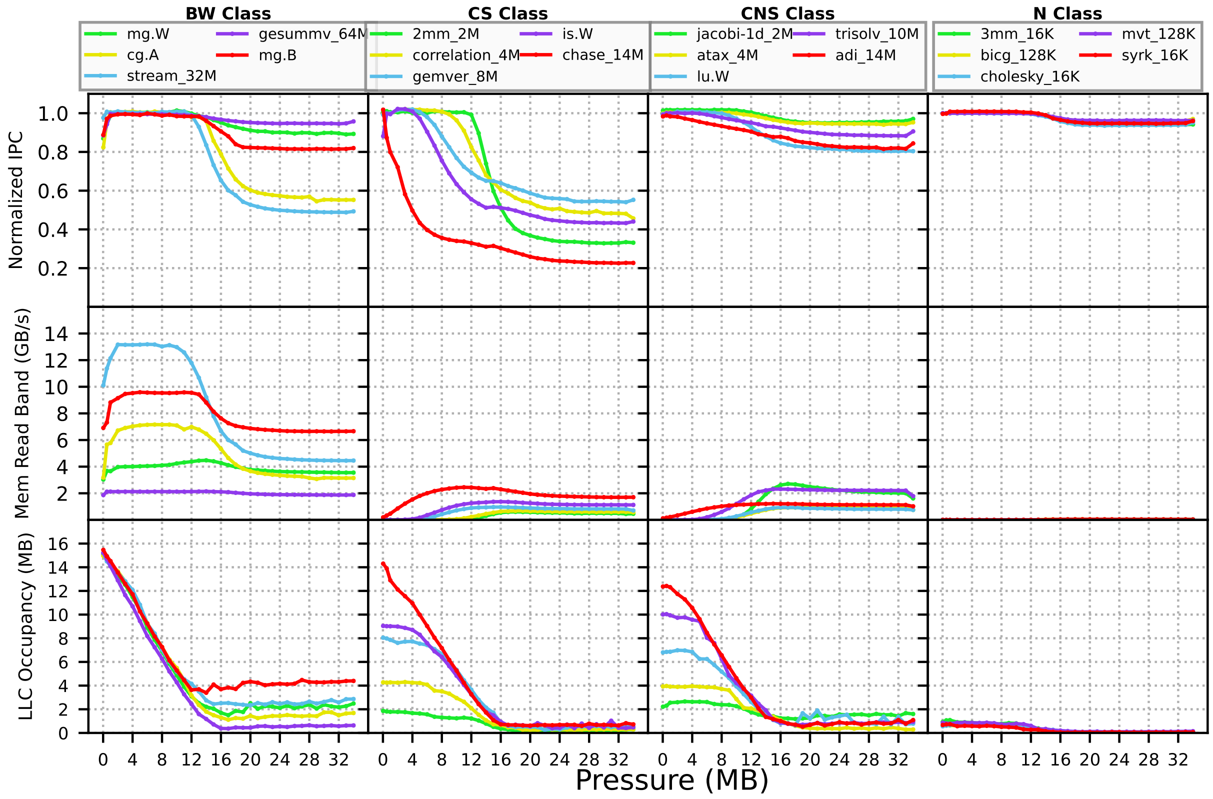


Figure 3.2: Behavior of the four identified classes under different pressure (max memory read bandwidth = 13.6 GB/s, LLC = 15.36MB).

them less susceptible to LLC interference and helps them retain a good performance level, even in the case of excessive LLC thrashing. Neutral (N): These are applications with negligible memory bandwidth requirements and no dependence on the LLC. Their activity is restricted to the lower parts of the memory hierarchy (L2 and L1 caches).

Table 3.2 depicts a summary in terms of their contentiousness and sensitivity on the shared resources. As experimental validation of these 4 classes, we executed our custom micro-benchmark along with a variety of applications selected from different benchmark suites

(NAS [75], Polybench [76], SPLASH2 [77], Stream [74], and Chase [78]) on a 6-core Intel Xeon E5-2620 v3 server, with LLC size of 15, 360 KB and maximum sustainable memory bandwidth of 33GB/s. More details of our set up can be found in Table 3.3,

Classes	**Memory Bandwidth**		**LLC**	
	Contributes	Affected	Contributes	Affected
BW	✓	✓	✓	✗
CS	✗	✓	✓	✓
CNS	✗	✗	✓	✗
N	✗	✗	✗	✗

Table 3.2: Qualitative comparison of application classes based on their contentiousness and sensitivity on the shared resources.

Section 3.4.1. Figure 3.2 depicts the interference profiles of several applications when co-executed with our micro benchmark exerting different levels of resource pressure. We note that the plots shown in Figure 3.2 were individually obtained per application, but were just plotted together in the same figure. In particular, each one of the selected applications is pinned on a core and it is tested against all different contention conditions, from negligible to severe interference in the LLC and memory link. The horizontal axis shows the size of the working dataset (in MB) of the micro benchmark, which is proportional to the level of pressure exerted on shared resources. Working datasets of 16MB and smaller apply pressure to the L1, L2, and L3 (LLC) caches, while working datasets greater than 16MB apply pressure to the memory bandwidth. Specifically, we are interested in the performance (normalized IPC), the memory read bandwidth, and LLC occupancy.

We group the results into four categories, based on the performance response of the tested applications: The first column of Figure 3.2 depicts BW applications with working set larger than the LLC size, that thrash the LLC and have no reliance on it (more in-depth analysis is presented in Section 3.3.1). While their LLC occupancy keeps reducing in proportion to the applied pressure of the micro-benchmark, their IPC is not affected. On the contrary, we observe that their performance is dictated by their achieved memory bandwidth. Reduction in memory bandwidth is directly connected with a drop in IPC, which occurs due to interference in memory requests that cannot be simultaneously satisfied for both the application being tested as well as for the micro-benchmark.

The second column of Figure 3.2 depicts CS applications with low memory bandwidth

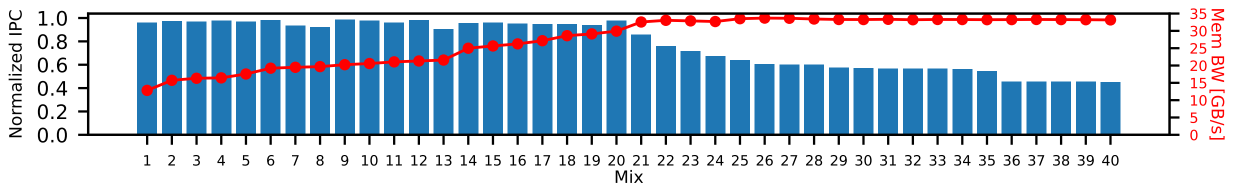


Figure 3.3: Memory bandwidth sensitive applications are not affected when the overall bandwidth is less than the maximum available (33 GB/s).

and a working set size that fits entirely within the LLC. We observe that when their cache capacity requirements are not satisfied, due to applied pressure on the LLC, their IPC drops up to 65%. Reduced cache occupancy results in higher number of evicted cache lines, increase in cache misses, and a subsequent increase in memory bandwidth pressure. Note that applications get negatively affected only beyond the point where both their working set size and the pressure applied by the micro-benchmark cannot be collectively accommodated by the LLC. For example, applications with high LLC occupancy (e.g., `chase_14M`, `is.W`) suffer from performance loss at a much lower level of resource pressure than applications with low LLC occupancy (e.g., `correlation_4M`, `2mm_2m`).

The third column of Figure 3.2 depicts CNS applications. We observe that their cache access pattern makes them less susceptible to LLC interference and helps them retain a good performance level (IPC drops around 20% in the worst case) even in cases of excessive LLC thrashing (i.e., when micro-benchmark working set $\geq 16\ MB$, causing high memory bandwidth).

For this reason they are able to retain a good level of performance (IPC drops around 20% in the worst case) even in the case of excessive LLC thrashing (i.e., micro-benchmark working set $\geq 16\ MB$, high memory bandwidth).

The last column of Figure 3.2 depicts N applications. It can be seen that their performance is not affected by the different levels of pressure exerted by our micro-benchmark on shared resources (max IPC drop $\simeq 5\%$) as their working set fits in the lower parts of the memory hierarchy (L2 and L1 caches).

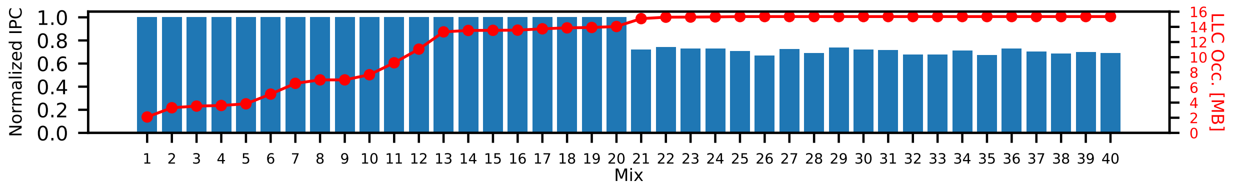


Figure 3.4: LLC sensitive applications are not affected when the overall pressure is less than the LLC size (15.36 MB).

Interference Analysis

In Section 3.3.1, we identified the characteristics and performance response for each class of applications when subject to different amounts of shared-resource pressure. However, not all applications that belong to the same class can be treated uniformly in terms of sensitivity to shared-resource contention, primarily because their slowdown occurs under different levels of pressure. For example, consider the first column of Figure 3.2 which depicts the performance of the BW class of applications to different levels of shared-resource pressure. The application `stream_32M` requires the highest memory bandwidth (∼13GB/s) to achieve its ideal performance (i.e., normalized IPC=1) and hence starts to experience performance degradation earlier than all other applications in its class (at pressure ∼12MB). Meanwhile, the application `cg.A`, which requires a much lower memory bandwidth for ideal performance (∼7GB/s), only starts to experience performance degradation beyond resource pressures of ∼14MB. Interestingly, the performance of BW applications is not affected as long as the aggregate pressure on the memory controller does not cause complete saturation of the available bandwidth. Therefore, BW class applications can be scheduled simultaneously with negligible performance loss if the individual memory bandwidths are not restricted. We observe the same trend for the CS applications. As long as the aggregate pressure does not exceed the size of the LLC, the application performance is not affected. For example, the application `2mm`, which requires an LLC occupancy of 2MB for ideal performance, experiences no IPC drop when the pressure on the LLC is under 12 MB. On the other hand, when the aggregate pressure exceeds the size of the LLC, the performance of the application is significantly affected. For example, `gemver`, which requires an allocation of 8MB in the LLC, starts to experience performance loss when the applied pressure exceeds 7MB. In general, applications that belong to the same class and have similar activity are not negatively affected if the overall pressure can be tolerated by the shared component. On any other case, the IPC loss is imminent for the co-scheduled applications. Hence, it is very important to introduce an additional feature for each class, which characterizes how much pressure each of the applications inside that class puts on the specific shared resource.

Figure 3.3 depicts the results from executing various mixes of memory-bandwidth sensitive applications. Each *mix* consists of 24 single-threaded benchmarks taken from the Polybench, Stream, and NAS benchmark suites. The right vertical axis and line plot show the average memory bandwidth achieved by each mix (in GB/s), while the left vertical axis and bar plot represent the mix's overall performance (normalized IPC), with a value of 1.0 representing alone execution without any pressure (no performance loss). We observe that there is no performance impact if multiple memory-bandwidth sensitive applications are executed simultaneously, as long as the overall pressure on the memory bus is less than its maximum sustainable value (33 GB/s in this example). In contrast, when the combined memory pressure exceeds the saturation point (mix #21 onwards), the performance of all applications is significantly affected. The same trend can be observed for the CS applications, as shown in Figure 3.4. The right vertical axis represents the total LLC occupancy of the mix (in MB) and the left vertical axis shows the overall performance (normalized IPC). When the overall pressure on the LLC is less than its maximum capacity (15.36 MB in this example), the performance of the applications is not affected. However, when the overall pressure exceeds the LLC size (mix #21 onwards), the IPC loss is significant. This is also the reason why coarse-grain classification techniques such as [72] fail to optimally schedule multiple applications. Based on the aforementioned experiments, we make the following observations:

- **Observation 1:** BW applications hurt CS applications due to high LLC pressure.

Co-execution is prohibitive.

- **Observation 2:** BW applications can be grouped together as long as the aggregate pressure on the memory controller is less than the maximum sustainable bandwidth.

- **Observation 3:** CS applications can be grouped together as long as the aggregate pressure on the LLC is less than its size.

- **Observation 4:** CNS applications increase pressure on the LLC and can possibly harm the CS applications. Co-execution is not recommended.

- **Observation 5:** N applications do not contribute to pressure on shared resources.

Regarding observation 1, we see from Fig. 3.2 and 3.4 that when the cache capacity requirement of the CS applications is not satisfied, due to interference on LLC, their IPC drops down by as much as 65%. On the other hand, BW applications typically thrash the LLC. Hence, we infer that when BW applications are executed together with CS applications, they will significantly degrade the performance of the latter. Similarly, regarding observation 4, CNS applications occupy a certain portion of the LLC, but their performance is not affected so much. Thus, putting together CS and CNS applications will affect them in unpredictable ways. The accesses on the LLC depend on the utilization pattern of each application. If a CNS application happens to thrash the LLC contents of the CS one, then the performance of the CS application will be significantly impacted. Due to this uncertainty, we recommend to not put them together (Observation 4), even though it may not have bad effects in some cases.

3.3.2 Static Pressure-Aware Scheduling Policy

Based upon the observations presented in Section 3.3.1, we propose a static pressure-aware scheduling policy. Let P be the number of processors/cores, L_{BW} be the maximum sustainable memory bandwidth, and L_{LLC} be the size of the LLC. We utilize the classification scheme presented in Section 3.3.1, where each application is assigned to a class BW, CS, CNS, or N. Let $t(a)$ indicate the number of threads of application $a \in BW \cup CS \cup CNS \cup N$, $b(a)$ indicate the memory bandwidth requirement of an application, and $c(a)$ indicate the cache occupancy requirement of an application. We want to group the applications into a given number of groups G, where the total number of application threads in each group is at most P, and each application in the group is executed concurrently with the others until its completion. The constraints that have to be respected when assigning applications to groups are as follows:

(C1) The total number of the application threads in each group should not exceed the number of cores P.

(C2) The sum of bandwidth requirements in each group should not exceed the bandwidth limit L_{BW}.

(C3) The sum of cache occupancy requirements in each group should not exceed the LLC limit L_{LLC}.

(C4) No member of class BW should be placed in the same group with a member of class CS, due to the fact that co-execution of BW and CS class applications is prohibitive given the LLC thrashing nature of BW applications and the high reliance of CS applications on LLC occupancy, as mentioned above.

We note that an individual process may do well if its number of threads exceeds P, but setting the limit in constraint C1 to a value higher than P may lead to over-subscription and thus deterioration of the system-wide throughput. We also note that applications belonging to the CNS class apply pressure to the LLC but can withstand resource interference imposed by BW applications. This makes CNS and BW applications acceptable co-runners when shared-resource pressure is high. However, since adequate LLC occupancy is critical to the performance of CS applications, co-execution with CNS type applications is allowed but may result in performance degradation.

In order to set this problem in context, we correlate it with two previously known and analyzed problems. In particular, our problem is related to the Multidimensional Bin Packing (MBP) problem [79, 80, 81] as well as the Machine Reassignment (MR) problem [82] . In the MBP problem, given a set of N items and a set of D dimensions, where each item i has weight w_{id} for dimension d, and given a set of m bins where each bin j has capacity c_{jd} for dimension d, the goal is to pack the items in the bins without exceeding the bin capacity in each dimension. In the MR problem, given a set of machines with specified resources (in terms of CPU, cache size, memory bandwidth, etc.), and a set of processes that have multiple resource requirements and are partitioned into "services", the goal is to reassign the processes to machines while respecting the resource capacity constraints, plus additional constraints (like the "service", "spread", "neighborhood", and "transient usage" constraints [82]) in order to improve the usage of the machines, as defined by a complex cost function.

Both the MBP and MR problems are NP-hard. In our problem, if we consider constraints C1, C2 and C3 only, then this becomes an instance of the MBP problem by considering the groups as bins, which have $D = 3$ dimensions (total bandwidth allowed, total LLC allowed, maximum number (P) of applications in group) and each application a has a 3-dimensional weight $(b(a), c(a), t(a))$. Also, with constraints C1, C2, and C3 only, this becomes an instance of the MR problem by considering the groups as machines (that have an additional artificial resource to account for the fact that at most P processes can be hosted by each machine).

A Mixed Integer Linear Programming (MILP) formulation for solving our problem (with constraints C1, C2, C3, and C4) is: Let x_{ij} be an integer binary variable indicating whether application i, $1 \leq i \leq Q$, where $Q = |BW \cup CS \cup CNS \cup N|$, is assigned to group j, $1 \leq j \leq G$. In this indexing, we assume that all applications are indexed consecutively starting with those in BW and followed by those in CS, CNS, and N, in that order. For each i, $1 \leq i \leq Q$, let t_i indicate the number of threads in application i, let b_i indicate the bandwidth requirement of application i (b_i is zero for applications in $CS \cup CNS \cup N$), and let c_i indicate the cache occupancy requirement for application i (c_i is zero for applications in $BW \cup N$). The constraints are formulated as follows:

$$\text{For } 1 \leq i \leq Q, \quad \sum_{j=1}^{G} x_{ij} = 1 \tag{3.1}$$

$$\text{For } 1 \leq j \leq G, \quad \sum_{i=1}^{Q} x_{ij} \cdot t_i \leq P \tag{3.2}$$

$$\text{For } 1 \leq j \leq G, \quad \sum_{i=1}^{Q} x_{ij} \cdot b_i \leq L_{BW} \tag{3.3}$$

$$\text{For } 1 \leq j \leq G, \quad \sum_{i=1}^{Q} x_{ij} \cdot c_i \leq L_{LLC} \tag{3.4}$$

$$\text{For } 1 \leq j \leq G, \text{ and } 1 \leq i \leq |BW|, \text{ and } |BW|+1 \leq \hat{i} \leq |BW|+|CS|, \quad x_{ij}+x_{\hat{i}j} \leq 1 \tag{3.5}$$

Constraint 3.1 makes sure that every application is assigned exactly to one group. Constraint 3.2 makes sure that each group has at most P application threads. Constraint 3.3 makes sure that the memory bandwidth bound is satisfied in each group. Constraint 3.4 makes sure that the LLC bound is satisfied in each group. Constraint 3.5 makes sure that no group contains both a BW and a CS application.

For a user-specified value of G, the above MILP, referred to as $MILPSS(G)$ ("MILP Static Scheduler"), determines whether a grouping can be done that satisfies all the constraints. (We note that this MILP does not have an objective minimization or maximization function, rather it has a "token" objective, since the aim is to satisfy all constraints for the given value of G). We are interested in minimizing the number of groups G, since the total execution time of the applications in the application mix to be scheduled

Algorithm 1 Static Scheduling Policy

```
1: Inputs: Applications in classes BW, CS, CNS, N, each with number of threads $t(a)$, $a \in BW \cup CS \cup CNS \cup N$, memory bandwidth $b(a)$, $a \in BW$, cache occupancy $c(a)$, $a \in CS \cup CNS$; $P$: Total number of cores; bandwidth limit $L_{BW}$; LLC limit $L_{LLC}$.
2: Output: Assignment of applications into the minimum number of groups respecting limits $P, L_{BW}, L_{LLC}$.
3: Initialize $L \leftarrow 1$
4: Initialize $U \leftarrow Q = |BW \cup CS \cup CNS \cup N|$
5: Initialize $s = \{\}$
6: while $L \leq U$ do
7:     $G \leftarrow \lfloor (L+U)/2 \rfloor$
8:     if MILPSS($G$) (i.e., constraints in Eqs. (1)–(5)) is satisfiable then
9:         $s \leftarrow$ assignment found by MILPSS($G$)
10:        $U = G - 1$
11:    else $L = G + 1$
12: if $s \neq \{\}$ then
13:     return $s$
14: else report infeasibility
```

is thus minimized and the utilization of the cores is maximized. Given that $MILPSS(G)$ provides a definite "yes or no" answer for a specific value of G, the minimum number of groups can be found by a binary search on the values of G, running the $MILPSS(G)$ for each value, and keeping the minimum G for which the $MILPSS(G)$ reports a solution. The range of the values of G is from 1 to the total number Q of applications to be scheduled (we note that the minimum value of G may actually be equal to Q in the case where, e.g., every application in the mix saturates one of the bounds P, L_{BW}, L_{LLC}). The overall pseudocode of the proposed static scheduler is given as Algorithm 1. Finally, we note that despite the general exponential time complexity of the MILP formulation, for the sizes of the application mixes that occur in practice, the proposed static scheduler finds the minimum number of groups very fast.

3.3.3 Dynamic Pressure-Aware Scheduling Policy

In this section, we present a dynamic resource-aware static scheduling policy based on the observations presented in Section 3.3.1. The dynamic scheduler operates in periods of pre-defined time duration ("quanta") and consists of two phases: *Application Characterization*, and *Application-to-Core Assignment.* A high-level flowchart of the

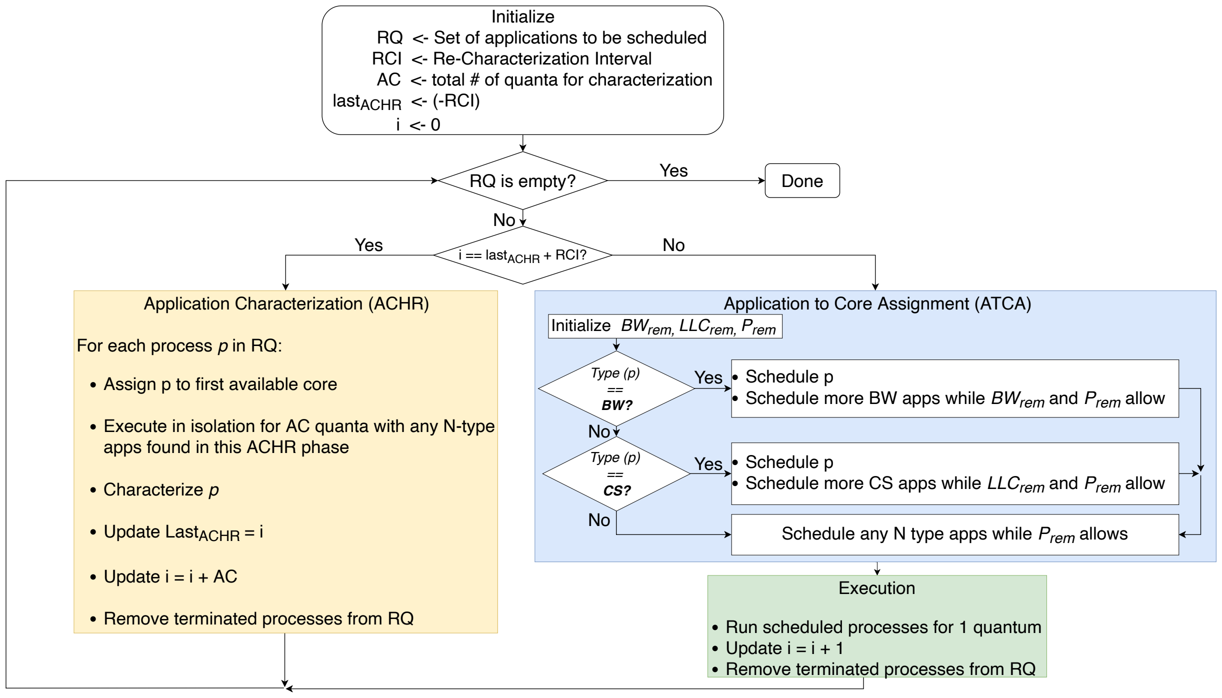

Initialize
RQ <- Set of applications to be scheduled
RCI <- Re-Characterization Interval
AC <- total # of quanta for characterization
last_ACHR <- (-RCI)
i <- 0
RQ is empty?
Yes
Done
No
i == last_ACHR + RCI?
Yes
No
Application Characterization (ACHR)
For each process p in RQ:
Assign p to first available core
Execute in isolation for AC quanta with any N-type apps found in this ACHR phase
Characterize p
Update Last_ACHR = i
Update i = i + AC
Remove terminated processes from RQ
Application to Core Assignment (ATCA)
Initialize BW_rem, LLC_rem, P_rem
Type (p) == BW?
Yes
Schedule p
Schedule more BW apps while BW_rem and P_rem allow
No
Type (p) == CS?
Yes
Schedule p
Schedule more CS apps while LLC_rem and P_rem allow
No
Schedule any N type apps while P_rem allows
Execution
Run scheduled processes for 1 quantum
Update i = i + 1
Remove terminated processes from RQ

Figure 3.5: Flowchart for Dynamic Pressure-Aware Scheduling Policy

scheduler is given in Fig. 3.5 and a more detailed pseudocode is given as Algorithm 2.

