Exploiting TLB