Application Characterization phase

The *application characterization (ACHR)* phase (lines 8–22 of Algorithm 2) is entered initially and also every RCI quanta after the last ACHR phase, where RCI ("recharacterization interval") is a user-defined parameter. Each ACHR phase lasts for AC quanta, where $AC < RCI$ is another user-defined parameter. The variable $last_{ACHR}$ is initialized to $-RCI$ so that the ACHR phase is necessarily entered at the first quantum $i = 0$. The purpose of the ACHR phase is to (re)determine the class type (BW, CS, N) of each application based on dynamically obtained measurements about its resource usage (note: the dynamic scheduler treats CNS applications as belonging to the CS or N class). The resource utilization of an application can be estimated more accurately when there is negligible resource interference from the neighboring cores. We accomplish this by enforcing a light-weight contention scenario in which one core is allocated to each specific application $\hat{p}$ in turn in the run-queue (RQ), while the rest of the cores are allocated to other applications (if any) that have been already characterized (in this ACHR phase) as belonging to the N class. To maintain fairness, the latter applications are furthermore selected based on their progress so far (as defined in [20]), given that all processes are sorted in increasing order of progress the beginning of the quantum (lines 6-7 of Algorithm 2). We note that in the very first time that the ACHR phase is entered, all processes have an empty type, and therefore each application $\hat{p}$ whose behavior is to be characterized is run on a single core with all other cores remaining idle. The above process assignment is run for $AC < RCI$ quanta (line 15) and any processes that have terminated are removed from RQ (line 16). Then, we record the IPC ($IPC_{alone}(p)$) (line 17) of every executed application p that is still in RQ, and in addition, for the specific application $\hat{p}$ whose behavior is being characterized, we record (if it has not terminated) its main memory bandwidth usage ($usg_{BW}(\hat{p})$), and LLC utilization ($usg_{LLC}(\hat{p})$) (line 19). Depending on whether the usages are greater than a user-specified categorization bound of CTG_{BW} for main memory bandwidth or CTG_{LLC} for LLC, we assign the type of $\hat{p}$ to one of BW, CS, or N lines (20-22).

Finally, the RCI phase must be long enough so that a stable measurement of each available application's resource utilization can be taken. In this work, we considered 30 time quanta of $200ms$ each. These values were extracted experimentally so that the overhead is not big enough to obscure the application execution, and at the same time the dynamic algorithm has enough time to decide the application class type. A more thorough investigation of these parameters is part of our future work. Additionally, regarding the user categorized bounds for CTG_{BW} and CTG_{LLC}, we considered $2GB/s$ as the threshold value of memory read bandwidth and $1MB$ for LLC occupancy accordingly.

Algorithm 2 Proposed dynamic scheduling policy

```
1: Initialize $last_{ACHR} = -RCI$
2: Initialize $i = 0$
3: $RQ \leftarrow$ set of applications in the mix to be scheduled
4: for every process $p \in RQ$, initialize $IPC_{co-running}(p) = 0$, $IPC_{alone}(p) = 1$, $Progress(p) = 0$,
   $type(p) = ' '$
5: while $RQ \neq \{\}$ do
6:     for every process $p \in RQ$ compute $Progress(p)$ += $IPC_{co-running}(p)/IPC_{alone}(p)$
7:     Sort $RQ$ in ascending order based on $Progress(p)$
8:     if $i == last_{ACHR} + RCI$ then
9:         for each $\hat{p} \in RQ$ do
10:            Initialize $P_{rem} = P$
11:            Assign $\hat{p}$ to the first core(s) and update $P_{rem}$ −= $t(\hat{p})$
12:            while $P_{rem} > 0$ do
13:                Select the first process $p \in RQ$ with $type(p) = 'N'$ and $t(p) \leq P_{rem}$
14:                Assign $p$ to the next available core(s) and update $P_{rem}$ −= $t(p)$
15:            Run the assignment above for $AC$ quanta and update $last_{ACHR} = i$, $i$ += $AC$
16:            Remove any process that has terminated from $RQ$
17:            Update the $IPC_{co-running}(p)$ of every executed process $p$ in this assignment that is still in
   $RQ$
18:            if $\hat{p} \in RQ$ then
19:                Record the usage $usg_{BW}(\hat{p}), usg_{LLC}(\hat{p})$ and the $IPC_{alone}(\hat{p})$ of $\hat{p}$.
20:                if $usg_{BW}(\hat{p}) > CTG_{BW}$ then $type(p) = 'BW'$
21:                else if $usg_{LLC}(\hat{p}) > CTG_{LLC}$ then $type(\hat{p}) = 'CS'$
22:                else $type(\hat{p}) = 'N'$
23:    else
24:        Initialize $CS_{on} = BW_{on} = False$
25:        Initialize $BW_{rem} = L_{BW}$, $LLC_{rem} = L_{LLC}$, $P_{rem} = P$
26:        for each $p \in RQ$ do
27:            if $type(p) == 'BW'$ and not $CS_{on}$ and $usg_{BW}(p) \leq BW_{rem}$ and $t(p) \leq P_{rem}$ then
28:                $BW_{Remain}$ −= $usg_{BW}(p)$, $P_{rem}$ −= $t(p)$, $BW_{on} = True$
29:            else if $type(p) == 'CS'$ and not $BW_{on}$ and $usg_{LLC}(p) \leq LLC_{Rem}$ and $t(p) \leq P_{rem}$
   then
30:                $LLC_{Remain}$ −= $usg_{LLC}(p)$, $P_{rem}$ −= $t(p)$, $LLC_{on} = True$
31:            else if $type(p) == 'N'$ and $t(p) \leq P_{rem}$ then
32:                $P_{rem}$ −= $t(p)$
33:        Run the assignment above for 1 quantum and update $i$ += 1
34:        Remove any process that has terminated from $RQ$
35:        Update the $IPC_{co-running}(p)$ of every executed process $p$ in this assignment that is still in $RQ$
```

Application-to-Core Assignment phase:

The *application-to-core assignment* (ATCA) phase (lines 24–35 of Algorithm 2) is triggered at every quantum when ACHR is not in effect. It estimates the progress of each application p by using the ratio of $IPC_{co-running}(p)$ over $IPC_{alone}(p)$. We utilized the term progress for this ratio, as this is how it has been used in the bibliography [20, 52, 69]. While an application is executed, this ratio changes based on the co-runners and the number of executed instructions. Thus, it shows how efficiently an application utilized the time that was assigned to the CPU in comparison to running alone [20] . Based on progress the algorithm assigns a process to a core or cores (depending on how many threads $t(p)$ the process inherently requires) based on the type $type(p)$ of the process and on the remaining resources available. (Note: the type, usage, and IPC_{alone} have been determined in the last ACHR phase, whereas the $IPC_{co-running}$ is determined by the ACHR initially and then updated by the ATCA phase). Unfairness is tackled by prioritizing applications that made the least progress in the workload [20]. This is done in line 6 which is also useful for the selection of any N class processes during the ACHR phase. We note that for single-phase application, the IPC does not fluctuate a lot between time-quanta. Thus, we utilize the average IPC of all previous quanta to make the subsequent scheduling decisions. Regarding multi-phase applications, we observed that any change on the IPC lasts for many RCI intervals (i.e., no sudden peaks or drops), thus we are able to record the change and take necessary actions.

Co-execution of BW with CS is avoided by introducing two flags BW_{on} and CS_{on} (line 24) and making sure that in every iteration of the application selection process, only one flag is true at a time. Namely, if BW_{on} is true, CS applications are not allowed to be assigned to the CPU in the current quantum and vice versa (lines 28 and 30). The rest of the applications for the current quantum are selected according to their progress order so far (lines 6-7) but also based on their resource requirements and the available provided resources. In the beginning of the quantum, the available resources BW_{rem}, LLC_{rem}, and P_{rem} for the main memory bandwidth, LLC and number of cores respectively, are initialized to their maximum values L_{BW}, L_{LLC} and P, respectively (line 25). Once the application is selected, its usage requirements ($usg_{BW}()$, $usg_{LLC}()$) determined in the ACHR phase are subtracted from the remaining resources available and the available cores are decreased by the number of threads that the application requires (lines 28–30). If an application belongs to the N class, it is directly allocated if it fits into the available cores (line 32), as it does not contribute to contention, as mentioned in Section 3.3.1. Finally, the above process assignment is run for one quantum (line 33), any processes that have terminated are removed from RQ (line 34) and the $IPC_{co-running}()$ of every executed process in this assignment that is still in RQ is updated (line 35).

3.4 EXPERIMENTAL RESULTS

This section describes the experimental setup and evaluates results obtained from testing various single and multi-threaded workloads executed using our proposed policies and four other resource-aware schedulers.

3.4.1 Experimental setup

The proposed scheduling policies are validated with extensive experimentation on two distinct commercially-available Chip Multiprocessors. This is done to ensure robustness of acquired results as well as to demonstrate that the proposed policies can be utilized on any processor that allows access to performance metrics. Table 3.3 shows the characteristics of each system. On both systems, we utilize one socket for application execution in order to provide run-time isolation from kernel threads. Additionally, we disable Hyperthreading and Turbo Boost Technology for reducing the impact of non-determinism in the acquired results. For completeness, a discussion of application performance with Hyperthreading enabled is presented in Section 3.4.3.

The resource utilization of applications is monitored by accessing specialized hardware registers ('performance counters') built into the Performance Monitoring Unit (PMU) of the processor. Specifically, we access the x86 architecture's Model Specific Register `MSR_OFFCORE_RSP_0` using the MSR interface for Linux to monitor the main memory bandwidth (as part of Intel Resource Director Technology's Memory Bandwidth Monitoring (MBM) feature) . In addition to per-core memory bandwidth, we record the counts of the architectural events `L3_CACHE_OCCUPANCY`, `L2_LINES_IN.ALL`, `INSTR_RETIRED`, UNHLT_CORE_CYCLES, and LLC_MISSES. The time taken for accessing each PMC event is 2-5 μs, which is factored into the calculation of total scheduling overhead for evaluated approaches. The scheduling policies are implemented in user-space, leveraging the cpuset subsystem provided by the cgroup mechanism in Linux kernel for isolating the socket and pinning processes to cores at run-time. Program execution is divided into discrete time quanta of predefined duration, which can be adjusted by the user prior to execution. We utilize quanta of duration $200ms$ as it provides a balance between accurate resource-monitoring and low scheduling overhead for the utilized platform. At the end of each time quantum, performance metrics for each application are updated and applications are assigned to cores based upon the decisions made by the relevant scheduling policy. Regarding the run-time overhead of the two policies, the static scheduler does not have any as all groups have been formed offline and the scheduler does not perform any additional actions at run-time. Regarding the dynamic scheduler, two types of overhead exist: (i) initial profiling overhead (30 time-quanta of $200ms$ per application), and (ii) per-quanta overhead of collecting resource-usage statistics during execution. The cost of accessing each performance counter described before is between $40us$ to $80us$, which together with the decision-making algorithm sums to an average overhead of about $1ms$ per time quanta. Therefore, the overhead accounts for 0.5% of execution time between successive quanta. Both overheads are accounted in the presented experiments. Finally, we utilized the lpsolve tool [83] for solving the MILP problem. Since the sizes of the application mixes that occur in practice for our evaluation are small (order of tens), the proposed static scheduler finds the minimum number of groups very fast.

We utilize applications taken from the STREAM [74] , Polybench [76], and NAS [75] benchmark suites to construct various application *mixes* that exert different levels of pressure on the system's shared resources. For evaluating single-threaded performance each application mix (Mix1_S – Mix8_S) has 24 and 48 single-threaded applications for Server 1 and 2 accordingly, and for evaluating multi-threaded performance each application mix

	Server 1	**Server 2**
System	Intel Xeon E5-2620 v3	Intel Xeon Gold 6130
Frequency	2.40 GHz	2.10 GHz
#Cores/socket	6	16
L2	Unified 256 KB	Unified 1024 KB
LLC size	15,360 KB	22,528 KB
Memory	256 GB DDR4	128 GB DDR4

Table 3.3: Server characteristics used for the evaluation.

(Mix1_M – Mix8_M) has 12 and 32 multi-threaded applications with either 1, 2, or 4 threads for each server respectively. The goal is to evaluate scheduling performance over the entire possible spectrum of shared-resource utilization of the system.

Evaluated Schedulers

Each application mix is executed using our proposed scheduling policies, and the results are evaluated against the following resource-aware scheduling approaches:

1. the Linux CFS (Completely Fair Scheduler), which acts as the baseline for the comparison of results;

2. the Distributed Intensity (DI) [47] contention-aware scheduler, which utilizes cache misses per million instructions as a heuristic for quantifying contention;

3. the Link and Cache-Aware (LCA) [72], [48] scheduler, which tracks the bandwidth between links of the memory hierarchy to capture the activity of the applications, identifies the cache-intensive ones, and avoids co-executing them with applications that heavily thrash the LLC;

4. the BAOS scheduler [22], which calculates the overall main memory requests and the average main memory bandwidth utilization for each application per quantum. The application with the best fit in terms of bandwidth is selected to run during the following quantum; and

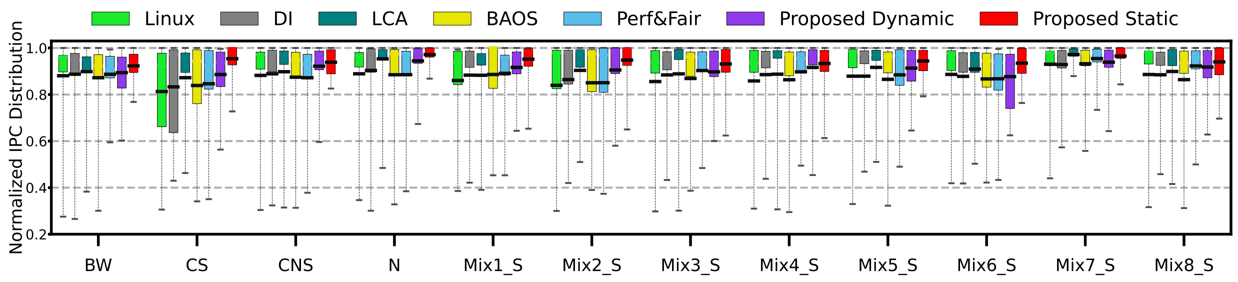


Figure 3.6: IPC distribution for single threaded workload mixes for Server 1

5. the Perf&Fair scheduler [52], which simultaneously addresses performance and fairness of applications by monitoring their main memory bandwidth, L1 bandwidth, and progress. An online average of the two metrics is then used as part of a fitness function to schedule applications for each time quantum.

Experimental results are quantified by each application mix's IPC distribution, total throughput, and overall fairness when executed with the respective schedulers. The achieved throughput is calculated using the summed `Weighted Speedup` of each application, which is defined as $WS = \sum_{i=1}^{n} \frac{IPC^{i}_{co-execution}}{IPC^{i}_{alone}}$, where N is the number of applications in the mix, $IPC^{i}_{co-execution}$ is the IPC of application i when executing concurrently in the mix, and IPC^{i}_{alone} is the IPC achieved during standalone execution (i.e., the ideal case when all shared-resources of the system are available to application i). Consequently, higher values of weighted speedup signify lower levels of performance loss from shared-resource contention. The total throughput of the mix is then computed as $Throughput = \sum_{i=1}^{n} WS_i$, where N is the number of applications in the workload and WS_i is the weighted speedup of each application i. Fairness describes the gap between the best and worst performance achieved by the applications in a workload and it is defined as $Fairness = \frac{\max\{WS_1, WS_2, \cdots, WS_n\}}{\min\{WS_1, WS_2, \cdots, WS_n\}}$, where N is the number of applications in the workload and WS_i is the weighted speedup of each application i.

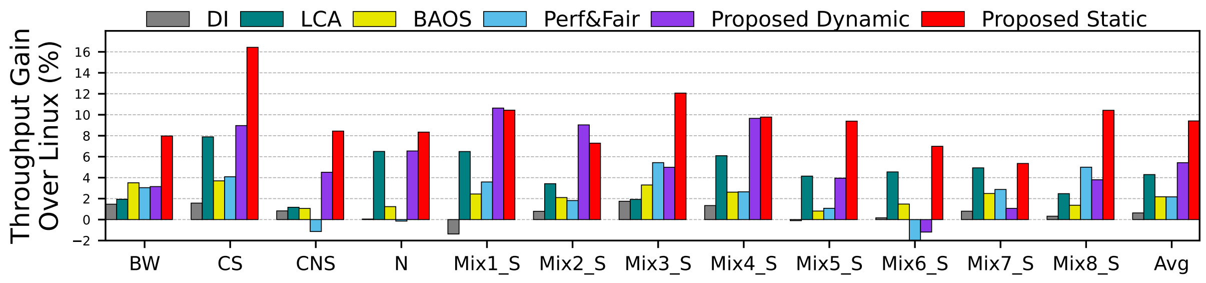


Figure 3.7: Throughput gain over Linux for single-threaded workloads for Server 1

3.4.2 Single-Threaded Workload Performance

In this section, we evaluate the results obtained from executing various single-threaded application mixes. We execute 8 random mixes (Mix1_S to Mix8_S) arranged in decreasing order of shared-resource pressure inflicted on the socket. Moreover, on Server 1 we provide a detailed analysis for four additional mixes with specific distributions dominated by each of the identified application classes in Section 3.3.1: *BW-dominated workload*: applications from the *Memory bandwidth sensitive* class dominate the mix with ratio 3(BW):1(CS):1(CNS):1(N); *CS-dominated workload*: applications from the *LLC sensitive* class dominate the mix with ratio 1(BW):3(CS):1(CNS):1(N); *CNS-dominated workload*: applications from the *Memory bandwidth non-sensitive* class dominate the mix with ratio 1(BW):1(CS):3(CNS):1(N); and *N-dominated workload*: applications from the *Neutral* class dominate the mix with ratio 1(BW):1(CS):1(CNS):3(N).

Figure 3.6 depicts the average value and the distribution of the normalized IPC for each evaluated scenario for Server 1. The proposed static methodology achieves the best average IPC for all mixes compared to other scheduling methods. The proposed dynamic scheduler achieves the second highest average IPC, except for mixes 6 and 7 where it is outperformed by 3% (LCA) and 3.4%,1.7% (LCA, Perf&Fair) respectively. Furthermore, our proposed policies achieved the lowest and second lowest average standard deviation among all schedulers. The next lowest standard deviation was achieved by Perf&Fair (0.151), followed by LCA (0.154), DI (0.184), BAOS (0.189), and Linux (0.195).

Figure 3.7 depicts the gain in throughput achieved by each scheduler compared to

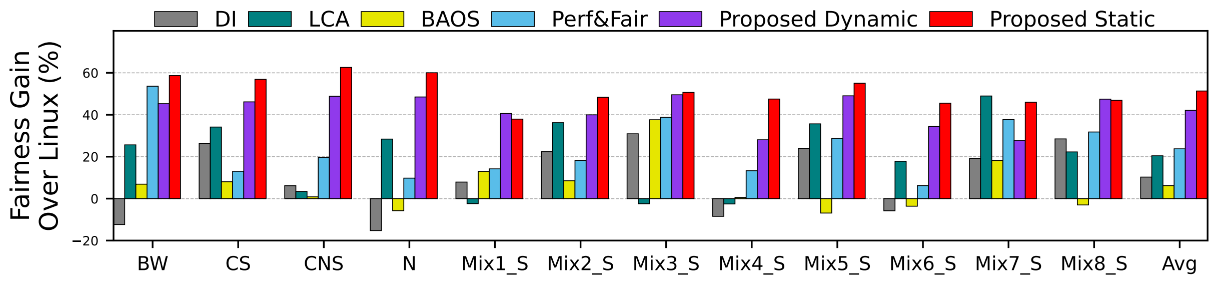


Figure 3.8: Fairness gain over Linux for single-threaded workloads for Server 1

CFS for single-threaded applications for Server 1. In all scenarios, the proposed static scheduler achieves the highest throughput gain, except for Mix1_S and Mix2_S where the proposed dynamic scheduler achieves 0.2% and 1.7% higher throughput respectively.

On average, the proposed static scheduler achieved a throughput gain of 9.4%, proposed dynamic 5.4%, LCA 4.3% , BAOS and Perf&Fair both 2.2% , and DI 0.6%. Finally, figure 3.8 depicts the gain in fairness over CFS for single threaded application mixes for Server 1. On average the proposed static scheduler achieved the highest fairness gain 51.3%, followed by proposed dynamic 42.1%, Perf&Fair 23.7%, LCA 20.4%, DI 10.2%, and BAOS 6.2%. Among the evaluated schedulers, only proposed dynamic and Perf&Fair account for run-time progress of applications.

Figure 3.9 depicts the normalized IPC for each application for the BW-dominated workload, CS-dominated workload, CNS-dominated workload, and N-dominated workload for Server 1. As baseline, we considered the IPC of each application when running alone. For the BW-dominated workload (Figure 3.9 (a)), all policies except for the proposed static and proposed dynamic achieved overall low performance for the CS applications. Our proposed approaches managed to capture the sensitivity of cache intensive applications and wisely isolated the CS from the LLC thrashing nature of BW. This resulted in high performance benefits for the CS applications for both the static and dynamic policies {70%,40%} for `chase` and {40%,40%} for `syr2k` compared to Linux CFS). LCA also offers isolation for cache intensive applications but treats each application uniformly. For this reason it made the oblivious choice to co-schedule `chase` and `syr2k` with the harmful BW co-runners, leading to a significant degradation in performance. In contrast, the proposed static policy selected the CNS applications for co-execution, which suffer much lower degradation (20% at most) compared to CS. The proposed dynamic policy achieves performance comparable to the static scheduler for all CS applications (within 2.6%) except for `chase` where it achieves 54.8% lower IPC. This is because the dynamic scheduler cannot detect and schedule the CNS applications and thus, it is unable to select an ideal co-runner for one of the CS applications in the given mix (`chase` in this case). The Perf&Fair scheduler achieves good performance for `chase` as it considers both the performance and progress of each application. However, it is unable to achieve consistent performance across all CS applications as it only monitors the resource pressure for main memory and L1 bandwidth, unlike our proposed schedulers that account for resource pressure at all levels of the memory hierarchy.

Furthermore, it is noteworthy to mention that the proposed static and dynamic policies experienced average performance losses of only 6% (same as CFS) and 9%, for the BW class, whereas LCA experienced a loss of 9%. These three policies group together the BW applications in order to avoid co-execution with the cache intensive ones. This performance gap occurs due to the pressure-unaware placement of LCA. Note that our proposed dynamic methodology is able to achieve similar BW performance level as LCA (within 2%), despite not requiring the characteristics of each application in the workload to be known in advance. Overall, despite the heavy contention environment in both memory bandwidth and the LLC, both proposed policies managed to accomplish isolation of the CS applications and keep the BW ones at a good performance level at the same time. The average IPC drop across all applications in this mix is 12.5% for the Linux CFS, 10.8% for the DI, 8.9% for the LCA, 10.3% for the BAOS, and only 7% and 9% for the proposed static and dynamic methodologies respectively. Additionally, the maximum IPC loss (worst case) for the proposed static methodology is only 22%, and for the dynamic is 40%. The next lowest IPC loss was experienced by Perf&Fair (60%) followed by BAOS (70.8%).

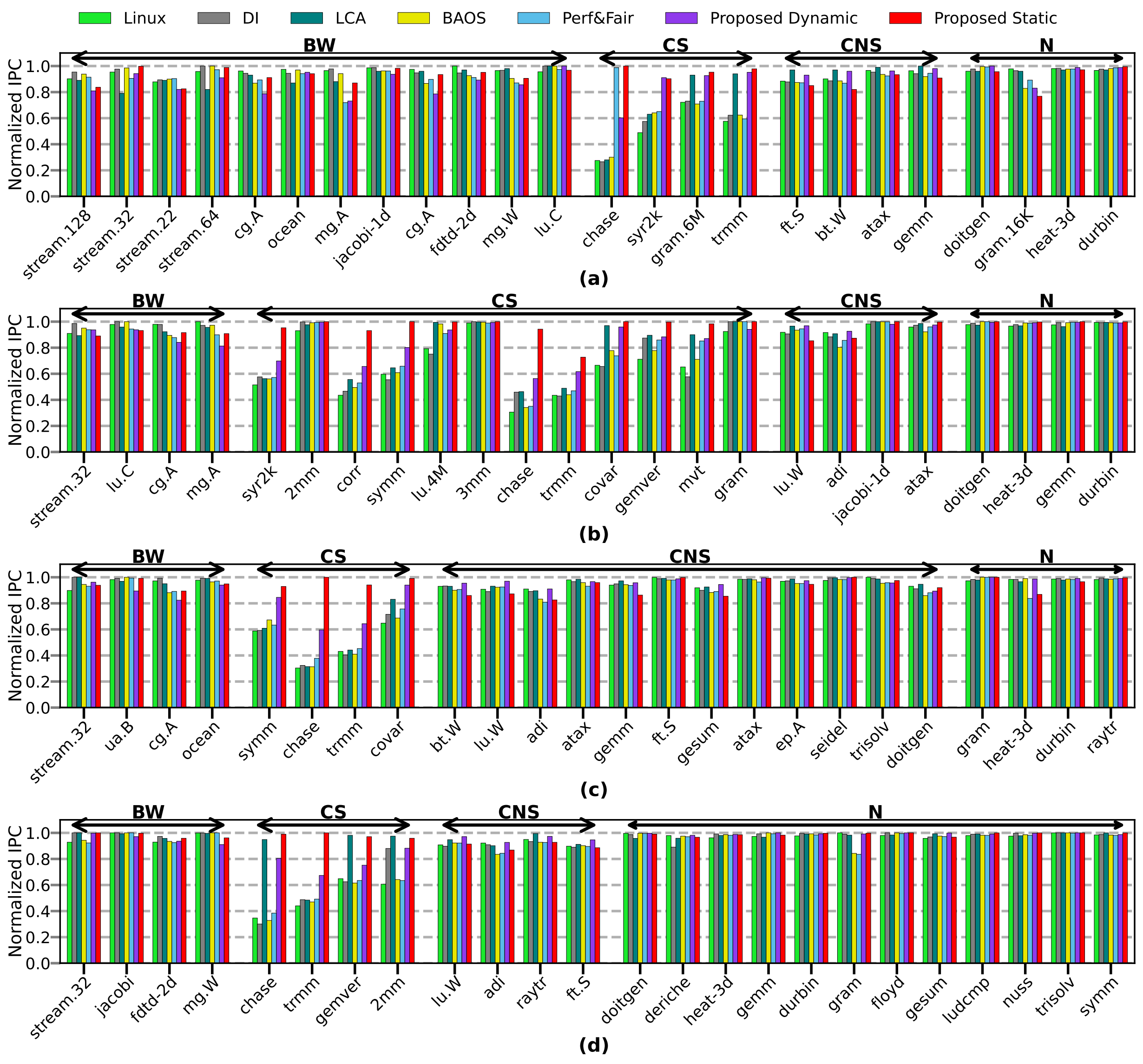


Figure 3.9: Detailed performance for BW/CS/CNS/N-dominated mixes: (a) BW-dominated; (b) CS-dominated; (c) CNS-dominated; and (d) N-dominated for Server 1

In the CS-dominated workload (Figure 3.9 (b)), we observe that our policies restrict the BW applications in a group in order to spread the dominant CS ones as much as possible. Even though this hurts the BW class, where four applications experience an average IPC drop of nearly 12% (static) and 18% (dynamic), it is compensated by the 22.8% (static) and 4.9% (dynamic) IPC increase of the CS class compared to the second highest (LCA). This gap proves that avoiding LLC interference from the streaming

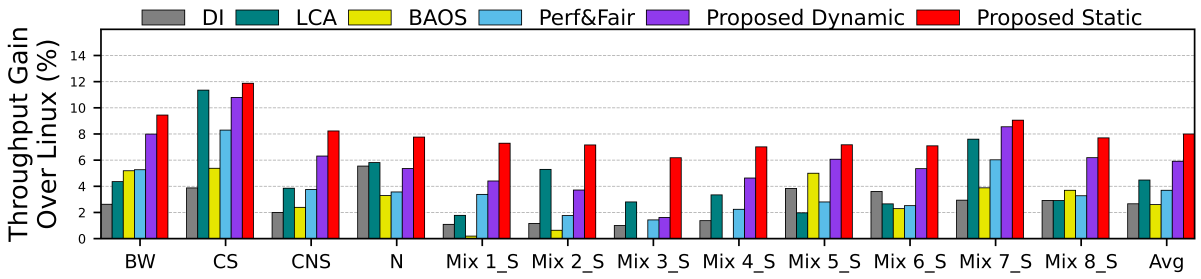


Figure 3.10: Throughput gain over Linux for single-threaded workloads on Server 2.

applications is not sufficient (LCA). The Perf&Fair scheduler is able to achieve high IPC (∼ 1) for three CS application (`2mm`,`3mm`, and `gram`), but on average performs 4.4% worse than LCA for the entire class, due to its focus on fairness in addition to performance. For DI and BAOS, we observe that they are able to maximize performance for BW class but they are again proved insufficient for the CS class. They showed behavior similar to CFS with an average improvement of 0.6% and 2.86% respectively for BW class. LCA reacted better depicting a gain of 6.26%, followed by Perf&Fair 8.5%, whereas the proposed methods achieved the highest IPC increase of 8.87% (static) and 11.8% (dynamic) over CFS. It is worth mentioning that DI and BAOS converge to CFS, because contention for memory bandwidth is really low in this workload. As a result the contention-agnostic placement of the few (only four) BW applications by the Linux scheduler achieves high performance and there is no room for further improvement by the DI and the BAOS. Overall, our policies scored the highest average IPC of 95.7% (static) and 88.6% (dynamic), bridging the performance gap between applications at the same time (minimum IPC of 72.7% (static) and 56.3% (dynamic)), with the second best policy (LCA) reaching 87.2% and 46.2% respectively.

Examining the CNS-dominated workload (Figure 3.9 (c)), we observe that our policy provides the lowest contended environment for the CS class reaching performance level of 96.9% (static) and 75.6% (dynamic) on average, whereas LCA achieves 54.8%, and Perf&Fair achieves 55.5%. Although the dominant CNS applications contribute to LLC

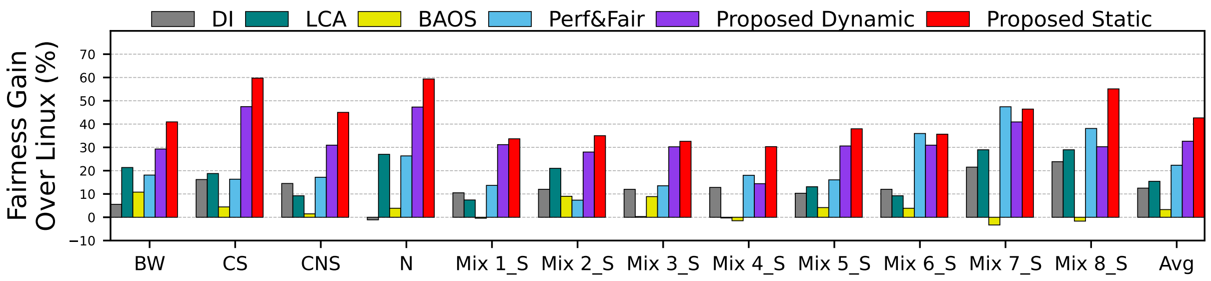


Figure 3.11: Fairness gain over Linux for single-threaded workloads on Server 2.

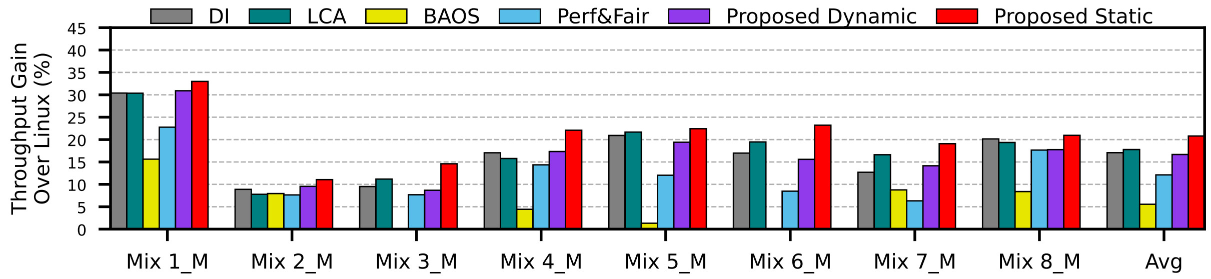


Figure 3.12: Throughput gain over Linux for multi-threaded workloads on Server 2.

pressure, our policy is able to wisely select the groups and avoid excessive contention for CS applications. It co-schedules a part of them with BW applications and leaves the N applications to be co-executed with CS ones, whereas LCA selected the opposite. For this reason, we notice a slight IPC drop for some CNS applications (`lu.w`, `raytrace`), which is on average 2.8% worse than the highest (LCA). Regarding DI and BAOS, we observe similar behavior with CFS, as they fail to address contention for cache-intensive applications.

In the N-dominated workload (Figure 3.9 (d)), resource interference among the application of the workload is low. However, CS applications still get severely degraded under the CFS, DI, BAOS, and Perf&Fair policies. This happens because all the four scheduling policies mix the CS applications with the LLC thrashing BW ones.

Our policy prioritizes the N to be co-executed with CS and dedicates the appropriate number of groups for each class, balancing their pressure simultaneously. Overall, our static policy achieves the best average IPC of 97.9%, followed by our proposed dynamic policy

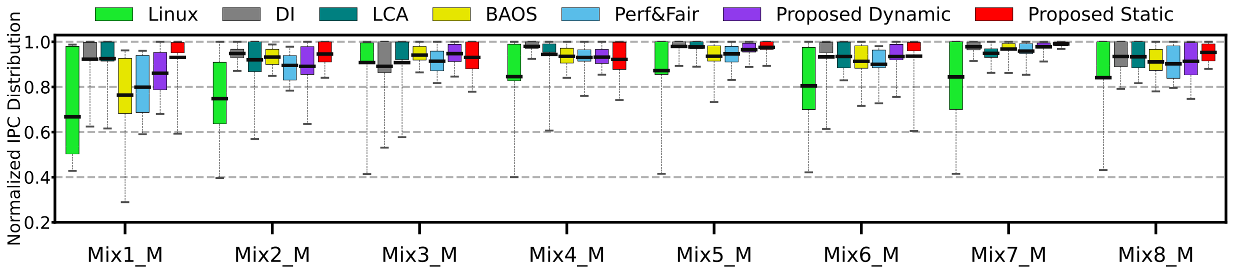


Figure 3.13: IPC distribution for multi-threaded workload mixes on Server 1

and LCA with 84.7% average IPC.

For Server 2, we obtained the results shown in Figures 3.10 and 3.11 for throughput and fairness, respectively. The results follow the same trend as in the previous server with the proposed static scheduler exhibiting the best performance with an average gain of 8%, while the proposed dynamic scheduler is the second best, validating the previous analysis. At this point, we would like to mention that although Server 2 supports Intel's Cache Allocation Technology (CAT), CAT was not utilized as the focus of this work is on contention in the LLC (in addition to main memory bandwidth) by monitoring the cache utilization and by subsequently performing sophisticated application-to-core assignment. Leveraging CAT synergistically increases the complexity and is left as part of future research.

3.4.3 Multi-Threaded Workload Performance

Figure 3.13 depicts the average value and the distribution of the normalized IPC in each evaluated multi-threaded scenario. The average standard deviation achieved by by DI, BAOS, Perf&Fair, proposed dynamic, and proposed static was 0.08, followed by LCA (0.09), and Linux (0.21).

Figure 3.14 shows the gain in throughput achieved by each scheduler for eight random mixes of multi-threaded applications for Server 1. The proposed static scheduler achieves the highest throughput gain in all mixes, except for mix 4 where DI performs 4% better. It

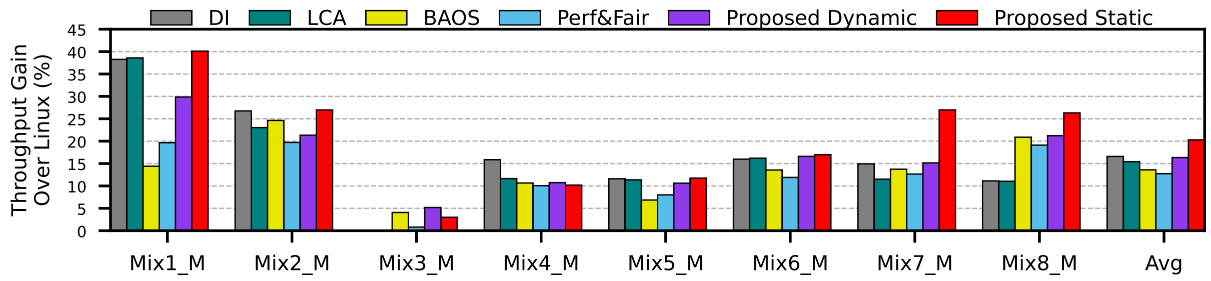


Figure 3.14: Throughput gain over Linux for multi-threaded workloads on Server 1

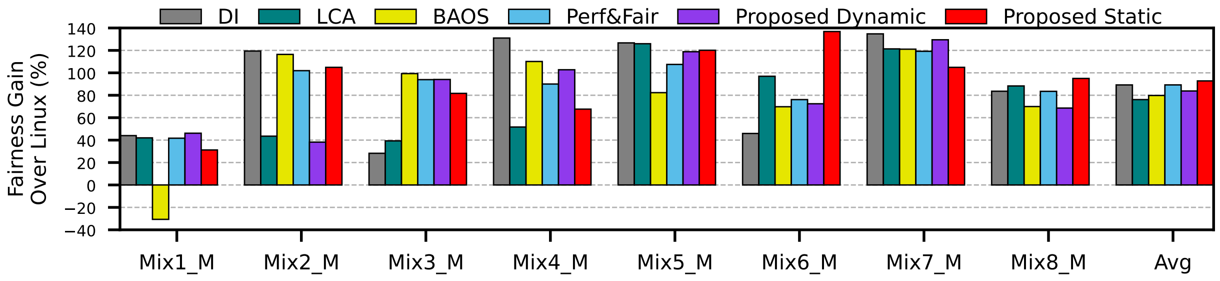


Figure 3.15: Fairness gain over Linux for multi-threaded workloads on Server 1

is able to outperform other schedulers significantly for mixes 7 and 8 because they consist of a large number of CNS applications that the static scheduler can schedule alongside the more resource intensive applications. On average, our proposed static scheduler achieved the highest throughput gain of 20.3%, followed by our proposed dynamic scheduler 16.4%, DI 16.4%, LCA 15.4%, BAOS 13.6%, and Perf&Fair 12.8%. Overall, there is less throughput variation in the multi-threaded case because there are fewer possible combinations of applications that can run concurrently. For effective utilization of multi-threading, all threads of an application must run during the same time quanta. However, this also decreases the possible choice of schedules since the number of threads during each time quantum must add up to a fixed value. similarly, Figure 3.15 depicts the *fairness* over CFS of all the scheduling approaches for Server 1. In contrast to the single-threaded case, there is less variation in the fairness gain compared to the single threaded case. This occurs because there are fewer possible combinations of applications that can be created. The average gain on fairness (in descending order) was proposed static

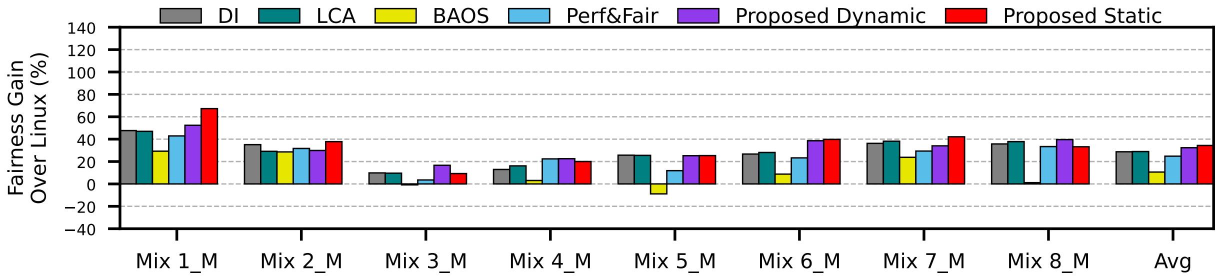


Figure 3.16: Fairness gain over Linux for multi-threaded workloads on Server 2.

92.8%, Perf&Fair and DI both 89.2%, proposed dynamic 83.8%, BAOS 79.8%, and LCA 76.1%. Multithreaded application execution exhibits similar trends on Server 2. Figures 3.12 and 3.16 show the throughput and fairness gains obtained by the schedulers over Linux.

3.4.4 Enabling hyper-threading

Figure 3.17 depicts the throughput improvement over Linux for both Server 1 and Server 2 when hyper-threading technology is enabled. The BW, CS, CNS, and N dominated mixes correspond to the ones presented in Figures 3.7 and 3.10 for Servers 1 and 2 respectively. Hyper-threading addresses each available core as two 'virtual'/'logical' cores which divide the computational workload among parallel execution units. Since the Linux scheduler is not resource-aware, it schedules processes on every available core of the system regardless of the total shared-resource pressure exerted by the workload, leading to an overall drop in performance. On the other hand, the proposed resource-aware schedulers are able to maintain throughput levels similar to the previous analysis by leaving certain cores unutilized whenever the exerted resource pressure does not allow for further performance gains. For Server 1, the proposed static and dynamic schedulers yielded an additional 9.3% and 8.8% average increase in throughput over the non-hyperthreaded case. For Server 2, the throughput increase is even higher due to the significantly larger number of cores made available to the schedulers. The resultant increase in throughput over the non-Hyperthreaded case is 17.3% and 13.1% on average for the static and dynamic schedulers respectively. Note that the proposed schedulers are able to outperform the

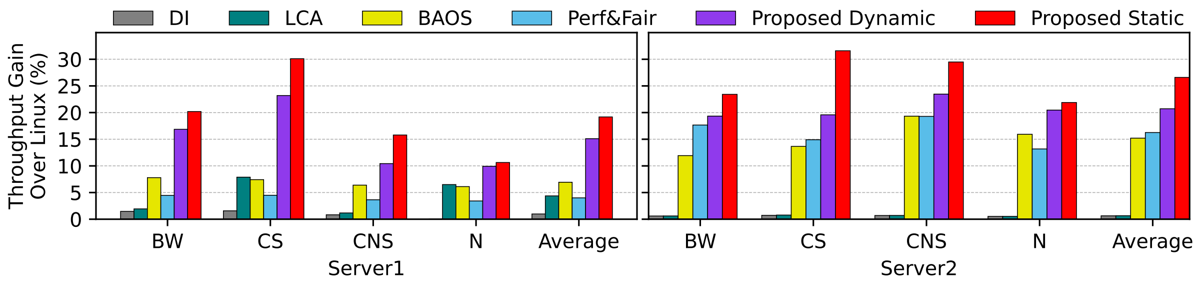


Figure 3.17: Enabling hyper-threading results for Server 1 and Server 2

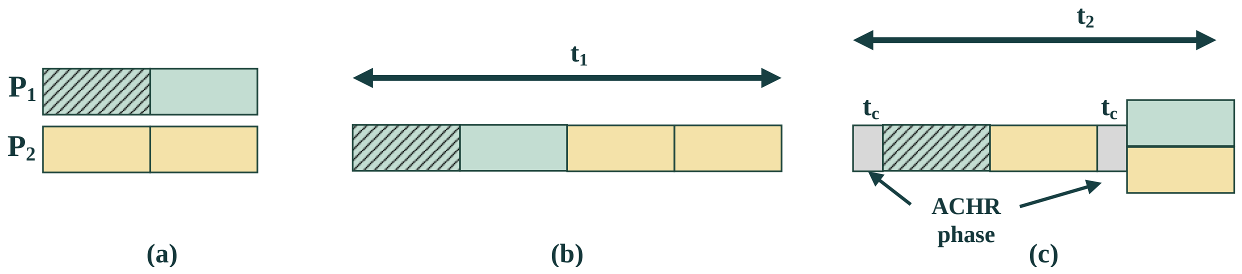


Figure 3.18: Example of the effect of dynamic vs. static scheduling: (a) Two BW processes P1 and P2 with bandwidth requirement a little less than $L_{BW}/2$, except for the first half of process P1 that has a bandwidth requirement of L_{BW}; (b) Static scheduling; (c) Dynamic scheduling.

evaluated resource-aware static and dynamic schedulers when the intensity of resource contention among applications is aggravated by the presence of additional cores.

3.4.5 Performance of Dynamic vs. Static Scheduling Policy

The static scheduler outperforms in general the dynamic scheduler assuming (i) the category of the applications is known in advance *and* (ii) the category of the applications does not change during execution. For most of the benchmark mixes used in the results so far, (ii) was true and consequently the dynamic scheduler performance lagged behind that of the static one. In addition, in the cases where the dynamic scheduler performed better than the static (as for e.g., Mix1 and Mix2 in Fig. 3.7), the improvement was not significant. In order to illustrate more the benefit that the dynamic scheduler can offer, we investigated the following scenario.

We considered two example processes P1 and P2, both of them of type BW. The first half of process P1 saturates the system's available memory bandwidth L_{BW}, while its second half has a bandwidth requirement that is a little less than half of L_{BW}. Process P2 has a bandwidth requirement throughout its execution that is a little less than half of L_{BW} (see Fig. 3.18(a)). Suppose that the static scheduler, which assumes that the category of each benchmark is fixed and known beforehand, is fed with the information that process P1 is of type BW with requirement L_{BW} and process P2 is of type BW with requirement $L_{BW}/2$. Since the total bandwidth that P1 and P2 require exceeds L_{BW}, the static scheduler schedules the two processes serially (Fig. 3.18(b)). For the synthetic applications that we used for this experiment, the overall execution time under the static scheduling turned out to be $t_1 = 1001.66$ sec.

This is to be contrasted with the behavior of the proposed dynamic scheduler which periodically recharacterizes processes (ACHR phase). Each process is run in isolation for AC quanta. In this experiment the value of AC was 15 ticks, each tick being $200msec$, and 2 recharacterization intervals were used. After time $t_C = 2 \cdot AC \cdot 200\text{msec} = 6$ sec, the dynamic scheduler determines that the type of P1 and P2 is BW but their total requirement exceeds L_{BW} so it also schedules them serially (Fig. 3.18(c)). After this is done, the second recharacterization takes place which finds out that both processes are still of type BW but their combined bandwidth is below L_{BW} and so they are scheduled together. This results in an overall execution time of $t_2 = 762.5$ sec, which constitutes an improvement of 23.88% over the static. The above experiment presents a particularly favorable example of the benefit that the dynamic scheduler can offer. The actual improvement depends, among other things, on the overhead of recharacterization (its frequency in terms of RCI and duration in terms of AC), but its full exploration is a topic of future research.

Overhead

The average execution overhead for each evaluated scheduler was Linux 2.9ms, DI 0.8ms, LCA 0.8ms, BAOS & Perf&Fair 3.7ms, Proposed Dynamic 3.1ms, and Proposed

Static 0.8ms.

3.5 CONCLUSION

Shared-resource contention is one of the primary bottlenecks to application performance in modern CMPs. It arises from the sharing of certain computing resources among the many cores of a CMP - such as last-level caches, data buses, and main-memory. If processes are executed without careful consideration given to resource-pressure, they can end up interfering with each others' execution, resulting in not only performance drops, but also wasted power.

This chapter presents two versions (static and dynamic) of a fine-grained pressure-aware scheduling policy that co-schedules applications based on their shared-resource interference profiles. The static scheduler requires a separate offline profiling phase while the dynamic scheduler functions entirely according to run-time performance statistics. We develop and validate our methodology on two different Intel servers with Cache Monitoring Technology (CMT). Experimental results demonstrate an increase in throughput of 16% to 40%, and an increase in fairness of 65% to 130% compared to Linux's completely fair scheduler.

CHAPTER 4
PRIORITY-AWARE SCHEDULING

This chapter describes a priority-aware scheduling technique for CMPs by monitoring shared-resource pressure.

4.1 INTRODUCTION

For many embedded CMP applications , it is typical practice to divide computing tasks into smaller processes, each with its own level of execution urgency, minimum performance requirement, or other related metrics such as tail latency or minimum response time. These requirements are abstracted into various priority levels with the conceptual details varying only slightly between a vast variety of applications. For example, in mission-critical applications such as for manned spacecraft or submarines, life-support and propulsion-related processes are of primary importance, compared to other tasks such as collecting scientific or cartographic data. Since any failure in these subsystems may be catastrophic, the effective prioritization and subsequent scheduling of all computing tasks is of utmost importance.

On the other hand, cloud service providers also rely heavily on the prioritization of specific workloads in order to fulfill minimum Quality of Service (QoS) requirements agreed upon in Service Level Agreements (SLAs). Since cloud infrastructure can be shared among multiple customers, a significant challenge that arises in server-based environments, termed the noisy-neighbor problem, can degrade performance by up to 63% in extreme cases. This is caused due to resource-interference from both the unpredictable execution of end-user workloads as well as the inherent non-determinism present in CMP systems. While the effect of Quality of Service (QoS) violations is not catastrophic in these cases, it can nevertheless result in significant financial and business-related penalties for the service provider. In short, the development of priority-aware scheduling techniques for CMP architectures is a multi-faceted challenge that is of growing importance in a world that is rapidly automating its most critical functions.

This chapter proposes a methodology to implement performance prioritization in a CMP system where each application is classified as either High Priority (HP) or Low Priority (LP). This approach ensures that all HP applications meet their minimum performance requirements, while *simultaneously* ensuring that LP applications do not experience resource starvation or other scheduling deadlocks, eventually leading to degradation in overall throughput, performance, and fairness of the system. This is achieved by providing a fair execution environment for processes by scheduling the processes with the least accumulated progress first. This is combined with a dynamic shared-resource pressure-aware application-to-core assignment policy that yields overall throughput comparable to hardware-assisted cache/resource allocation technologies. Although some hardware-allocation features have previously been incorporated into modern CMPs to aid in the prioritization of specific processes, not all models of CMPs incorporate them because of the increased complexity, cost, and power requirements. Intel's Cache Allocation Technology (CAT) and HP's Non Uniform Memory Access (NUMA) are examples of such hardware.

Experimental validation of the proposed methodology on an Intel Xeon Gold 6130 CPU demonstrates an HP application performance improvement of 36.4% over Linux *while also maintaining* high throughput for LP applications. In addition, results show that our methodology achieves HP application performance comparable to a state-of-the-art hardware cache partitioning method (Intel's POCAT) [26]. Specifically, it achieves average performance within 14.5% to 0.2% of POCAT without the need for built-in hardware support.

4.2 PROPOSED METHODOLOGY

This section lists definitions and describes our proposed scheduling approach.

4.2.1 Application Characterization

We utilize a generalized version of the application categorization scheme presented in [84] and [85], where applications can belong to one of three types depending upon their shared-resource requirements:

1. **Streaming** This category comprises applications with working set sizes larger than the available LLC of the CMP system. When executed on a system with a Least Recently Used (LRU) cache replacement policy, they degrade the performance of other applications that rely on LLC availability for ideal execution.

2. **Cache Intensive** Consists of applications with working set sizes that fit entirely within the available LLC. Many *Cache Intensive* applications take up the bulk of the available space in the LLC. As a result, their performance can degrade significantly in the presence of other applications (excluding *Core Fitting*). These applications perform best when executed in isolation.

3. **Core Fitting** This category contains applications with working set sizes that fit completely within the lower-level L2 and L1 caches. As a result, they are agnostic to the presence of shared last-level caches, and do not impact the performance of any co-scheduled *Streaming* and *Cache Intensive* applications.

The authors experimentally demonstrate this to be an effective characterization technique for developing scheduling methods based on the minimization of shared-resource contention in CMPs.

4.2.2 Progress

The progress of each process is quantified using the metric *Progress Ratio* which can be readily extracted online without the need for prior information about the workload. In addition to quantifying execution progress, use of the *Progress Ratio* metric also allows for the dynamic compensation of performance losses by identifying the source of destructive resource interference. This contrasts with other progress-aware scheduling approaches that try to compensate for performance losses by combining processes in an alternate way.

The progress ratio of an application A when executing in parallel with co-runners is defined as the ratio of the instruction rate during all time periods of execution to the estimated ideal instruction rate acquired at the beginning of execution. Equation 4.1 represents $Progress$ in mathematical form:

$$Progress^A = \frac{IPC^A{}_{co-running}}{IPC^A{}_{ideal}} \tag{4.1}$$

We calculate the initial instruction rate by using the light-weight resource-contention scenario presented in [9]. This involves scheduling each application on one core of the system, while also executing any available *Core-Fitting* applications on the remaining cores of the CMP. This is done repeatedly for a fixed amount of time until all applications have been characterized. We empirically determined this to be 30 time quanta of 200ms each to effectively profile incoming applications.

4.2.3 Priority

Priority is enforced by utilizing a user-selectable minimum progress threshold P_{thr} for HP applications. At runtime, the accumulated progress of HP applications is checked at the end of each time quantum. If any HP application achieves progress lower than the required threshold, it is assigned to a core and scheduled to be executed in the next iteration. The scheduler initially attempts to enforce P_{thr} for each of the prioritized applications in this manner. To fill the rest of the available cores, LP applications are selected according to a fitness function that determines the best-fitting application given the system's available resources. This ensures adequate progress, and consequently performance, for the HP applications while also avoiding resource starvation for the LP applications. This approach works well when *Core Fitting* applications are present in the workload, which help to offset performance degradation for the more resource-intensive HP and LP applications.

However, in practice a workload could contain a combination of only *Streaming* and/or *Cache Intensive* applications. In such cases, the performance of HP applications cannot be met by simple repeated scheduling of applications. For these workloads, the resource requirements for any possible combination of N (N=#CPUs) applications will exceed the total amount of resources available to the system. When such a case is encountered, the proposed scheduler utilizes a 'core-dropping' methodology to enforce the progress of HP applications. When the scheduler detects a performance violation ($Progress^A < P_{thr}$) for one or more HP applications despite being re-scheduled for several consecutive time quanta (T_{thr}), it can disable the assignment of applications to one core of the system. This allows for reduction of shared-resource contention, which is the primary bottleneck to application performance for resource-intensive workloads on CMPs. By 'dropping' a single core along with the use of a progress-aware scheduling methodology, we are able to successfully enforce a minimum progress threshold for HP applications while also achieving greater throughput for the LP applications (compared to Linux). Section 4.4 demonstrates that the 'dropping' of a single core does not hurt overall performance of workloads with resource requirements far exceeding the total capacity of the system.

4.2.4 Application-to-Core Assignment

Alg.3 presents the runtime behavior of our proposed scheduler. It functions by dividing application execution into discrete time quanta of predefined duration. At the end of each quantum, it decides a priority and progress aware app-to-core mapping for the next

Algorithm 3 Proposed scheduling policy

```
    INPUTS
 1: $(BW_{MM}^{peak}, SIZE_{LLC}, \#CPUs) \leftarrow$ system-specific limits
 2: $runqueue \leftarrow$ list of p applications with corresponding priorities [H/L]
 3: $P_{thr} \leftarrow$ min progress ratio for HP applications
 4: $T_{thr} \leftarrow$ # of successive quanta after which core is disabled

    APPLICATION-TO-CORE ASSIGNMENT
 5: Update $Progress^A$, $\forall p$ run on CPU
 6: Sort $runqueue$ in ascending order based on $Progress^A$
 7: $Cache_{on} = Stream_{on} = False$
 8: $BW_{Remain}$=$BW_{MM}^{peak}$, $LLC_{Remain}$=$SIZE_{LLC}$
 9: $CPU_{Remain} = \#CPUs$
10: $Drop_{Core} = False$
11: while $CPU_{Remain} > 0$ do
12:     for each $p \in runqueue$ do                                ▷ First check HP apps
13:         if $priority^A = High$ then
14:             if $(Progress^A < P_{thr})$ then
15:                 Select $p$ for execution and remove from $runqueue$
16:                 $t\_fail^A + +$, $CPU_{Remain} - -$
17:                 if $type^A = Streaming$ then
18:                     $BW_{Remain} - = BW_{MM}^A$
19:                 if $type^A = Cache$ then
20:                     $LLC_{Remain} - = USAGE_{LLC}^A$
21:         else $t\_fail^A = 0$
22:     if $(t\_fail^A > T_{thr})$ and $(Drop_{Core} = False)$ then
23:         $Drop_{Core} = True$, $CPU_{Remain} - -$
24:     for each $p \in runqueue$ do                                ▷ Schedule LP apps
25:         if $type^A = Streaming$ and not $Cache_{on}$ then
26:             $Stream_{on} = True$
27:             if $BW_{MM}^A < BW_{Remain}$ then
28:                 $BW_{Remain} - = BW_{MM}^A$, $CPU_{Remain} - -$
29:         if $type^A = Cache$ and not $Stream_{on}$ then
30:             $Cache_{on} = True$
31:             if $USAGE_{LLC}^A < LLC_{Remain}$ then
32:                 $LLC_{Remain} - = USAGE_{LLC}^A$, $CPU_{Remain} - -$
33:         if $type^A = Core$ then $CPU_{Remain} - -$
34:         if $(Stream_{on})$ then
35:             $(Limit, Usage^A) = (BW_{Remain}, BW_{MM}^A)$
36:         else
37:             $(Limit, Usage^A) = (LLC_{Remain}, USAGE_{LLC}^A)$
38:     Select p from $runqueue$ that maximizes FITNESS$(A)$
39:     $Limit - = Usage^A$, $CPU_{Remain} - -$
```

iteration.

System	Intel Xeon Gold 6130
Clock Frequency	2.10 GHz
#Cores/socket	16
L2	Unified 1024 KB
LLC size	22,528 KB
Memory	128 GB DDR4
Cache Monitoring Technology	Available
Cache Allocation Technology	Available
Hyperthreading	Disabled

Table 4.1: System specifications for evaluated server platform.

The algorithm requires the following inputs (lines ① to ④) : (1) the system's maximum main-memory bandwidth, LLC size, and number of available CPUs, (2) a queue of applications to be scheduled (*runqueue*), (3) the minimum required progress ratio P_{thr} for HP applications, and (4) T_{thr} which represents the maximum number of successive time quanta an HP application can fail to meet P_{thr}, before the scheduler disables a core. The parameter T_{thr} helps to mitigate wasteful core-dropping in cases when an HP application temporarily fails to meet P_{thr}. For example, if an HP application does not meet its P_{thr} for only one time quantum, disabling a core may result in CPU under-utilization. This is because minor fluctuations in performance can occur due to the inherent non-determinism present in the system. However, if an HP application fails to meet P_{thr} for several consecutive time quanta, core-dropping can be utilized to ease resource interference and improve the application's performance.

The algorithm begins by updating the progress ratio $Progress^{A}$ (using Equation 4.1) for each application A in the *runqueue* (line ⑥). The queue is then sorted in ascending order of accumulated progress for each application to ensure that applications with the least progress will be considered for scheduling first (line ⑥). This is done to ensure fair execution time across all applications, thereby preventing starvation. The two flags, $Cache_{on}$ and $Stream_{on}$ are initialized to $False$ (line ⑦). These flags will be used later to avoid co-execution of *Streaming* and *Cache* type applications, which would otherwise

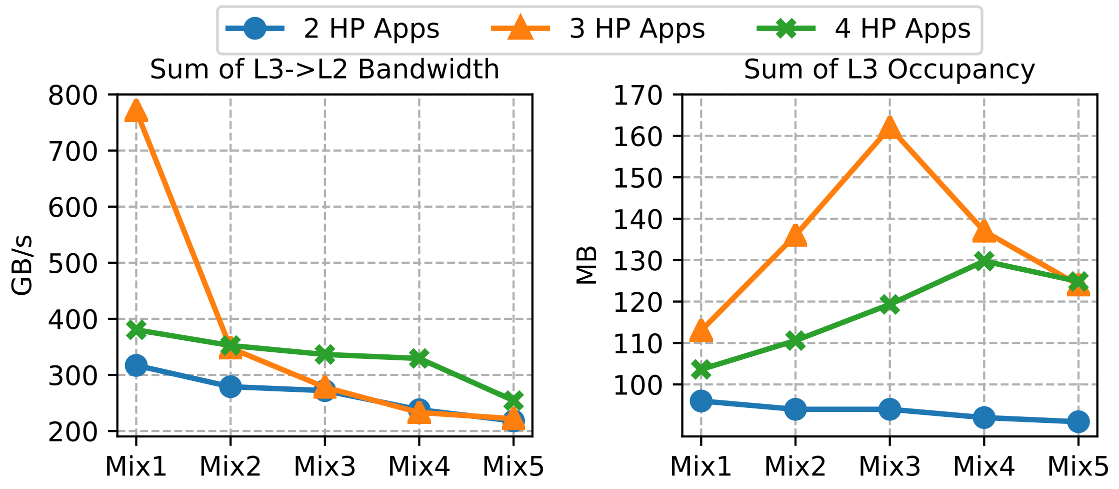


Figure 4.1: Sum of individual shared resource requirements for each evaluated application mix

result in high resource contention and low performance. The parameters BW_{Remain}, LLC_{Remain}, and CPU_{Remain} keep track of the system's available resources and are hence initialized to their maximum available values (lines 8 to 9). As each application is scheduled, its resource requirements are subtracted from the aforementioned parameters. Additionally, the flag $Drop_{core}$ is initialized to $False$ (line 10), signifying that all cores are available for scheduling.

The scheduling process begins by traversing *runqueue* to first check the progress ratio of each HP application (lines 12 and 13). If an HP application's accumulated progress is found to be lower than P_{thr}, it is scheduled to be executed in the next iteration and t_fail^A is incremented to reflect the drop in performance (lines 14 to 16). The parameter t_fail^A keeps track of how many successive time quanta an HP application failed to meet P_{thr}. In lines 17 to 20, the system's remaining resources are updated in order to effectively schedule remaining LP applications in the subsequent stage. If no violation of P_{thr} is detected, t_fail^A is reset to 0 since we are only interested in recording consecutive violations of the progress threshold (line 21). Finally, the scheduler checks if the current HP application's t_fail^A exceeded T_{thr} and sets the flag $Drop_{core}$ to $True$ if the condition is met (lines 22 and 23). The $Drop_{core}$ flag ensures that only one core is dropped (at most) for each time quantum. The process repeats until all HP applications with $Progress^A < P_{thr}$ have been scheduled.

Next, remaining LP applications are selected repeatedly from *runqueue* until all available cores have been filled (line 24). In each iteration, only one of the two flags $Cache_{on}$ or $Stream_{on}$ can be enabled, indicating the type of applications that will be scheduled in the following quantum. If $Cache_{on}$ is $True$, *Streaming* applications will not be scheduled, while if $Stream_{on}$ is $True$ then *Cache Intensive* applications will not be scheduled (lines 25 and 29). Depending upon the application type, the system's resource utilization will be updated by subtracting either BW^A_{MM} or $USAGE^A_{LLC}$ from BW_{Remain} or LLC_{Remain} respectively (lines 28 and 32). *Core Fitting* applications can be scheduled with with either group as they do not contribute to resource contention (line 33). In lines lines 34 to 37, the appropriate resource utilization for the application A is updated. The tuple $(Limit, Usage^A)$ is then used as input to a fitness function (Equation 4.2) to fill the unallocated cores with applications that best fit within the system's remaining resources (line 38).

$$\textsc{Fitness}(A) = 1/\left|\frac{Limit}{CPU_{Remain}} - Usage^A\right| \tag{4.2}$$

4.3 EVALUATION SETUP

We evaluate our proposed approach on an Intel Xeon Gold 6130 CPU @ 2.10GHz using all 16 cores of a single NUMA node consisting of a shared L3 cache (22MB) and main-memory controller. We consider the progress threshold P_{thr} of 0.7, which is close to the upper bound for application performance in a noisy neighbor environment [86]. We test 5 application mixes (each) for the cases of 2, 3, and 4 HP applications. Each application mix consists of 32 single-threaded applications taken from the Polybench [76] and Stream [74] benchmark suites. These mixes do not contain any *Core Fitting* applications, thus resulting in extremely high pressure on the system's shared resources. The mixes are

Table 4.2: High Priority Applications and Datasets

#HP	App1	App2	App3	App4
2	2mm (14M)	syrk (14M)	-	-
3	symm (12M)	correlation (12M)	gramschmidt (14M)	-
4	2mm (12M)	symm (12M)	stream (29M)	correlation (12M)

arranged in decreasing order of total L3→L2 bandwidth requirement, as shown in Figure 4.1. The scheduler is implemented in user-space as an extension of the work presented in [87]. Table 4.1 summarizes the system specifications.

4.4 EXPERIMENTAL RESULTS

We compare the performance of our proposed approach against: (1) Linux's Completely Fair Scheduler (CFS), (2) Perf [22], (3) Perf&Fair [88], and (4) POCAT [26]. Perf calculates the overall main memory requests and the average main memory bandwidth utilization of each application and then schedules the application with the best fit. Perf&Fair simultaneously addresses performance and fairness of applications by monitoring their main-memory bandwidth, L1 bandwidth, and progress ratio. It then uses the online average values in a fitness function to find applications for the next time quantum. POCAT uses Intel's TMAM metrics [89] to predict an optimal cache-ways allocation for HP applications (single partition for all HP applications). This method requires specialized hardware support (i.e. Intel's Cache Allocation Technology). We implement POCAT's predictive algorithm (based on AdaBoost) and compare the results to demonstrate how well our approach performs when measured against a hardware-assisted prioritization methodology.

4.4.1 High Priority Application Performance

Table 4.2 lists the HP applications in each evaluated application mix. Figures 4.2, 4.3, and 4.4 depict the HP performance improvement over Linux for the cases of 2, 3, and 4 HP

#	**Mix 1**	**Mix 2**	**Mix 3**	**Mix 4**	**Mix 5**
1	gemm_10M	syrk_14M	2mm_14M	nussinov_6M	syrk_14M
2	correlation_14M	symm_12M	syrk_14M	nussinov_14M	symm_12M
3	symm_14M	nussinov_6M	symm_12M	2mm_14M	nussinov_14M
4	symm_12M	2mm_14M	nussinov_6M	syrk_14M	nussinov_6M
5	nussinov_12M	nussinov_14M	trmm_14M	symm_12M	trmm_14M
6	syrk_2M	trmm_14M	nussinov_14M	nussinov_12M	nussinov_12M
7	gemm_4M	ludcmp_6M	nussinov_12M	deriche_14M	deriche_14M
8	gramschmidt_2M	lu_14M	deriche_14M	deriche_12M	deriche_12M
9	3mm_8M	nussinov_12M	ludcmp_6M	ludcmp_6M	adi_6M
10	trisolv_6M	gramschmidt_8M	adi_6M	adi_6M	ludcmp_6M
11	cholesky_14M	deriche_14M	deriche_12M	gramschmidt_8M	3mm_14M
12	lu_4M	deriche_12M	3mm_14M	3mm_14M	gramschmidt_8M
13	atax_12M	adi_6M	gramschmidt_8M	trisolv_10M	floyd_6M
14	gesummv_14M	3mm_14M	fdtd-2d_12M	floyd_6M	trisolv_10M
15	fdtd-2d_12M	floyd_6M	floyd_6M	bicg_2M	syrk_10M
16	correlation_2M	syrk_10M	trisolv_10M	cholesky_8M	cholesky_8M
17	doitgen_10M	syrk_6M	bicg_2M	syrk_6M	2mm_4M
18	adi_6M	floyd_12M	cholesky_8M	doitgen_8M	floyd_12M
19	lu_12M	cholesky_8M	lu_2M	lu_2M	syrk_6M
20	gesummv_4M	lu_2M	syrk_6M	syrk_10M	lu_2M
21	lu_10M	3mm_2M	doitgen_8M	fdtd-2d_12M	doitgen_8M
22	syrk_8M	syrk_8M	syrk_10M	syrk_8M	3mm_2M
23	syr2k_12M	fdtd-2d_12M	syrk_8M	3mm_2M	fdtd-2d_12M
24	floyd_2M	2mm_4M	3mm_2M	lu_14M	syrk_8M
25	2mm_4M	heat-3d_10M	heat-3d_10M	2mm_4M	seidel-2d_10M
26	adi_10M	seidel-2d_2M	doitgen_14M	heat-3d_10M	seidel-2d_2M
27	gemver_12M	trisolv_10M	seidel-2d_10M	seidel-2d_10M	heat-3d_10M
28	syr2k_8M	seidel-2d_6M	seidel-2d_2M	seidel-2d_6M	seidel-2d_6M
29	floyd_6M	jacobi-2d_6M	lu_14M	seidel-2d_2M	doitgen_14M
30	doitgen_4M	doitgen_14M	jacobi-2d_6M	jacobi-2d_6M	jacobi-2d_4M
31	syr2k_6M	jacobi-1d_12M	jacobi-1d_12M	jacobi-1d_12M	jacobi-2d_6M
32	syr2k_2M	jacobi-2d_4M	jacobi-2d_4M	jacobi-2d_4M	jacobi-1d_12M

Table 4.3: List of benchmarks in each evaluated mix

applications respectively. We quantify the performance of each mix by its *throughput*, which is the ratio of the total IPC achieved by our policy to the total IPC achieved by using Linux. For 2 HP applications (Fig. 4.2), our policy achieves the highest throughput gain among non hardware-assisted methodologies, yielding an average 42.9% increase in throughput over Linux. For the group with the highest L3→L2 bandwidth requirement (Mix #1), POCAT was able to achieve 20% greater throughput than the proposed approach. However, it requires the use of Intel's Cache Allocation Technology which is currently limited to the Xeon W family of processors [90]. Moreover, as the shared-resource

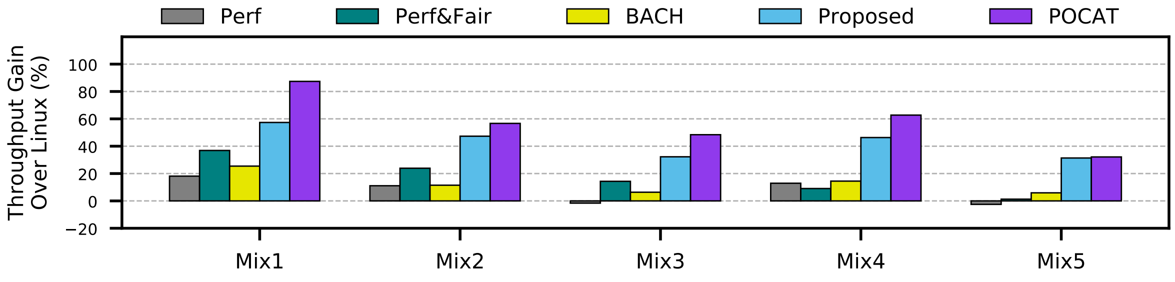


Figure 4.2: High Priority Throughput gain % (2 HP Apps)

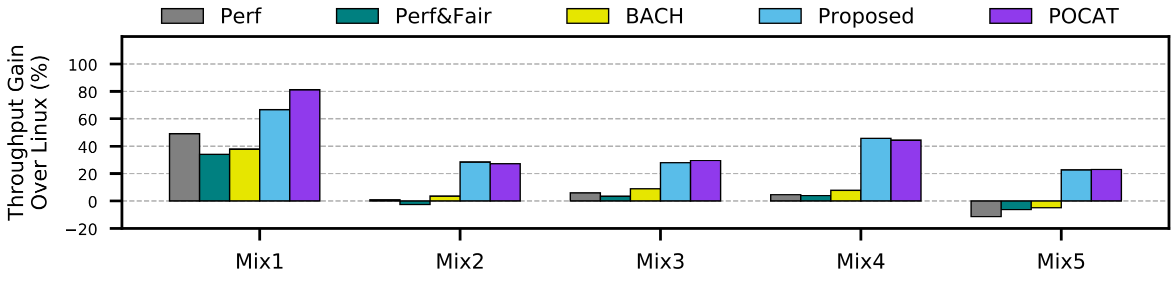


Figure 4.3: High Priority Throughput gain % (3 HP Apps)

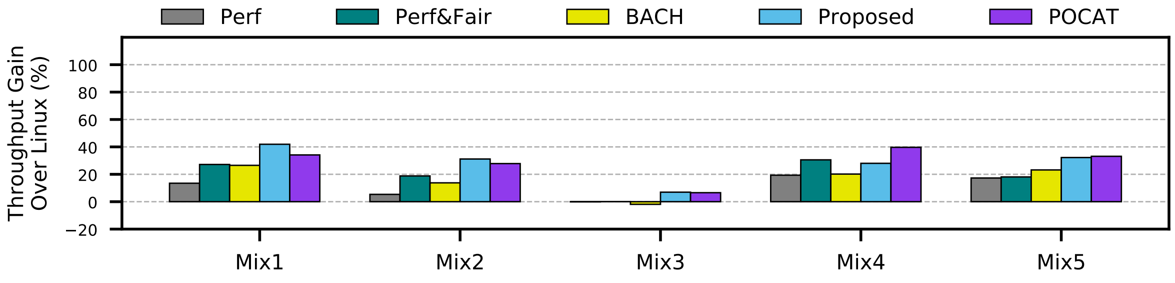


Figure 4.4: High Priority Throughput gain % (4 HP Apps)

pressure decreases (while still being enough to cause performance loss due to contention), our proposed approach is able to achieve performance comparable to the hardware-assisted POCAT, with the difference in throughput being only 0.76% for Mix #5. The average difference in performance over all mixes for 2 HP applications was 14.54%

Similarly, this trend also holds for 3 and 4 HP applications (Fig. 4.3 & 4.4) but with diminishing difference in performance between our proposed approach and the hardware-assisted POCAT. In addition, the average gain in throughput lowers with an increase in the number of HP applications. Table 4.4 summarizes the gain in performance

Table 4.4: Avg High Priority Throughput Gain %

#HP Apps	Over Linux	Over POCAT
2	42.9%	-14.5%
3	38.3%	-2.8%
4	28.1%	-0.2%

for all three cases.

4.4.2 Overall Workload Performance

Regarding the overall performance of each of the fifteen evaluated mixes, Figures 4.5, 4.6, and 4.7 show the IPC distribution of all 32 applications in the mix. Because of its progress-aware nature, our proposed approach successfully avoids the low IPCs incurred by Linux. In the case of 2 HP applications (Fig. 4.5), our approach achieved the highest minimum IPC for all 5 mixes. The average min IPC achieved over all 5 mixes was 0.45 for Linux, 0.47 for Perf, 0.53 for Perf&Fair, and 0.65 for the proposed approach. The hardware-assisted POCAT method achieved marginally higher min IPC of 0.7. For 3 HP applications (Fig. 4.6), our approach achieved an average min IPC of 0.45 compared to Linux's 0.36. The Perf, Perf&Fair, and POCAT approaches also yielded average min IPC of 0.45. Finally, in the case of 4 HP applications (Fig. 4.7), the average of the min IPC achieved over all 5 mixes by our approach was 0.45, compared to Linux's 0.35 and POCAT's 0.46. The Perf and Perf&Fair schedulers achieved an average min IPC of 0.42 and 0.49 respectively.

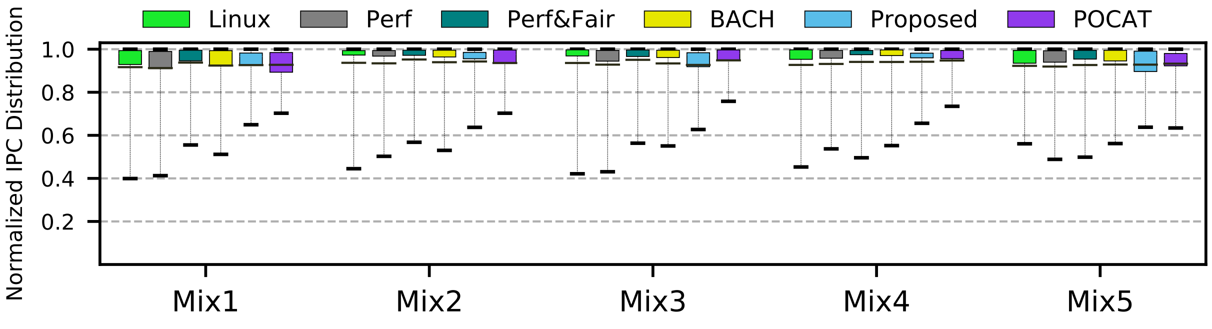


Figure 4.5: IPC distribution for all applications in mix (2HP Apps)

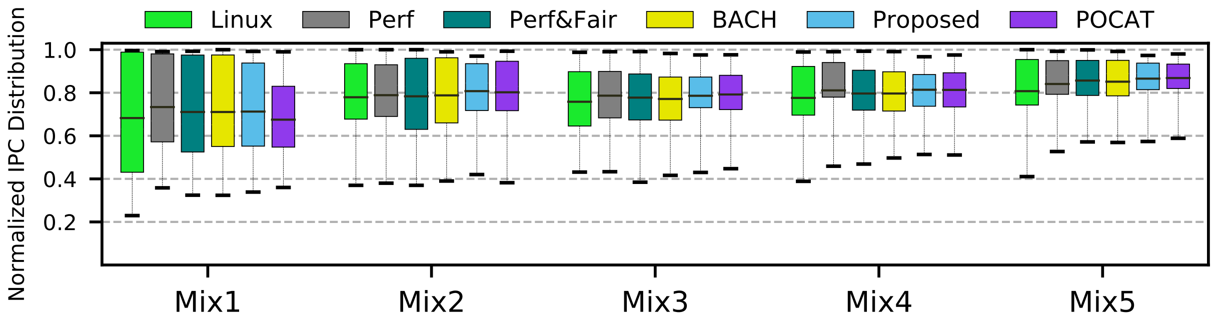


Figure 4.6: IPC distribution for all applications in mix (3HP Apps)

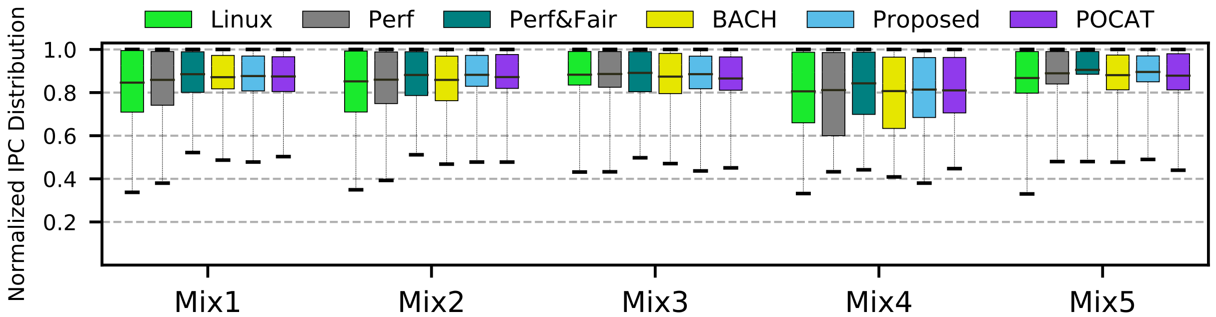


Figure 4.7: IPC distribution for all applications in mix (4HP Apps)

4.4.3 Overhead

The average overhead of our proposed policy is 3.5ms for each time quanta.

4.5 CONCLUSION

The prioritization of certain computing workloads over others is an inherent challenge in the field of embedded systems and computing as a whole. Distinct levels of priority are used as an abstraction for signifying the importance of different computing tasks. This model is useful for a large variety of CMP applications, such as for the safe functioning of mission-critical real-time systems, or for ensuring the promised level of application performance to customers in cloud-based server environments.

This chapter presents a methodology for improving the performance of high-priority applications under concurrent application execution, *while also maintaining* high throughput for low-priority applications. Experimental results on an Intel Xeon Gold 6130

CPU demonstrate an average 36.4% improvement in high-priority application performance over Linux. Furthermore, the proposed approach achieves high-priority performance comparable to a hardware-assisted prioritization methodology (between 14.2% to 0.2% difference in performance).

CHAPTER 5
POWER-AWARE SCHEDULING

This chapter describes a power-aware scheduling technique for Chip Multi Processors which reduces power consumption without affecting application performance.

5.1 INTRODUCTION

While CMPs were able to largely address the problem of the rapidly approaching *Power Wall* that plagued development of single-core architectures in the late 90's, advances in CMP design in the 21^{st} century have gradually yielded entirely new power-related challenges. In addition to the unprecedented proliferation of battery-powered embedded systems and increasing concern for the environmental impacts of computing, several manufacturing and design concerns have also exacerbated the need for aggressive reduction of CPU power consumption. One of the most complicated challenges arises from the high density of devices (transistors) on each die, combined with the continually shrinking physical footprints of chips [2]. This results in hot-spots which exacerbate the already existing problem of efficient heat dissipation in processors ($P_{static} + P_{dynamic} \propto heat$). As a result, the whole processor cannot be powered on at the same time, leading to a phenomenon termed Dark Silicon. Finding solutions to the problem of dark silicon is further complicated by the already existing problems of premature aging or component failure due to extreme heat. Advancements in technology such as the introduction of Dynamic Voltage And Frequency Scaling (DVFS) have helped increase power efficiency of CMPs by allowing frequencies to be adjusted in response to dynamically changing performance and power requirements. DVFS implementations control the clock frequency by varying the rate of the charge-discharge cycle of capacitors, which is directly proportional to the magnitude of voltage applied across their terminals. Higher voltages speed up the charge-discharge rate of capacitors, thereby increasing clock frequency, while lower voltages slow down the charge-discharge rate to decrease the clock frequency. Due to the exponential relation of electrical power and voltage ($P_{dynamic} \propto V^2 \cdot f$), a small drop in voltage leads to a large reduction in power consumption. In single-core processors, DVFS implementations adjust the frequency in response to the performance of the currently running workload. A popular approach is to determine if the workload can run without noticeable performance drop on a lower frequency, and then adjust the voltage accordingly. The same approach cannot be used for multi-core processors because the area/power overhead of having multiple voltage regulators for each core outweighs the potential benefits of having per-core voltage control [16]. Consequently, a major constraint of cluster-wide DVFS implementations is that all cores belonging to the same die must operate at the same frequency. This is not ideal for power consumption as different types of applications reach peak performance at different frequencies. For many types of applications, increasing the frequency beyond this point does not yield any performance benefits.

This chapter presents a run-time manager that focuses on improving power efficiency for clustered CMPs. Specifically, it monitors the activity of concurrently executing applications and utilizes neural networks to determine the minimum possible clock frequency beyond which no performance gains will be obtained. Experimental results on the Odroid-XU3 board (Samsung Exynos 5422 SoC) show that the proposed methodology improves power efficiency ($MIPS/Watt$) by 23% compared to Linux's performance governor in exchange for a negligible performance drop of only 3%.

5.2 MOTIVATION

As a motivation, Figure 5.1 illustrates the varying rates of change in power consumption and normalized Instructions Per Second (IPS), in response to changes in the processor's clock frequency, for three different application groups on the Odroid-XU3 platform [91] (quad-core ARM Cortex-A15 cluster on Samsung Exynos 5422 SoC).

Each group consists of four concurrently executing applications. Moreover, the groups

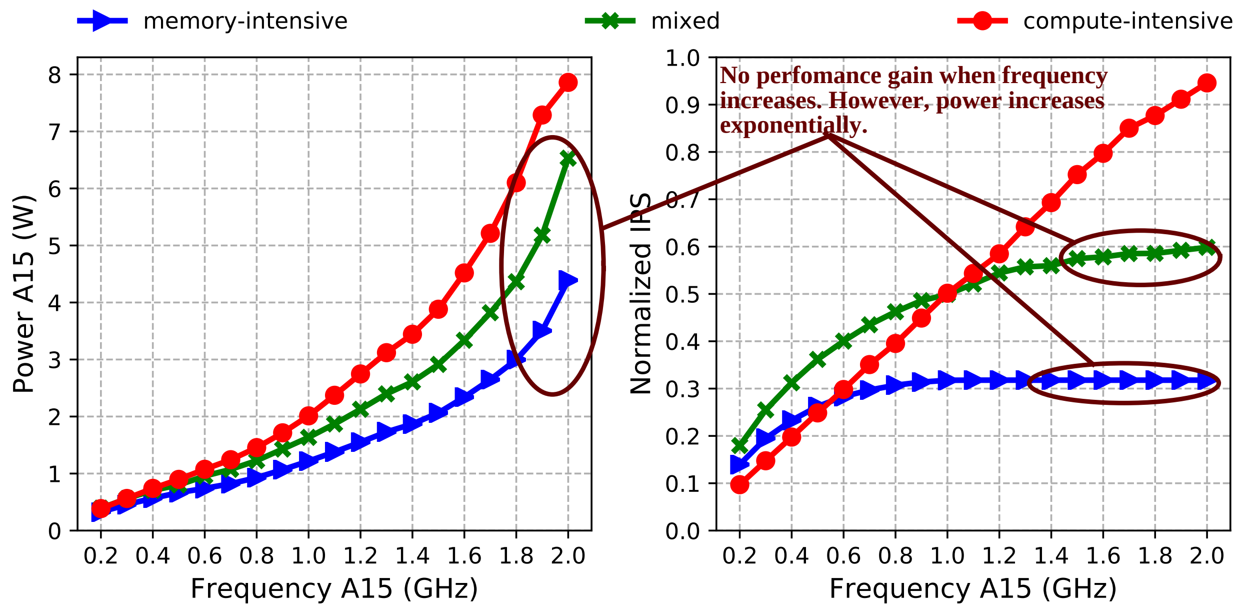


Figure 5.1: Power consumption and normalized IPS for different types of application groups.

are categorized into three different classes based on the type of the applications: (1) a *memory-intensive* group, which consists of applications with average shared-bus bandwidth greater than 1GB/s; (2) a *compute-intensive* group, which consists of applications with average shared-bus bandwidth less than 10MB/s; and (3) a *mixed* group, which consists of both memory- and compute-intensive applications. The rate of change in performance is governed by the composition of each application group. Particularly, the memory-intensive group achieves maximum performance at lower frequencies than the compute-intensive one. Additionally, the mixed workload experiences performance gains up till the maximum frequency but with diminishing returns at each increment. Despite the varying rates of change in IPS, which saturates after a certain point, the change in power consumption is quadratic due to the non-linear connection between power and voltage [35].

5.3 PROPOSED METHODOLOGY

The proposed methodology uses multiple MLP neural networks as classifiers to pro-actively set the operating frequency in order to increase power-efficiency while satisfying specific performance thresholds. Particularly, our method consists of three steps: (1) selection of the appropriate PMC events; (2) training of the MLPs; (3) run-time frequency selection. We utilized the Odroid-XU3 board which consists of a big and a LITTLE cluster with four Cortex-A15 and four Cortex-A7 cores respectively. Although we use the A15 cluster for developing and validating our methodology, the approach can also be used for the A7 cluster which accesses PMC counts in a similar way. Finally, Odroid-XU3 supports a built-in Power Monitoring Unit (PMU) to accurately monitor power consumption per cluster at run-time.

5.3.1 Selection of PMC events

The Cortex-A15 cluster includes six PMCs with a total of 66 available events to gather statistics regarding the operation of the processor and the memory system. Figure 5.2 shows a block diagram of the A15's Performance Monitoring Unit (PMU). As only six events can be monitored simultaneously, deciding which set of events to count determines the accuracy of our frequency prediction (Section 5.3.2). Due to this constraint, one of the most important factors when choosing a set of PMC events is the degree of multicollinearity between them. Multicollinearity occurs when two or more inputs record 'overlapping' information. Identifying events with minimal multicollinearity helps to improve the accuracy of our frequency estimation with neural networks.

Some events such as `0x6D`[1] (`STREX_PASS_SPEC`) show minimal variation in their counts for different types of application mixes, while others events such as `0x62` (`BUS_ACCESS_SHARED`) show large variations. We shortlist the PMC events that have significant variation between the different application mixes: 16 PMC events related to the memory-hierarchy of A15 (e.g., shared memory resources and private L1 cache) and 2 PMC events related to computing performance (instructions retired and cycle count). Figure 5.3 shows an overview of the top 18 shortlisted events. This will be reduced to a maximum of 6

[1]Hexadecimal code for the PMC event

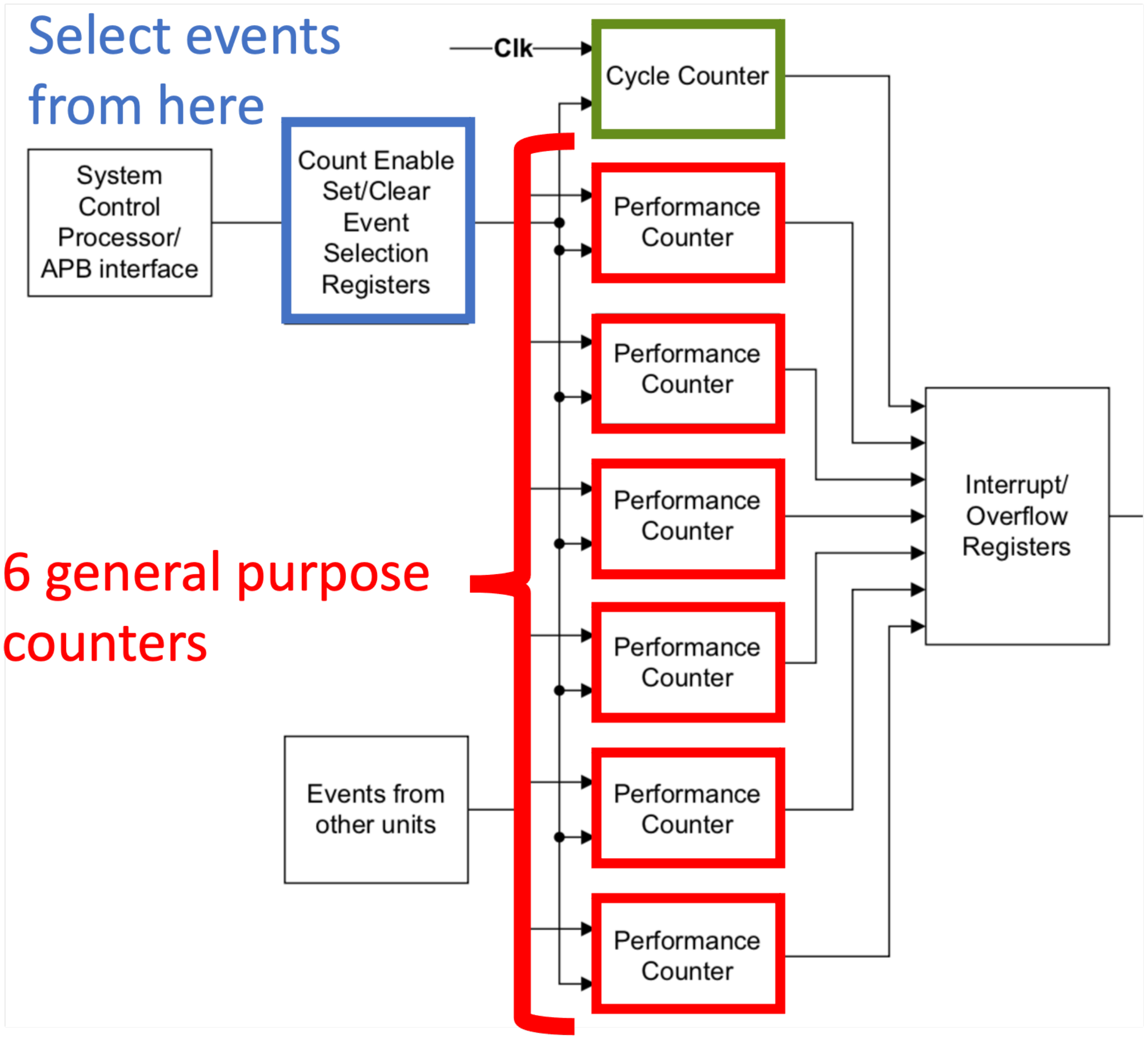
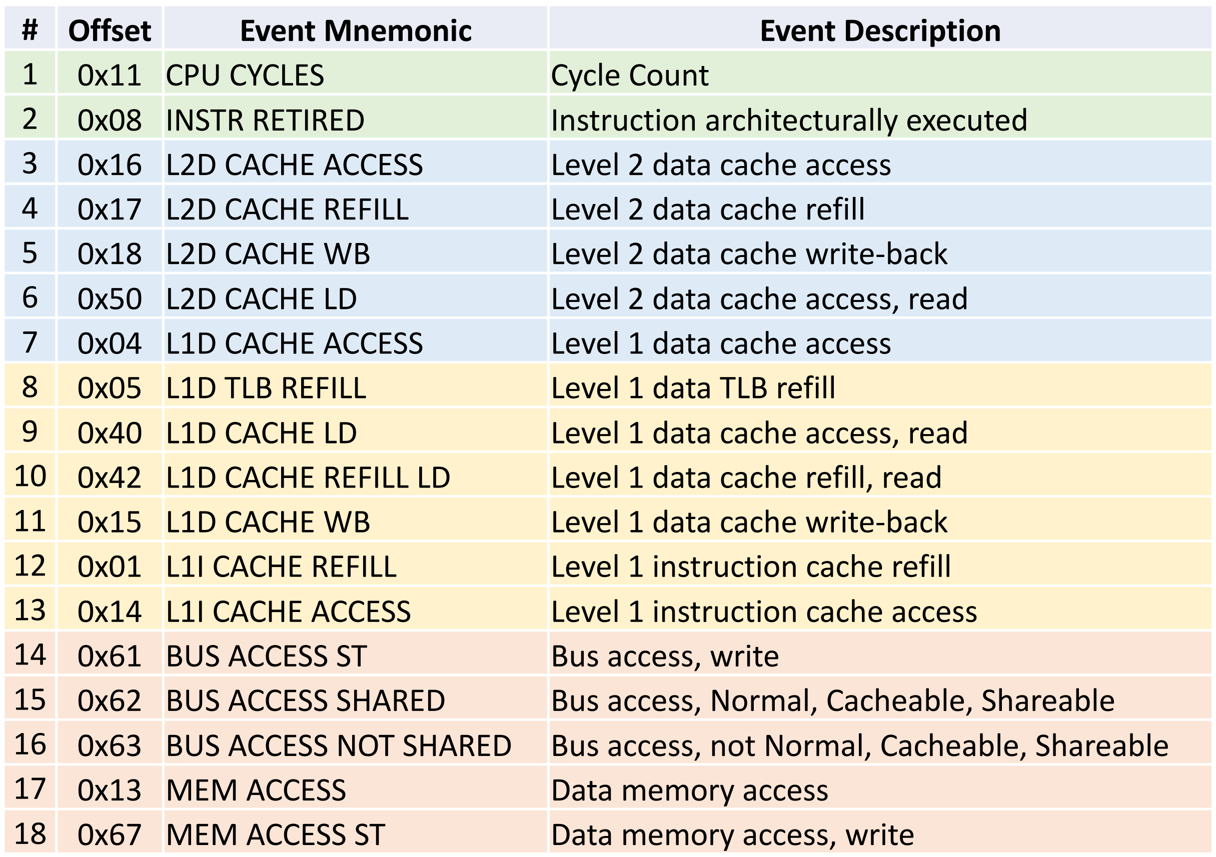

Select events from here
Clk
Cycle Counter
System Control Processor/ APB interface
Count Enable Set/Clear Event Selection Registers
Performance Counter
6 general purpose counters
Interrupt/ Overflow Registers
Events from other units

Figure 5.2: Block Diagram for Cortex-A15 PMU (ARM PMUv2 architecture)

events through an analysis of multi-collinearity and hierarchical clustering.

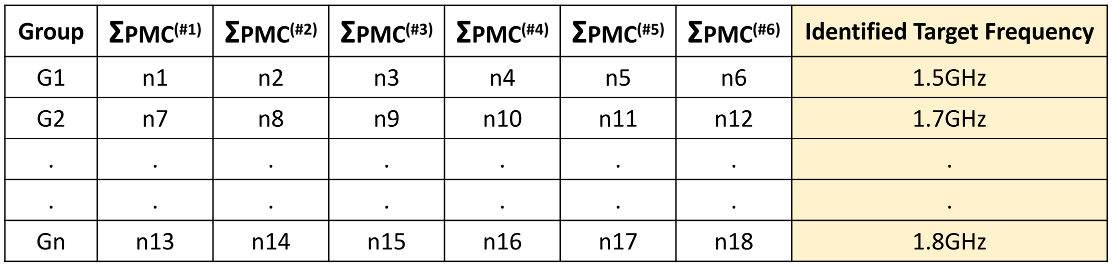
5.3.2 Training of MLPs

After identifying the most appropriate set of PMC events, the next step is to create a training set, based on the values of these PMCs. The goal is to estimate the *minimum possible frequency* on which an application group (four concurrent applications) should run in order to satisfy a performance threshold IPS_{thr}. For each required threshold, a neural



#	Offset	Event Mnemonic	Event Description
1	0x11	CPU CYCLES	Cycle Count
2	0x08	INSTR RETIRED	Instruction architecturally executed
3	0x16	L2D CACHE ACCESS	Level 2 data cache access
4	0x17	L2D CACHE REFILL	Level 2 data cache refill
5	0x18	L2D CACHE WB	Level 2 data cache write-back
6	0x50	L2D CACHE LD	Level 2 data cache access, read
7	0x04	L1D CACHE ACCESS	Level 1 data cache access
8	0x05	L1D TLB REFILL	Level 1 data TLB refill
9	0x40	L1D CACHE LD	Level 1 data cache access, read
10	0x42	L1D CACHE REFILL LD	Level 1 data cache refill, read
11	0x15	L1D CACHE WB	Level 1 data cache write-back
12	0x01	L1I CACHE REFILL	Level 1 instruction cache refill
13	0x14	L1I CACHE ACCESS	Level 1 instruction cache access
14	0x61	BUS ACCESS ST	Bus access, write
15	0x62	BUS ACCESS SHARED	Bus access, Normal, Cacheable, Shareable
16	0x63	BUS ACCESS NOT SHARED	Bus access, not Normal, Cacheable, Shareable
17	0x13	MEM ACCESS	Data memory access
18	0x67	MEM ACCESS ST	Data memory access, write

Figure 5.3: Top 18 events with minimal variation in recorded counts



Group	$\Sigma PMC^{(\#1)}$	$\Sigma PMC^{(\#2)}$	$\Sigma PMC^{(\#3)}$	$\Sigma PMC^{(\#4)}$	$\Sigma PMC^{(\#5)}$	$\Sigma PMC^{(\#6)}$	Identified Target Frequency
G1	n1	n2	n3	n4	n5	n6	1.5GHz
G2	n7	n8	n9	n10	n11	n12	1.7GHz
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
Gn	n13	n14	n15	n16	n17	n18	1.8GHz

Figure 5.4: MLP classifier training process

network takes the summed values of the selected events as input. Equation 5.1 shows the input tuple for our neural networks. The trained neural networks can then select the target frequency that satisfies IPS_{thr} from one of 10 available output classes, each corresponding to an available frequency of the processor ($1.1GHz$ up to $2.0GHz$).

$$\mathbf{X_{f_i}} = \Big\{ \sum_{i=1}^{4} PMC_i^{(\#1)}, \sum_{i=1}^{4} PMC_i^{(\#2)}, \sum_{i=1}^{4} PMC_i^{(\#3)}, \sum_{i=1}^{4} PMC_i^{(\#4)}, \sum_{i=1}^{4} PMC_i^{(\#5)}, \sum_{i=1}^{4} PMC_i^{(\#6)} \Big\} \tag{5.1}$$

Each performance threshold requires an input training set for each of the available frequency levels. Figure 5.4 shows a visual representation of the training methodology.

5.3.3 Run-Time Frequency Selection

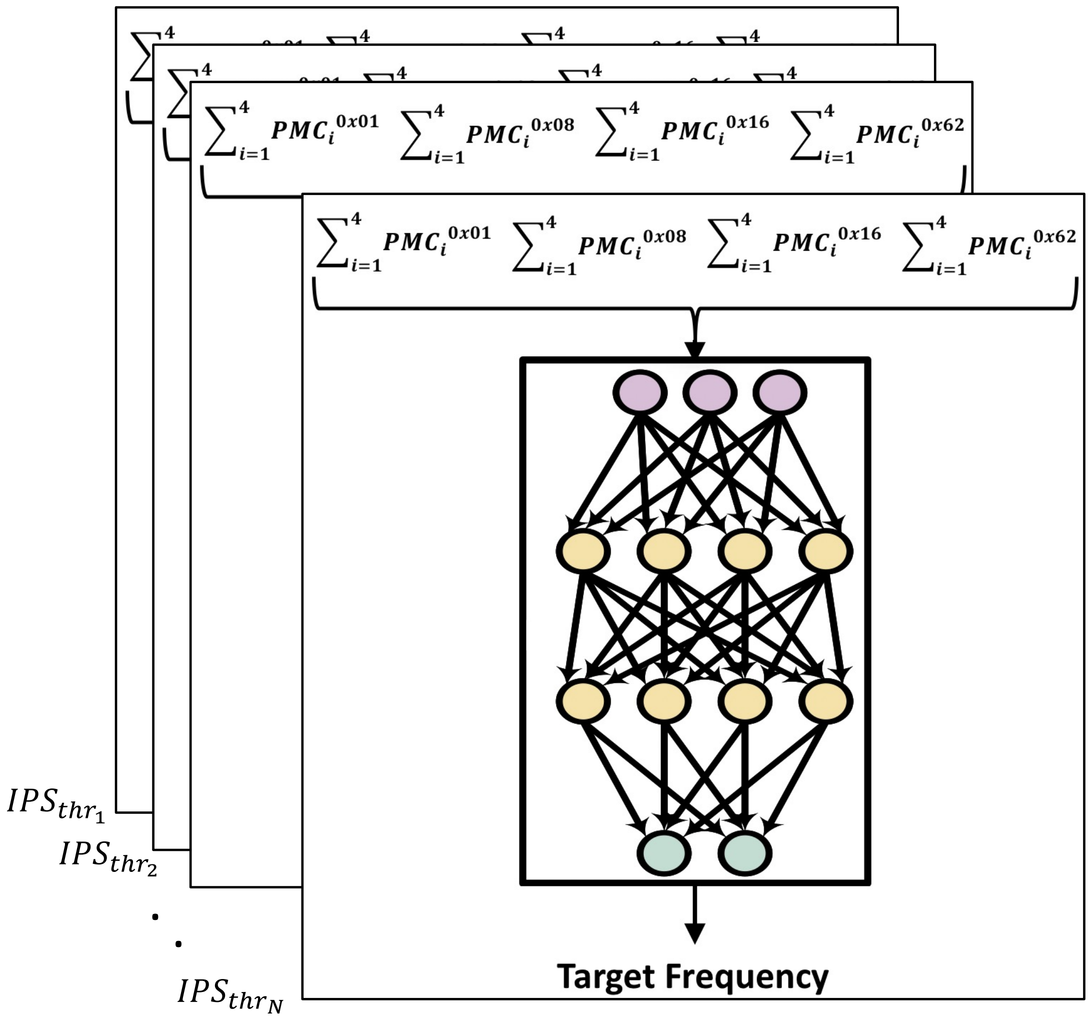
The proposed frequency prediction mechanism can be utilized at runtime by converting the trained neural networks to look-up tables for fast access. By dividing application execution into discrete time quanta, frequency predictions can be made in the intermediate stages of execution. At the end of each time quantum, the runtime manager should check if the required performance threshold IPS_{thr} was satisfied and should scale frequency accordingly. Figure 5.5 depicts a visual representation of the runtime model.

Table 5.1: Comparison of different classifiers

Method	Prediction Accuracy		
	$P_{thr} \leq 5W$	$P_{thr} \leq 6W$	$P_{thr} \leq 7W$
MLP	0.964	0.969	0.957
Tree	0.908	0.894	0.897
CN2 Rule Induction	0.857	0.876	0.931
Naive Bayes	0.851	0.879	0.923
Stochastic Gradient Descent	0.743	0.760	0.839
k-Nearest Neighbors	0.769	0.768	0.825
Logistic Regression	0.786	0.629	0.877

5.3.4 Comparison of classification approaches

Table 5.1 shows that the MLP achieved better classification accuracy, during validation, compared to other classifiers. Even though the number of frequencies is



$\sum_{i=1}^{4} PMC_i^{0x01}$
$\sum_{i=1}^{4} PMC_i^{0x08}$
$\sum_{i=1}^{4} PMC_i^{0x16}$
$\sum_{i=1}^{4} PMC_i^{0x62}$
IPS_{thr_1}
IPS_{thr_2}
IPS_{thr_N}
Target Frequency

Figure 5.5: Expected runtime model

relatively small, our method selects faster the maximum frequency comparing to greedy or bisection approaches as both iterate more frequency values in order to find the optimal one. Due to the nonlinear connection between power and frequency [35], operating at higher frequency levels will increase operational risks. Finally, the MLP inference overhead is approximately 10 μs.

5.4 EVALUATION

Regarding the evaluation of the proposed approach on the Odroid-XU3 board, we tested nine application mixes with varying main memory bandwidth utilization, and we

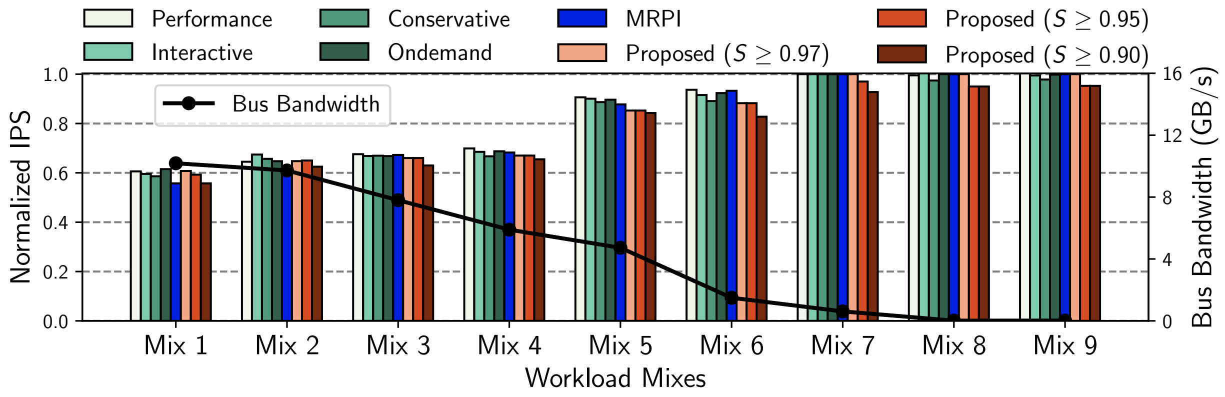


Figure 5.6: Normalized IPS and shared bus bandwidth for all application mixes.

considered three scenarios for the IPC_{thr} (i) maximum IPS drop of 3% ($S \geq 0.97$), (ii) maximum IPS drop of 5% ($S \geq 0.95$), and (iii) maximum IPS drop of 10% ($S \geq 0.90$). In other words, these values of S indicate the minimum performance requirements for our mixes. Each evaluated application mix consists of four simultaneously executing benchmarks from the Polybench [76] and Stream [74] benchmark suites. Finally, we compared the results of our method against: (1) Linux's performance, interactive, conservative, and on-demand governors and (2) the method presented in [37] which selects the processor's clock frequency based upon the Memory Reads per Instruction (MRPI) of each application.

Figure 5.6 depicts the normalized IPS and bus bandwidth (right axis) for the nine application mixes. For ease of representation, the mixes are arranged in increasing order based on their shared-bus bandwidth utilization. Mix #1 is an aggressive, pure memory-intensive mix which has the highest bandwidth utilization on the board. Contrary, the Mix #9 is a pure compute-intensive group with minimum (almost 0) shared bandwidth utilization. The IPS achieved by the Performance governor is used as baseline for Figure 5.6 because it is the most aggressive in terms of frequency scaling. Additionally, Figures 5.7(a)-(c) show the corresponding power consumption, $MIPS/Watt$, and normalized energy consumption respectively.

Memory intensive application groups benefit the most from our proposed approach due to their logarithmic relation between IPS and frequency. Specifically, for the memory-intensive mixes #1 and #2, the proposed approach selected an operating frequency that reduced power consumption by 14.6% and 16.5% (Figure 5.7(a)) and increased $MIPS/Watt$ by 20% and 22% compared to Linux's performance governor. Additionally, the IPS drop was only 2% at worst case, thus satisfying the IPS_{thr} in all three scenarios.

For purely compute-intensive groups such as for Mix #8 and Mix #9 and $S \geq 0.97$, our method has the same behavior as Linux's Performance governor. This happens because the allowable performance drop is very small (only 3%) and any frequency change, below $2.0GHz$, results in higher performance drop. However, if we increase the maximum allowable IPS drop to 5% ($S \geq 0.95$), Mixes #8 and #9 yield 20% and 18% reduction in power consumption, and 24% and 19% increase in the $MIPS/Watt$ respectively, while experiencing an IPS loss of 5% each completely satisfying IPC_{thr}.

For the Mixes #3 to #7 and $S \geq 0.97$, the proposed approach selected a frequency that yields an average gain of 9% reduction in power consumption and 16% increase in $MIPS/Watt$ compared to Linux's Performance governor. Similarly, for a speedup factor of $S \geq 0.95$, Mixes #3 to #7 achieve 12% reduction in power and 22% gain in $MIPS/Watt$ compared to Linux's Performance governor. In all cases, the proposed approach satisfies the minimum performance requirements imposed by the speedup factor.

Finally, compared to the proposed approach, the MRPI-based approach [37] achieves comparable speedup factor and lower power consumption for the purely memory-intensive mixes (#1 and #2). This happens because MRPI selects the frequency of the workload according to the most compute-intensive application in the mix in order to satisfy the application-specific minimum performance requirements. However, if a mix of applications contains even a single compute-intensive application (very low MRPI), the frequency of the cluster is set to the maximum (or near-maximum) value. For that reason MRPI has higher power consumption for mixes #3 to #9.

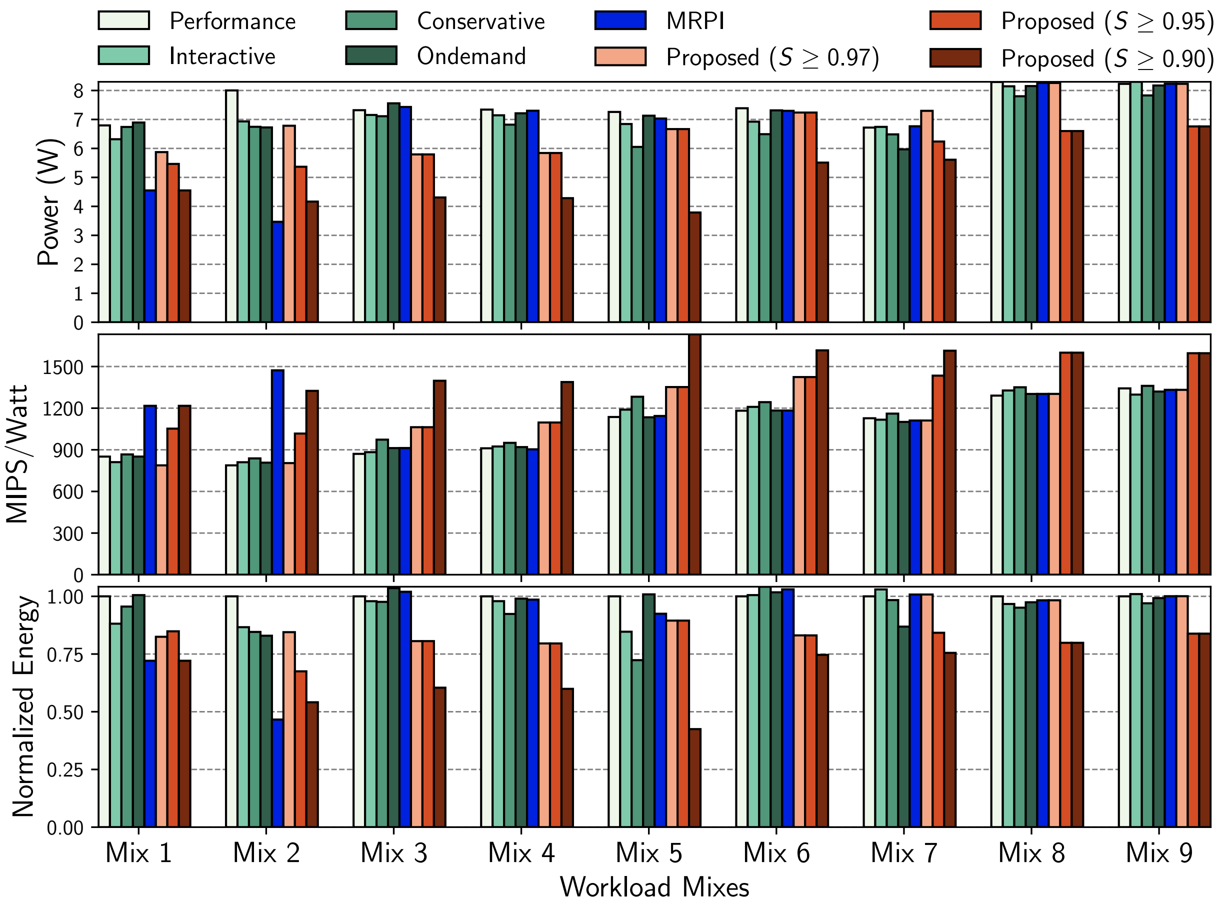


Figure 5.7: (a) Power consumption, (b) MIPS/Watt, and (c) Normalized energy.

5.5 CONCLUSION

The reduction of power consumption in modern CMPs is of utmost importance in order to tackle a wide variety of engineering challenges. The proportion of dark silicon on a chip, premature component aging/failure from heat, and increasing battery life for mobile devices are among the major challenges.

This chapter proposes a machine-learning approach for runtime estimation of power consumption of concurrently executing groups of applications on CMPs. The primary objective of this approach is to estimate the *minimum possible frequency* on which an application group (four concurrent applications) should run in order to satisfy a specified performance threshold IPS_{thr}. By avoiding unnecessary use of higher frequencies (in cases where it would offer no performance gain), significant savings on power consumption can be achieved.

CHAPTER 6
PERFORMANCE & POWER AWARE SCHEDULING

This chapter proposes a methodology to predict both power consumption and performance for groups of concurrently executing applications at all available frequencies of a CMP. The methodology uses a combination of hardware-based application profiling, contention-aware scheduling, and multi-layer perceptron artificial neural networks in order to implement a holistic power and resource-pressure aware scheduling mechanism. It builds upon the machine-learning approach presented in Chapter 5.

6.1 INTRODUCTION

Clustered CMP architectures present system designers with two major challenges. First, since all cores belonging to the same die must operate at the same voltage-frequency level [2, 35], maintaining low power consumption while simultaneously satisfying performance requirements becomes a non-trivial problem. This is due to different types of applications reaching peak computational performance at different clock frequencies. Second, concurrently executing applications suffer performance degradation if their collective resource requirements exceed the total amount of resources available to the CMP [21]. If resource allocation is not carefully considered, performance gains from having multiple cores can be outweighed by the losses due to contention for shared resources among concurrently executing processes.

Predicting system behavior in terms of application performance and power consumption can help run-time resource managers satisfy specific thresholds by proactively adjusting the operating frequency of the CMP. However, current techniques to predict performance and power [92, 93] are based on regression models, which only work for specific voltage-frequency levels and ignore contention effects. Additionally, models that try to predict the impact of shared-resource contention fail to provide consistent results for large numbers of concurrently executing applications as the shared environment becomes more noisy [94, 95].

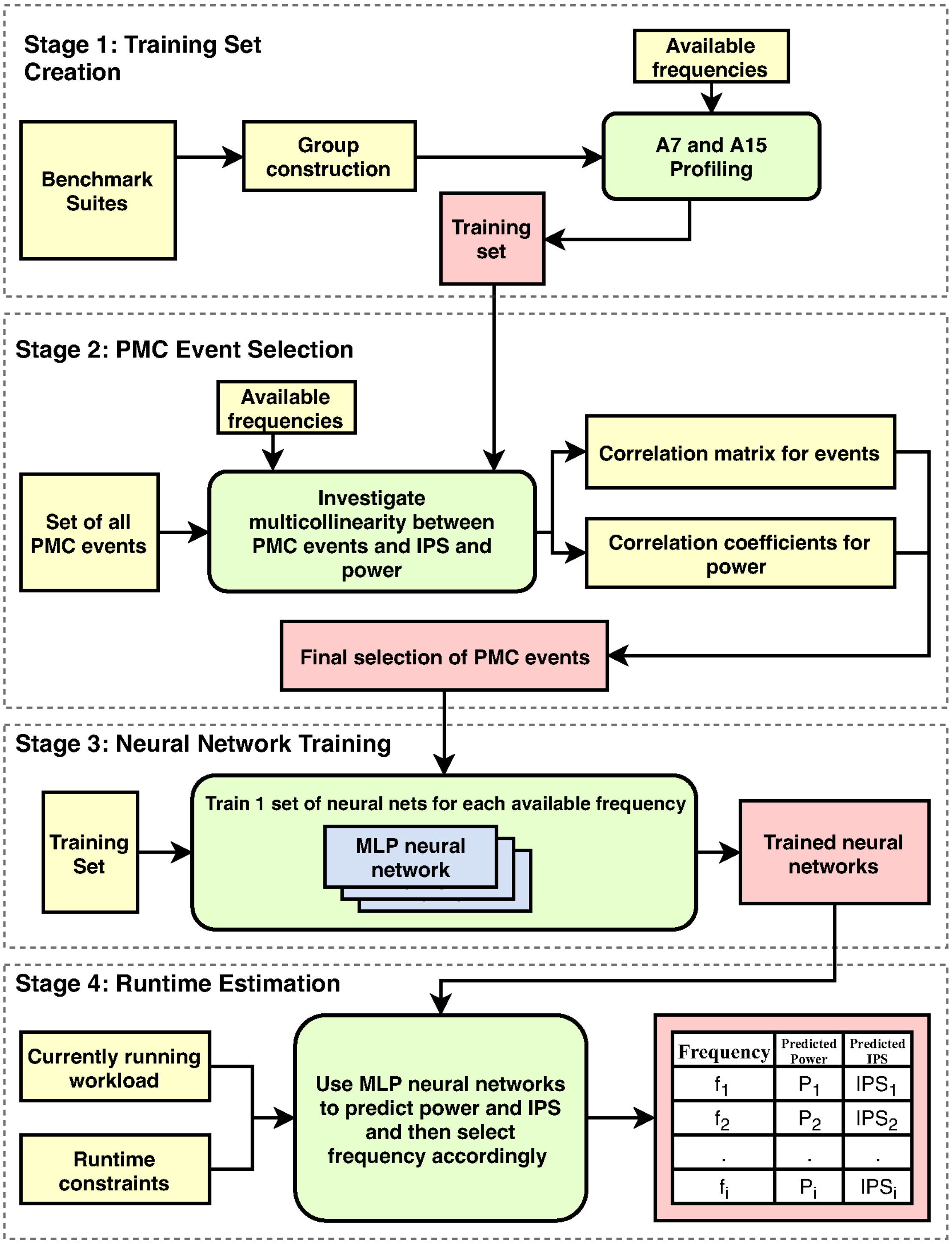
Therefore, we propose a resource management technique that quantifies the execution characteristics of concurrently executing groups of applications for different frequencies of the CMP. This method can then guide run-time resource managers to satisfy specific performance/power thresholds by proactively adjusting the operating frequency of the CMP.

6.2 PROPOSED METHODOLOGY

The proposed methodology aims to utilize regression-based Multi-Layer Perceptron (MLP) artificial neural networks to pro-actively estimate the performance and power consumption of simultaneously executing applications for all frequencies supported by the CMP system. Then, based on these estimates and the required run-time power and performance constraints, the appropriate frequency should be selected. Figure 6.1 depicts an overview of the methodology, which is implemented in four stages: (1) training set creation, (2) PMC event selection, (3) MLP neural network training, and (4) run-time power and performance estimation.

6.2.1 Training Set Creation

The goal of this stage is to create an unbiased training set for our subsequent neural networks. We start by selecting multiple benchmarks from the Polybench [76] and Stream [74] benchmark suites to construct groups of concurrently executing applications that exert varying amounts of pressure on the memory controller and cover as many run-time scenarios as possible. Particularly, we modified the memory-intensive benchmarks in order to have constant memory bandwidth with configurable size. Applications whose performance does not rely upon shared resources (compute-intensive) scale linearly with clock frequency, whereas the performance of memory-intensive benchmarks scales


Stage 1: Training Set Creation
Available frequencies
Benchmark Suites
Group construction
A7 and A15 Profiling
Training set
Stage 2: PMC Event Selection
Available frequencies
Set of all PMC events
Investigate multicollinearity between PMC events and IPS and power
Correlation matrix for events
Correlation coefficients for power
Final selection of PMC events
Stage 3: Neural Network Training
Training Set
Train 1 set of neural nets for each available frequency
MLP neural network
Trained neural networks
Stage 4: Runtime Estimation
Currently running workload
Runtime constraints
Use MLP neural networks to predict power and IPS and then select frequency accordingly

Frequency	Predicted Power	Predicted IPS
f_1	P_1	IPS_1
f_2	P_2	IPS_2
.	.	.
f_i	P_i	IPS_i

Figure 6.1: Overview of the proposed methodology on the Odroid-XU3 board

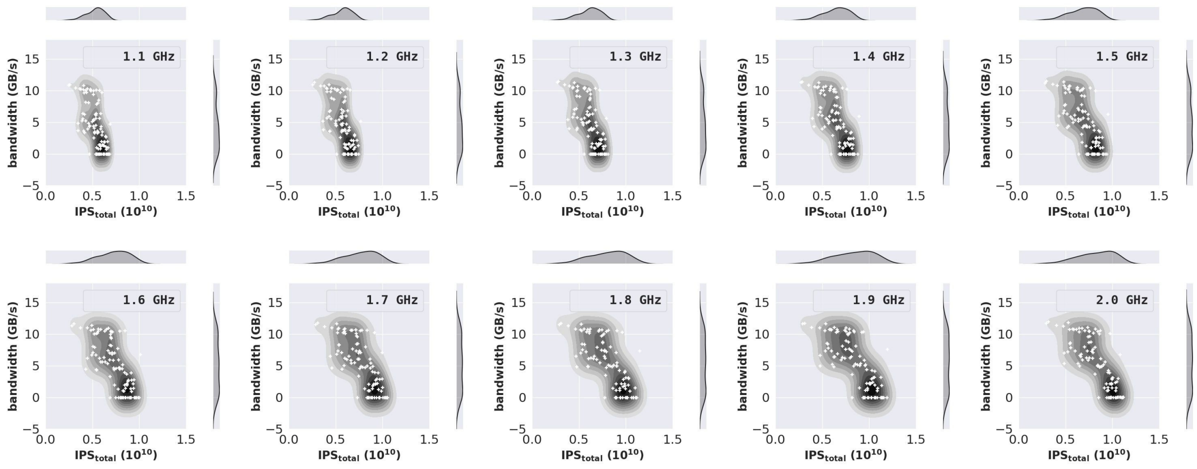


Figure 6.2: Distribution of IPS_{total} and memory bandwidth of the created dataset on the A15 cluster using kernel density estimation. Each group (denoted by a white cross) utilizes all four cores of the cluster

logarithmically.For the A15 cluster, the frequency range was $1.1GHz$ to $2.0GHz$, while for the A7 was $0.5GHz$ to $1.4GHz$. We chose this frequency range for A15 because the memory speed is limited to $933MHz$, thus making impractical all frequencies lower than $1.1GHz$. *For each group, we recorded the values of all the available PMCs (for both clusters) along with power consumption.* Figure 6.2 depicts the distribution of the constructed dataset for the A15 cluster. Each plot visualizes the distribution of the memory bandwidth in GB/s corresponding to IPS_{total} for different frequencies. We want to highlight the following points:

1. for all the evaluated frequencies, the distribution of the memory- and compute-intensive applications remains the same;

2. the pressure on the memory controller is diverse and well balanced (the whole memory bandwidth spectrum is covered);

3. there are memory-intensive groups that saturate the memory controller; and

4. there are pure compute-intensive groups that increase IPS_{total} linearly with frequency.

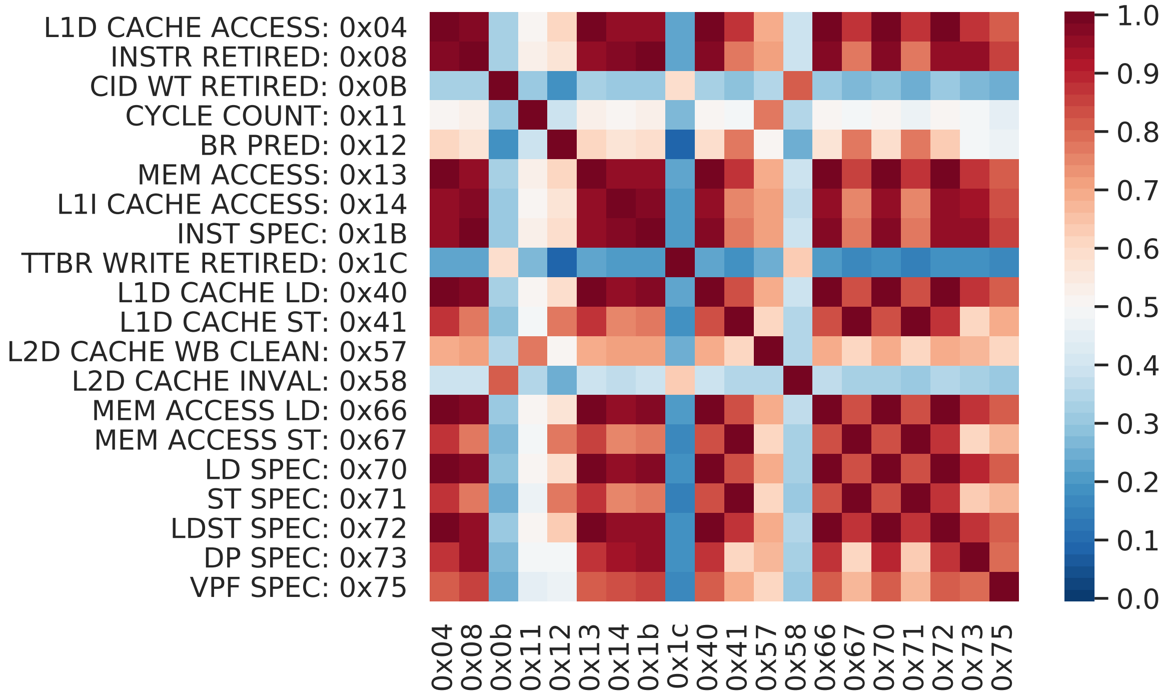


Figure 6.3: Pearson's Product Moment Correlation for a subset of A15's PMCs.

Overall, the values of the PMC events depend on the actual workload. In order to build a generic estimator that does not depend on the individual characteristic of an application, we constructed our training set in a manner that represents as broad a range of execution characteristics as possible. We accomplish this by creating application groups that apply resource pressure *over the entire spectrum* of memory hierarchy for both the A15 and A7 clusters. Since shared-resource utilization is one of the primary determinants of application performance in CMPs, our dataset is well-equipped to capture execution characteristics for groups of applications not explicitly included in the training set.

6.2.2 PMC Selection

We follow the approach presented in Chapter 5. Due to the limited number of PMCs, one of the most important factors when choosing a set of events is the degree of multicollinearity between them [96]. Multicollinearity occurs when two or more

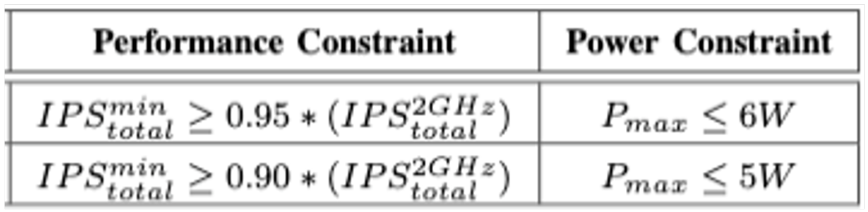
Table 6.1: Power and IPS_{total} Error Percentage for A15 and A7 Clusters

	Power			**IPS**		
	Layers			**Layers**		
Neurons	**1**	**2**	**3**	**1**	**2**	**3**
	A7					
32	23.26	8.35	5.89	13.81	7.63	8.25
64	25.89	7.59	4.50	17.81	8.27	7.98
128	10.05	6.21	4.40	11.01	8.39	8.43
	A15					
32	23.98	16.11	18.94	9.58	7.04	10.71
64	13.45	5.89	5.80	12.20	5.02	5.78
128	11.01	5.44	5.61	7.64	5.65	5.41

independent variables in a linear-regression model have a substantial amount of correlation between them. In the context of PMC events, two or more events with a high degree of multicollinearity will record ‘overlapping’ information about the state of the processor and the behavior of workloads. For example, choosing PMC events based solely on their correlation with average CPU power results in a poor model as the PMC events that correlate well with power also correlate well with each other [96]. Therefore, we follow the approach presented in Chapter 5 that uses Pearson’s product-moment estimation as well as other methods such as Ward’s variance minimization for hierarchical clustering. A subset of Pearson’s correlation results for the A15 is depicted in Figure 6.3. A good selection of events can be made by choosing the events with correlation values closest to 0.

6.2.3 MLP Training

The goal of this stage is to determine the best architecture for an MLP which will be tuned to satisfy both performance and power criteria while also keeping execution overhead low. Table 6.1 shows the error percentages obtained from testing MLPs with different numbers of layers and neurons.



Performance Constraint	Power Constraint
$IPS_{total}^{min} \geq 0.95 * (IPS_{total}^{2GHz})$	$P_{max} \leq 6W$
$IPS_{total}^{min} \geq 0.90 * (IPS_{total}^{2GHz})$	$P_{max} \leq 5W$

Figure 6.4: Example constraints for A15 cluster

6.2.4 Run-time power and performance estimation

The proposed methodology will be implemented as a user-space scheduler, on top of the exploration tool presented in [96]. The trained MLPs will be triggered based on the current operating frequency. The set of trained MLP neural networks should output the frequency which best satisfies the performance and power thresholds imposed by the user. Figure 6.4 shows an example of performance and power constraints for the A15 cluster.

6.3 EXPERIMENTAL RESULTS

We evaluate our proposed approach on the quad-core ARM Cortex-A15 and Cortex-A7 clusters of a Samsung Exynos 5422 Octa SoC [97], embedded onto a Hardkernel Odroid-XU3 development board [91] using Linux kernel version 3.10. Our motivation for utilizing this specific combination of processor and board is two-fold: 1. the Samsung Exynos 5422 processor with octa-core `big.LITTLE` architecture is similar to many of the latest ARM-based embedded systems which contain clusters of both `big` (power-hungry) and `LITTLE` (power-efficient) cores [98,99], and 2. the built-in Power Monitoring Unit (PMU) of the Odroid-XU3 allows for accurate measurement of power consumption for both the A15 and A7 clusters separately. This functionality was discontinued for newer models of Odroid/ Hardkernel development boards [100], resulting in XU3 being the only commercially available board of the Odroid family that allows for power measurement in a `big.LITTLE`-style processor. Additionally, we evaluate our method against the methods described in [36,39,101], which are all validated on the Odroid-XU3 board running Linux kernel 3.10 as well. This kernel version is the only one that allows us to access the PMU. Thus, by utilizing the same board and Linux kernel version, we ensure consistency and fairness in the comparison of evaluated results.

Our proposed approach comprises two steps. In the first step, we set separate power and performance thresholds and we measured whether the selected frequency (after MLPs estimations) violated the constraints and by how much. In the second step, we considered joint power and performance constraints compared with Linux's governors and three more resource managers.

6.3.1 Workload description

We evaluate our proposed approach using 14 application mixes for the A15 cluster and 10 application mixes for the A7 cluster. Each application mix consists of four concurrently executing benchmarks taken from the Polybench [76] and Stream [74] benchmark suites. We selected these particular mixes for evaluation since they represent a broad spectrum of shared-resource utilization for the A15 and A7 clusters. Tables 6.2 and 6.3 list the actual benchmarks and their corresponding datasets that comprise each mix for the A15 and A7 clusters accordingly.

Overall our goal was to create a testing set of mixes comprising random benchmarks, in order to provide operating conditions with varying pressure on the shared resources. To that end, Figures 6.5 and 6.6 depict the memory bandwidth (shared-resource utilization) for each of these mixes. In both cases, it can be observed that the selected mixes show great variation in memory activity. Particularly for the A15 cluster (Figure 6.5), some mixes are very memory-intensive (e.g., Mixes 13, 14), some consist of a combination of memory- and compute-intensive benchmarks (e.g., Mixes 1, 2) and some comprise of more compute-intensive benchmarks (e.g., Mixes 9, 10). Accordingly for the A7 cluster (Figure 6.6), some mixes contain fully compute-intensive workloads (e.g., Mixes 1, 3, 9), mixes of compute- and memory-intensive workloads (e.g., Mixes 2, 6) and mixes consisting of entirely memory-intensive benchmarks (e.g., Mix 8). Note that the max memory bandwidth on the Odroid-XU3 platform is 14.9 GB/s.

Table 6.2: Benchmarks & datasets used for the A15 cluster.

Mix	**App1**	**App2**	**App3**	**App4**
1	gesummv_32M	deriche_16K	2mm_16K	stream_6M
2	gesummv_32M	gemver_64M	bicg_64M	stream_6M
3	heat-3d_32M	bicg_64M	jacobi-2d_32M	bicg_32M
4	heat-3d_32M	fdtd-2d_64M	gemver_64M	bicg_32M
5	jacobi-1d_128M	gemver_64M	correlation_16K	jacobi-2d_32M
6	jacobi-1d_128M	seidel-2d_16K	bicg_64M	nussinov_16K
7	jacobi-1d_64M	jacobi-2d_32M	adi_16K	gemm_128M
8	jacobi-2d_64M	gemver_64M	jacobi-2d_32M	3mm_16K
9	trisolv_128M	gemver_64M	cholesky_16K	gemver_128M
10	trisolv_128M	heat-3d_32M	deriche_32M	nussinov_16K
11	trisolv_128M	jacobi-1d_128M	deriche_32M	bicg_64M
12	trisolv_128M	jacobi-1d_128M	fdtd-2d_32M	bicg_32M
13	stream_22M	stream_22M	gesummv_32M	stream_22M
14	stream_22M	stream_22M	stream_22M	stream_22M

Table 6.3: Benchmarks & datasets used for the A7 cluster.

Mix	**App1**	**App2**	**App3**	**App4**
1	3mm_16K	syrk_16K	adi_16K	floyd-war_16K
2	bicg_32M	trisolv_64M	trisolv_128M	heat-3d_32M
3	deriche_16K	correlation_16K	adi_16K	floyd-war_16K
4	deriche_64M	stream_4M	symm_16K	covariance_16K
5	gemm_128M	gram_16K	fdtd-2d_16K	heat-3d_32M
6	gesummv_16K	jacobi-1d_16K	stream_6M	ludcmp_16K
7	jacobi-1d_16K	jacobi-2d_16K	heat-3d_16K	bicg_64M
8	jacobi-2d_32M	gemver_128M	gesummv_32M	deriche_64M
9	symm_16K	atax_16K	jacobi-1d_16K	3mm_16K
10	trisolv_32K	atax_16K	2mm_16K	cholesky_16K

6.3.2 Evaluation on the A15 cluster

Regarding evaluation on the A15 cluster, we considered the two scenarios depicted in Table 6.4. Regarding the tested applications, we constructed fourteen groups of

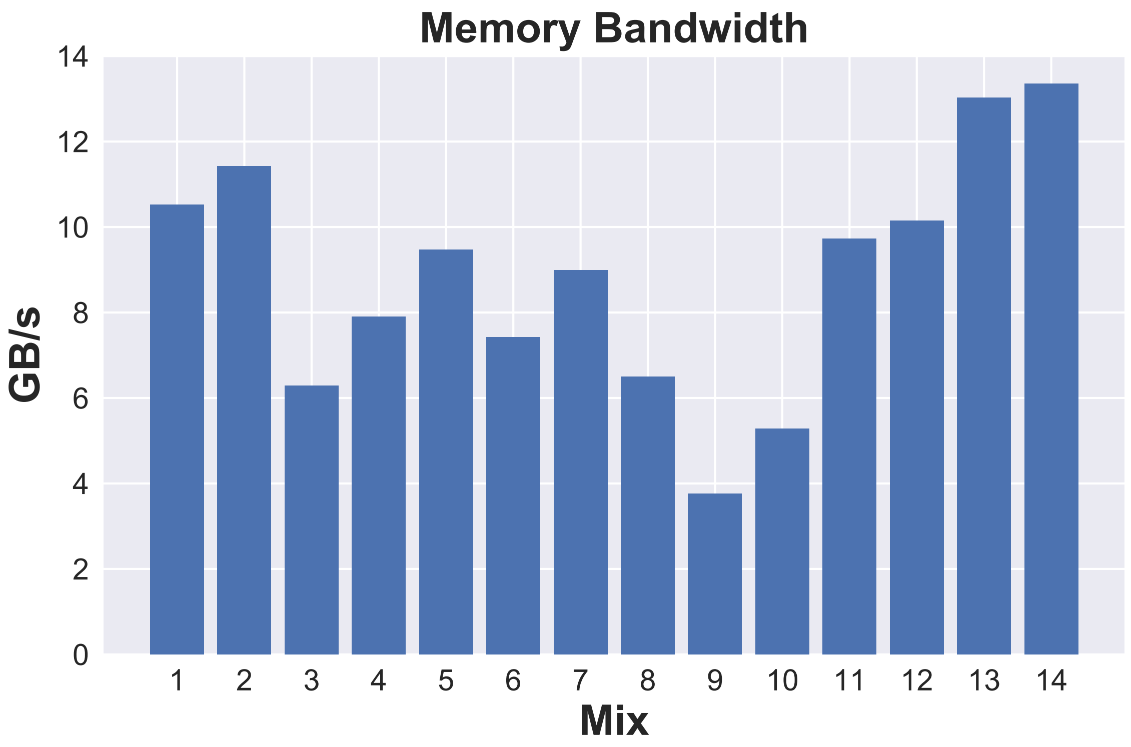


Figure 6.5: Memory bandwidth (shared-resource utilization) for applications mixes used for the A15 cluster.

simultaneously executing applications which apply varying amounts of pressure on the cluster's available resources (Table 6.2). Similarly, these groups are not included in the training set. Initially, the execution starts at $2GHz$ and the run-time manager uses PMC counts as input to the MLPs in order to select a lower frequency that best satisfies both constraints for each scenario.

We compare the achieved IPS_{total}, power consumption, and power-efficiency $(MIPS/Watt)$ of all groups against other approaches. We compare with the MRPI

Table 6.4: Joint run-time constraints for different scenarios on A15 cluster

Scenario	**Performance Constraint**	**Power Constraint**
Scenario #1	$IPS_{total}^{min} \geq 0.95 * (IPS_{total}^{2GHz})$	$P_{max} \leq 6W$
Scenario #2	$IPS_{total}^{min} \geq 0.90 * (IPS_{total}^{2GHz})$	$P_{max} \leq 5W$

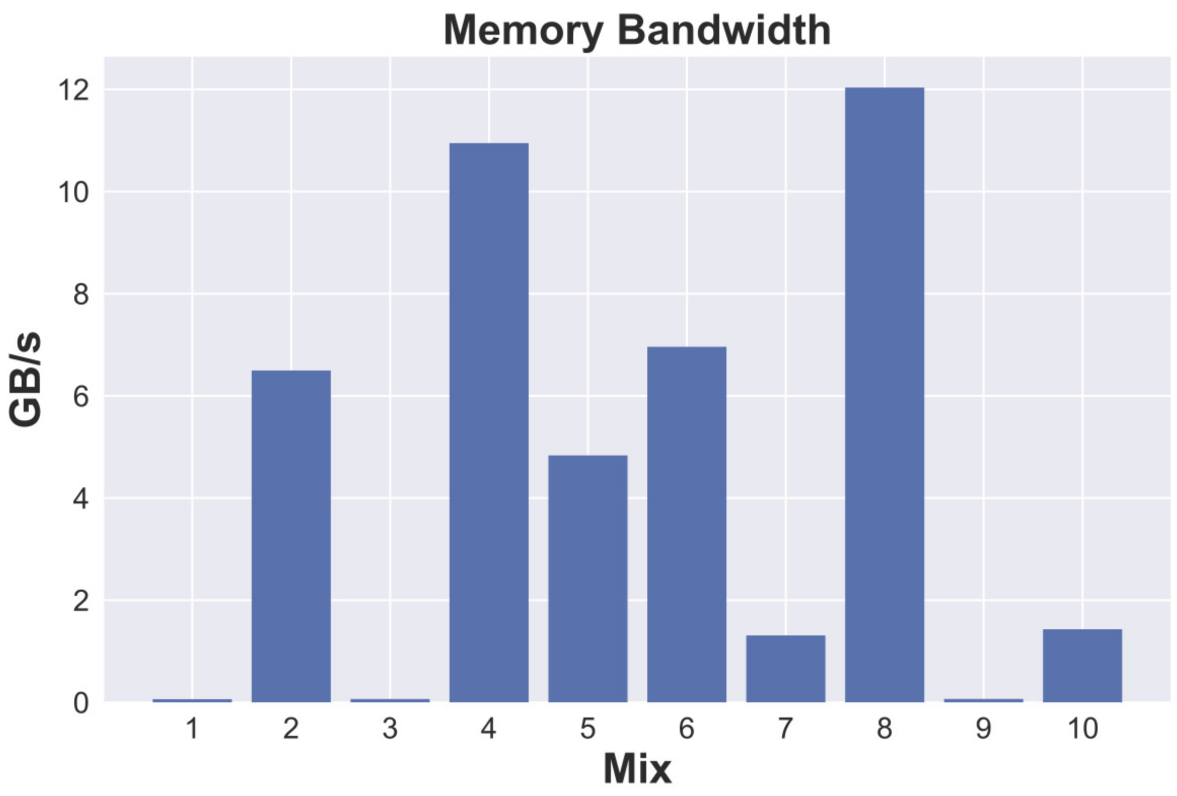


Figure 6.6: Memory bandwidth (shared-resource utilization) for applications mixes used for the A7 cluster.

approach [36] which uses Memory Reads Per Instruction as a metric for classification. Additionally, we measure the outcome of the proposed method with the results given from the MLP approach described in [39]. This method supports the training of separate neural networks for different power thresholds.

Added into this analysis is also the SPARTA binning classification method presented in [101]. Finally, we also compare the aforementioned methods with the returned results from the governors supported by Linux's completely fair scheduler: Performance, Interactive, Conservative, and Ondemand.

Figure 6.7(a) depicts the comparison of all approaches regarding power consumption on the A15 cluster. From the graph, we see that the proposed methodology managed to satisfy the power constraints for all groups in both scenarios. Respectively, Figure 6.7(b) shows the comparison of all approaches based on the achieved performance (IPS). The

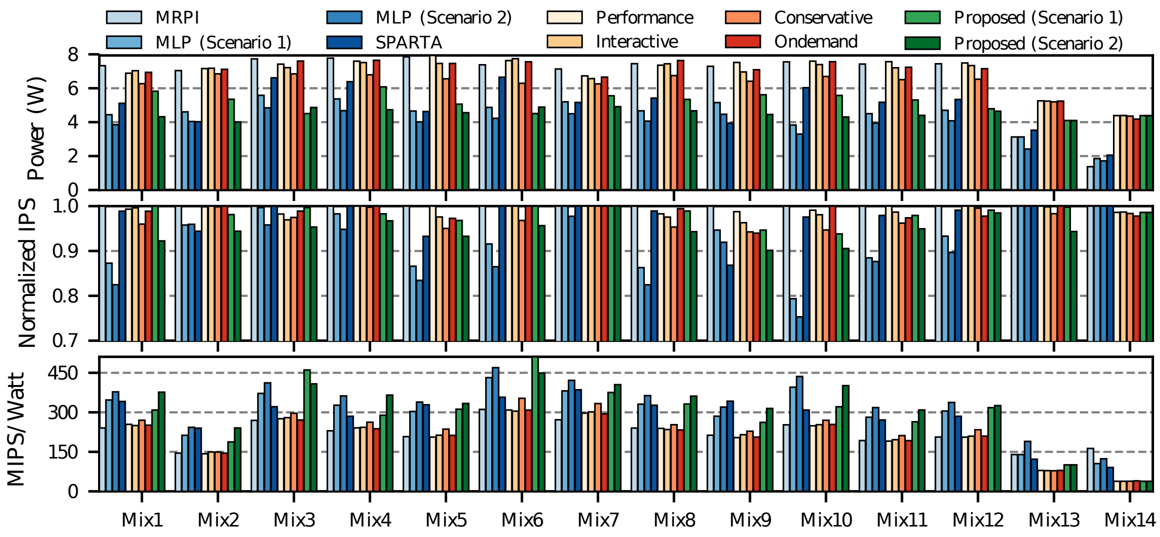


Figure 6.7: (a) Power consumption; (b) performance in terms of normalized IPS; and (c) power efficiency in terms of $MIPS/WATT$ for 14 random application mixes on the A15 cluster under two constraint scenarios (Table 6.4). Each mix consist of four concurrent executing applications

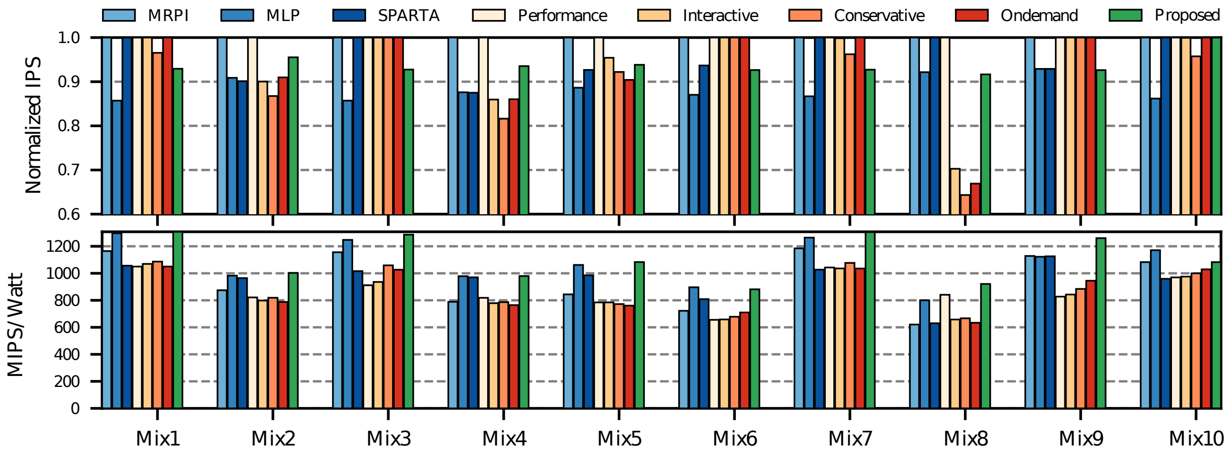


Figure 6.8: (a) Performance in terms of normalized IPS and (b) power efficiency in terms of $MIPS/WATT$ for 14 random application mixes on the A7 cluster for the selected scenario. Each mix consist of four concurrent executing applications

baseline for our experiments is the IPS monitored when each mix is executed at the highest possible frequency ($2GHz$).

The MLP method [39] manages to satisfy the power constraints in both scenarios, since there are two different neural networks trained for different target frequencies to match the power thresholds and is also characterized by power-awareness. The trade-off of this method is that due to the lower predicted frequencies, the performance that is finally achieved is also low. The authors do not use the absolute value of the power thresholds for training, but a lower value which is referenced as safety net. For that reason their predictions are more pessimistic. This method fails to satisfy Scenario #1 for 8 out of the 14 mixes (regarding the IPS threshold). It also violates Scenario #2 for half of the mixes.

The MRPI method [36] employs the metric of memory reads per instruction to classify groups to frequencies, specifically the fraction. Therefore, it achieves lower power observations for mixes that contain memory-intensive workloads. As a result, this method achieves high performance in all mixes, while resulting in high power observations for 12 out of the 14 mixes. Overall, the MRPI method violates both of the Scenarios for these 12 mixes.

The classification method used in SPARTA [101] has two levels of bins, a bin regarding memory-boundness and a bin regarding compute-boundness. SPARTA violates Scenario #1 in 6 of the 14 mixes and only satisfies Scenario #2 in 4 of the cases. However, in most of the the violating cases, SPARTA doesn't exceed the predefined thresholds of each corresponding scenario by significant percentages.

The proposed method completely satisfies both scenarios. Figure 6.7(b) shows that even though our proposed methodology lowered the frequency in order to satisfy the power thresholds (Figure 6.7(a)), it managed to select such a value so as to satisfy the IPS requirements of Table 6.4 as well. Compared to Linux's performance governor and completely fair scheduler, our methodology was able to reduce the overall power consumption of the A15 cluster by an average of 30.5%, with the highest power saving percentage being 51.5% for Scenario #1. Respectively, our methodology reduced the power consumption of the cluster by an average of 42.5%, with the highest power saving percentage being 56.3% for Scenario #2.

Finally, Figure 6.7(c) depicts the comparison regarding power efficiency (MIPS/Watt).

As expected, the proposed approach achieves better efficiency than the Linux governors. This is because Linux governors do not account for performance bottlenecks caused by saturation on memory bandwidth. Contrary, with the usage of various workloads as training set for the MLPs, we are able to detect at run-time the bandwidth saturation and estimate the lowest possible frequency at which we still retain the required performance and satisfy also power thresholds. Compared to Linux's Performance governor, our methodology achieved an average improvement of 29.3% and 38% regarding power efficiency for scenarios #1 and #2 respectively, while also achieving the minimum required performance. The MLP approach also achieves high $MIPS/Watt$ observations, since it is a energy-aware method that sacrifices performance for overall low power. Specifically, in power efficiency terms the MLP method performs better by 11% and 14% for Scenario #1 and Scenario #2 respectively. However, our method manages to achieve high power efficiency while satisfying all performance constraints defined in both scenarios. The MRPI method shows higher power efficiency when the mixes are characterized by more memory-intensive workloads since in this case it classifies workloads to lower frequencies and consequently results in the satisfaction of the set performance thresholds. Compared to MRPI, our method achieves an average of 17.1% and 25.8% higher power efficiency for Scenario #1 and Scenario #2 accordingly. The SPARTA method accomplishes 6% higher power efficiency than our method for Scenario #1, while we result in 3% higher power efficiency than SPARTA for Scenario #2.

6.3.3 Evaluation on the A7 cluster

Since A7 is a low power processor, we consider power improvements in this cluster to be of insignificant value, since all resulting power values are below 1 $Watt$. For this reason, we choose just one Scenario for the A7 cluster as follows: <u>Scenario 1:</u> $IPS_{total}^{min} \geq 0.90 * IPS_{total}^{1.4GHz}$). For the MLP method, the same neural network architectures were used with the corresponding PMC events of the A7 cluster. However in this case since we do not have separate power threshold use cases, there was only one neural network trained and evaluated. The MRPI method uses a similar MRPI-based classification method for the A7 cluster which is also being used in this evaluation. The corresponding PMC events used by SPARTA on the binning approach were also used on the A7 cluster for this evaluation. Likewise, we again compare our methodology with the aforementioned methods and the Linux governors.

Figure 6.8(a) shows the performance results for all methods. In all mixes (Table 6.3) our methodology manages to satisfy the scenario defined for the A7. As aforementioned, A7 is a low power cluster and all of the tested groups resulted in power values less than 1W. Therefore, we consider the comparison of raw power values redundant.

The MRPI method manages to also achieve 100% performance satisfaction based on Scenario 1. On the contrary, the MLP method violated the performance threshold on the majority of the groups, while SPARTA only violated the performance threshold on one occasion. The low observed performance of the MLP method is again justified by the fact that this is a power-aware method, hence achieving low power alongside with low performance observations on both clusters.

Figure 6.8(b) depicts the comparison of all methods based on power efficiency terms. Overall the $MIPS/Watt$ values are higher in this case since the power values under comparison are very low. As expected, the MLP approach achieves once again high $MIPS/Watt$ observations since the resulting power of the chosen operating frequencies is in each group the lowest of all methods. The MRPI method achieves equivalent or higher power efficiency when compared to the Performance governor. Our method obtains an average of 23.9% overall increase in power efficiency compared to the Performance governor as well. Overall, compared to the other three methods, the proposed methodology showed an average increase of 11.4% in power efficiency on the A7 cluster.

6.4 CONCLUSION

Modern Chip Multi-Processors (CMPs) are required to be increasingly power efficient while also offering higher performance and lower costs. A combination of Dynamic Voltage-Frequency Scaling (DVFS) and sophisticated resource-aware scheduling is needed to address the underlying problem of maximizing performance-per-Watt of CMP architectures.

In this chapter, we propose a methodology to predict the power consumption and performance for groups of concurrently executing applications at all available frequencies of a CMP. The methodology uses a combination of hardware-based application profiling, contention-aware scheduling, and artificial neural networks. Experimental results on an Odroid-XU3 board demonstrate an increase in average performance per Watt of 30.5% (A15 cluster) and 11.4% (A7 cluster) over Linux's Completely Fair Scheduler (CFS) and power governors. In addition, our methodology outperforms three state-of-the-art resource managers, yielding the highest performance-per-Watt in all evaluated use cases.

CHAPTER 7
CONCLUSION

7.1 REMARKS

Chip Multi-Processors were first developed to address the growing shortcomings of single-core architectures in the late 1990's/early 2000's. Among the major challenges to improving single-thread throughput were the diminishing returns on application performance at higher clock frequencies and the growing difficulty of heat dissipation due to unprecedented device densities on the die [2]. Manufacturers tried to "squeeze out" additional performance from existing architectures by increasing parallelism at the instruction level (ILP). Among the developments were deep execution pipelines, superscalar architectures, Very Long Instruction Word (VLIW) architectures, and proprietary systems such as Explicitly Parallel Instruction Computing (by Intel) [5]. While these approaches improved single-thread performance, they came at the cost of greater hardware complexity and a corresponding increase in power consumption. The approach of increasing ILP through architectural additions would eventually yield diminishing returns on application performance.

CMPs addressed the shortcomings of single-core architectures by integrating multiple simpler *cores* onto a single die that share certain computing resources among them such as last-level caches, data buses, and main memory. This enabled architectural simplicity while also boosting performance for multi-threaded applications. However, a major trade-off associated with this approach is that concurrently executing applications incur performance degradation if their collective resource requirements exceed the total amount of resources available to the system. Without dynamic resource-aware scheduling methodologies, the potential performance gain from having multiple cores can be outweighed by the losses due to contention for allocation of shared resources. Additionally, CMPs with inbuilt dynamic voltage-frequency scaling (DVFS) mechanisms may try to compensate for the performance bottleneck by scaling to higher clock frequencies. For performance degradation due to shared-resource contention, this does not necessarily improve performance but does ensure a significant penalty on power consumption due to the quadratic relation of electrical power and voltage ($P_{dynamic} \propto V^2 \cdot f$).

This dissertation presented novel methodologies for balancing the competing requirements of high performance, fairness of execution, and enforcement of priority, while also ensuring overall power efficiency of CMPs. Specifically, we (1) Analyzed the problem of resource interference during concurrent process execution and propose two fine-grained scheduling methodologies for improving overall performance and fairness, (2) Developed an approach for enforcement of priority (i.e., minimum performance) for specific processes while avoiding resource starvation for others, and (3) Presented a machine-learning approach for maximizing the power efficiency (performance-per-Watt) of CMPs through estimation of a workload's performance and power consumption limits at different clock frequencies.

REFERENCES

[1] "microprocessor-trend-data/42yrs/42-years-processor-trend.pdf at master · karlrupp/microprocessor-trend-data · github," https://github.com/karlrupp/microprocessor-trend-data/blob/master/42yrs/42-years-processor-trend.pdf, (Accessed on 09/14/2023).

[2] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Power challenges may end the multicore era," *Commun. ACM*, vol. 56, no. 2, pp. 93–102, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408797

[3] "Spec benchmarks and tools," https://www.spec.org/benchmarks.html, (Accessed on 04/04/2024).

[4] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: The end of the road for conventional microarchitectures," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 248–259. [Online]. Available: http://doi.acm.org/10.1145/339647.339691

[5] M. Schlansker and B. Rau, "Epic: Explicitly parallel instruction computing," *Computer*, vol. 33, no. 2, pp. 37–45, 2000.

[6] "31 stages: What's this, baskin robbins? - intel's pentium 4 e: Prescott arrives with luggage," https://www.anandtech.com/show/1230/3, (Accessed on 10/25/2023).

[7] L. MENABREA, C. Babbage, A. Lovelace, and A. L, *Sketch of the Analytical Engine invented by Charles Babbage ... with notes by the translator. Extracted from the 'Scientific Memoirs,' etc. [The translator's notes signed: A.L.L. ie. Augusta Ada King, Countess Lovelace.]*. R. & J. E. Taylor, 1843. [Online]. Available: https://books.google.com/books?id=hPRmnQEACAAJ

[8] "Ibm100 - power 4 : The first multi-core, 1ghz processor," https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/, (Accessed on

09/14/2023).

[9] T. Marinakis, A. Haritatos, K. Nikas, G. Goumas, and I. Anagnostopoulos, “An efficient and fair scheduling policy for multiprocessor platforms,” in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2017, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/8050758

[10] I. Galanis, T. Marinakis, and I. Anagnostopoulos, “Workload-aware management targeting multi-gateway internet-of-things,” in *Proceedings of the International Conference on Omni-Layer Intelligent Systems.* ACM, 2019, pp. 110–115.

[11] J. S. Koduri and I. Anagnostopoulos, “Spa: Simple pool architecture for application resource allocation in many-core systems,” in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1364–1368. [Online]. Available: https://ieeexplore.ieee.org/document/8342225

[12] D. Olsen and I. Anagnostopoulos, “Performance-aware resource management of multi-threaded applications on many-core systems,” in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, ser. GLSVLSI ’17. New York, NY, USA: ACM, 2017, pp. 119–124. [Online]. Available: http://doi.acm.org/10.1145/3060403.3060426

[13] G. E. Suh, S. Devadas, and L. Rudolph, “A new memory monitoring scheme for memory-aware scheduling and partitioning,” in *Proceedings Eighth International Symposium on High Performance Computer Architecture*, ser. HPCA ’02. Washington, DC, USA: IEEE Computer Society, Feb 2002, pp. 117–128. [Online]. Available: http://dl.acm.org/citation.cfm?id=874076.876484

[14] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, “L1-bandwidth aware thread allocation in multicore smt processors,” in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 123–132. [Online]. Available: http://dl.acm.org/citation.cfm?id=2523721.2523741

[15] S. S. Jha, W. Heirman, A. Falcón, J. Tubella, A. González, and L. Eeckhout, “Shared

resource aware scheduling on power-constrained tiled many-core processors," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '16. New York, NY, USA: ACM, 2016, pp. 365–368. [Online]. Available: http://doi.acm.org/10.1145/2903150.2903490

[16] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, ser. ISLPED '07. New York, NY, USA: ACM, 2007, pp. 38–43. [Online]. Available: http://doi.acm.org/10.1145/1283780.1283790

[17] X. Wang and J. F. Martínez, "Xchange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 113–125. [Online]. Available: https://ieeexplore.ieee.org/document/7056026

[18] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 318–329. [Online]. Available: https://doi.org/10.1109/MICRO.2008.4771801

[19] R. P. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas, "Using multiple input, multiple output formal control to maximize resource efficiency in architectures," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 658–670. [Online]. Available: https://ieeexplore.ieee.org/document/7551430

[20] T. Marinakis, A.-H. Haritatos, K. Nikas, G. Goumas, and I. Anagnostopoulos, "An efficient and fair scheduling policy for multiprocessor platforms," in *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 1–4.

[21] T. Marinakis and I. Anagnostopoulos, "Performance and fairness improvement on cmps considering bandwidth and cache utilization," *IEEE Computer Architecture*

Letters, vol. 18, no. 2, pp. 1–4, 2019.

[22] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Bandwidth-aware on-line scheduling in smt multicores," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 422–434, 2016.

[23] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory bandwidth management for efficient performance isolation in multi-core platforms," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 562–576, 2015.

[24] A. Jaleel, H. H. Najaf-Abadi, S. Subramaniam, S. C. Steely, and J. Emer, "Cruise: cache replacement and utility-aware scheduling," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 249–260.

[25] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 104–117.

[26] Y. Kim, A. More, E. Shriver, and T. Rosing, "Application performance prediction and optimization under cache allocation technology," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1285–1288.

[27] J. Arora, E. Tovar, and C. Maia, "Shared resource contention aware schedulability analysis for multiprocessor real-time systems," in *Design, Automation and Test in Europe Conference (DATE 2023)*, 2023.

[28] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 269–279.

[29] G. Durrieu, M. Faugère, S. Girbal, D. G. Pérez, C. Pagetti, and W. Puffitsch, "Predictable flight management system implementation on a multicore processor," in *Embedded Real Time Software (ERTS'14)*, 2014.

[30] C. Maia, L. Nogueira, L. M. Pinho, and D. G. Pérez, "A closer look into the aer model," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2016, pp. 1–8.

[31] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, 2022.

[32] R. Xu, R. Kumar, P. Wang, P. Bai, G. Meghanath, S. Chaterji, S. Mitra, and S. Bagchi, "Approxnet: Content and contention-aware video object classification system for embedded clients," *ACM Trans. Sen. Netw.*, vol. 18, no. 1, oct 2021. [Online]. Available: https://doi.org/10.1145/3463530

[33] K. Fu, W. Zhang, Q. Chen, D. Zeng, X. Peng, W. Zheng, and M. Guo, "Qos-aware and resource efficient microservice deployment in cloud-edge continuum," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 932–941.

[34] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 270–282. [Online]. Available: https://doi.org/10.1145/3387514.3405868

[35] B. Donyanavard *et al.*, "Gain scheduled control for nonlinear power management in cmps," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 921–924.

[36] B. K. Reddy, A. K. Singh, D. Biswas, G. V. Merrett, and B. M. Al-Hashimi, "Inter-cluster thread-to-core mapping and dvfs on heterogeneous multi-cores," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 369–382, July 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8051086

[37] B. K. Reddy, G. V. Merrett, B. M. Al-Hashimi, and A. K. Singh, "Online concurrent workload classification for multi-core energy management," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 2018, pp. 621–624. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8342084

[38] I. Anagnostopoulos, J.-M. Chabloz, I. Koutras, A. Bartzas, A. Hemani, and D. Soudris, "Power-aware dynamic memory management on many-core platforms utilizing dvfs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 1s, p. 40, 2013.

[39] T. Marinakis, S. Kundan, and I. Anagnostopoulos, "Meeting power constraints while mitigating contention on clustered multi-processor system," *IEEE Embedded Systems Letters*, 2019.

[40] M. Mohammad and I. Anagnostopoulos, "Drop: Distributed run-time and power constraint mapping for many-core systems," in *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS).* IEEE, 2018, pp. 245–248.

[41] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on.* IEEE, 2018, pp. 104–117.

[42] F. Romero and C. Delimitrou, "Mage: Online and interference-aware scheduling for multi-scale heterogeneous systems," in *27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–13.

[43] N. Kulkarni, F. Qi, and C. Delimitrou, "Leveraging approximation to improve datacenter resource efficiency," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 171–174, 2018.

[44] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth*

International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 107–120.

[45] N. Kulkarni, F. Qi, and C. Delimitrou, "Pliant: Leveraging approximation to improve datacenter resource efficiency," in *2019 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2019, pp. 159–171.

[46] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 19–33.

[47] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. ACM, 2010, pp. 129–142.

[48] A.-H. Haritatos, N. Papadopoulou, K. Nikas, G. I. Goumas, and N. Koziris, "Contention-aware scheduling policies for fairness and throughput," in *COSH@HiPEAC*, 2016.

[49] S. Kundan, O. Spantidi, and I. Anagnostopoulos, "Online frequency-based performance and power estimation for clustered multi-processor systems," *Computers & Electrical Engineering*, vol. 90, p. 106971, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790621000045

[50] S. Kundan and I. Anagnostopoulos, "Priority-aware scheduling under shared-resource contention on chip multicore processors," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[51] O. Spantidi, I. Galanis, and I. Anagnostopoulos, "Frequency-based power efficiency improvement of cnns on heterogeneous iot computing systems," in *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, 2020, pp. 1–6.

[52] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Perf&fair: A progress-aware scheduler

to enhance performance and fairness in smt multicores," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 905–911, 2016.

[53] Y. Song, O. Alavoine, and B. Lin, "A self-aware resource management framework for heterogeneous multicore socs with diverse qos targets," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 2, Apr. 2019. [Online]. Available: https://doi.org/10.1145/3319804

[54] O. Spantidi, I. Anagnostopoulos, and G. Fainekos, "Efficient resource management of clustered multi-processor systems through formal property exploration," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1673–1678.

[55] B. Salami, H. Noori, and M. Naghibzadeh, "Fairness-aware energy efficient scheduling on heterogeneous multi-core processors," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 72–82, 2021.

[56] A. Gamatie, G. Devic, G. Sassatelli, S. Bernabovi, P. Naudin, and M. Chapman, "Towards energy-efficient heterogeneous multicore architectures for edge computing," *IEEE Access*, vol. 7, pp. 49 474–49 491, 2019.

[57] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 4, 2012.

[58] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer, "Cruise: Cache replacement and utility-aware scheduling," in *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. ACM, 2012, pp. 249–260.

[59] Y. Xie and G. H. Loh, "Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 174–183.

[60] S. Srikantaiah, R. Das, A. K. Mishra, C. R. Das, and M. Kandemir, "A case for

integrated processor-cache partitioning in chip multiprocessors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* ACM, 2009, p. 6.

[61] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang, "Mcc-db: minimizing cache conflicts in multi-core processors for databases," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 373–384, 2009.

[62] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques.* IEEE Computer Society, 2004, pp. 111–122.

[63] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory bandwidth management for efficient performance isolation in multi-core platforms," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 562–576, 2016.

[64] D. Xu, C. Wu, P.-C. Yew, J. Li, and Z. Wang, "Providing fairness on shared-memory multiprocessors via process scheduling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, pp. 295–306, 2012.

[65] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *Proceedings of the 39th Annual IEEE/ACM international Symposium on Microarchitecture.* IEEE Computer Society, 2006, pp. 208–222.

[66] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 450–462.

[67] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 2011, pp. 248–259.

[68] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using os observations to

improve performance in multicore systems," *IEEE micro*, vol. 28, no. 3, pp. 54–66, 2008.

[69] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Addressing fairness in smt multicores with a progress-aware scheduler," in *Parallel & Distributed Processing Symposium, 2015 IEEE International*. IEEE, 2015, pp. 187–196.

[70] L. Tang, J. Mars, and M. L. Soffa, "Contentiousness vs. sensitivity: Improving contention aware runtime systems on multicore architectures," in *1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, ser. EXADAPT '11. New York, NY, USA: ACM, 2011, pp. 12–21.

[71] A.-H. Haritatos, K. Nikas, G. Goumas, and N. Koziris, "A resource-centric application classification approach," in *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications*, C. Trinitis and J. Weidendorfer, Eds., Jan 2016, p. 7.

[72] A.-H. Haritatos, G. Goumas, N. Anastopoulos, K. Nikas, K. Kourtis, and N. Koziris, "Lca: A memory link and cache-aware co-scheduling approach for cmps," in *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*. IEEE, 2014, pp. 469–470.

[73] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 657–668.

[74] J. D. McCalpin, "A survey of memory bandwidth and machine balance in current high performance computers," *IEEE TCCA Newsletter*, 1995.

[75] H.-Q. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," 1999.

[76] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[77] S. C. Woo *et al.*, "The SPLASH-2 programs: Characterization and methodological considerations," in *ACM SIGARCH computer architecture news*, vol. 23, no. 2. ACM, 1995, pp. 24–36.

[78] D. Pase, "The pchase benchmark page," 2008.

[79] V. H. W.-J. Kell B., "An mdd approach to multidimensional bin packing," in *Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2013)*, ser. Lecture Notes in Computer Science, S. M. e. Gomes C., Ed. Berlin, Heidelberg: Springer, 2013, vol. 7874, pp. 128–143.

[80] S. Gualandi and M. Lombardi, "A simple and effective decomposition for the multidimensional binpacking constraint," in *Principles and Practice of Constraint Programming (CP 2013)*, ser. Lecture Notes in Computer Science, S. C. (eds). Nelson, Ed. Berlin, Heidelberg: Springer, 2013, vol. 8124, pp. 356–364.

[81] M. M.D., "Multidimensional bin packing revisited," in *Principles and Practice of Constraint Programming (CP 2013)*, ser. Lecture Notes in Computer Science, S. C. (eds). Nelson, Ed. Berlin, Heidelberg: Springer, 2013, vol. 8124, pp. 513–528.

[82] S. H. Mehta D., O'Sullivan B., "Comparing solution methods for the machine reassignment problem," in *Principles and Practice of Constraint Programming (CP 2012)*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, vol. 7514, pp. 782–789.

[83] P. N. M. Berkelaar, K. Eikland, "Cran package 'lpsolve'," 2015.

[84] "Cruise: cache replacement and utility-aware scheduling: Acm sigarch computer architecture news: Vol 40, no 1," https://dl.acm.org/doi/10.1145/2189750.2151003, (Accessed on 09/06/2023).

[85] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 104–117.

[86] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* IEEE, 2004, pp. 111–122.

[87] T. Marinakis and I. Anagnostopoulos, "Performance and fairness improvement on cmps considering bandwidth and cache utilization," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 1–4, 2019.

[88] J. Feliu *et al.*, "Perf&fair: A progress-aware scheduler to enhance performance and fairness in smt multicores," *IEEE Transactions on Computers*, 2017.

[89] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.

[90] "Do the intel® xeon® processors come with cache allocation technology..." https://www.intel.com/content/www/us/en/support/articles/000035900/processors/intel-xeon-processors.html, 06 2020.

[91] "Odroid-xu3," https://www.hardkernel.com/shop/odroid-xu3/.

[92] T. Mück *et al.*, "Run-dmc: Runtime dynamic heterogeneous multicore performance and power estimation for energy efficiency," in *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis.* IEEE Press, 2015.

[93] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 15:1–15:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555729.2555744

[94] J. Feliu *et al.*, "Addressing fairness in smt multicores with a progress-aware scheduler," in *2015 IEEE International Parallel and Distributed Processing*

Symposium. IEEE, 2015, pp. 187–196.

[95] D. Xu *et al.*, “Providing fairness on shared-memory multiprocessors via process scheduling,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 295–306.

[96] M. J. Walker *et al.*, “Accurate and stable run-time power modeling for mobile and embedded cpus,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 1, pp. 106–119, 2016.

[97] “Exynos 5 octa 5422 processor: Specs, features — samsung exynos,” https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5422/, (Accessed on 10/04/2020).

[98] “Smartphones with exynos processors — samsung exynos,” https://www.samsung.com/semiconductor/minisite/exynos/showcase/smartphone/, (Accessed on 10/04/2020).

[99] “big.little – arm,” https://www.arm.com/why-arm/technologies/big-little, (Accessed on 10/04/2020).

[100] “Odroid xu4 onboard processor/dram power sensors - odroid,” https://forum.odroid.com/viewtopic.php?t=18701, (Accessed on 10/04/2020).

[101] B. Donyanavard *et al.*, “Sparta: Runtime task allocation for energy efficient heterogeneous manycores,” in *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS).* IEEE, 2016, pp. 1–10.

VITA

Graduate School
Southern Illinois University Carbondale

Shivam Kundan

shivamkundan@hotmail.com

Southern Illinois University Carbondale
Master of Science, Electrical & Computer Engineering, May 2019

Dissertation Paper Title:
RESOURCE-OPTIMIZED SCHEDULING FOR ENHANCED POWER EFFICIENCY AND THROUGHPUT ON CHIP MULTI-PROCESSOR PLATFORMS

Major Professor: Dr. I. Anagnostopoulos

Publications:

1. Shivam Kundan, Ourania Spantidi, Iraklis Anagnostopoulos, *Online frequency-based performance and power estimation for clustered multi-processor systems*, Computers & Electrical Engineering, Volume 90, 2021, 106971, ISSN 0045-7906, https://doi.org/10.1016/j.compeleceng.2021.106971.

2. S. Kundan and I. Anagnostopoulos, *"Priority-Aware Scheduling under Shared-Resource Contention on Chip Multicore Processors,"* 2021 IEEE International Symposium on Circuits and Systems (ISCAS), Daegu, Korea, 2021, pp. 1-5, doi: 10.1109/ISCAS51556.2021.9401337.

3. Shivam Kundan, Theodoros Marinakis, Iraklis Anagnostopoulos, and Dimitri Kagaris. 2022. *A Pressure-Aware Policy for Contention Minimization on Multicore Systems*. ACM Trans. Archit. Code Optim. 19, 3, Article 40 (September 2022), 26 pages. https://doi.org/10.1145/3524616

4. S. Kundan and I. Anagnostopoulos, *"A Machine Learning Approach for Improving Power Efficiency on Clustered Multi-Processor System"*, 2020 IEEE International Symposium on Circuits and Systems (ISCAS), Seville, Spain, 2020, pp. 1-5, doi: 10.1109/ISCAS45731.2020.9180474.

5. T. Marinakis, S. Kundan and I. Anagnostopoulos, *"Meeting Power Constraints While Mitigating Contention on Clustered Multiprocessor System"*, in IEEE Embedded Systems Letters, vol. 12, no. 3, pp. 99-102, Sept. 2020, doi: 10.1109/LES.2019.2956990.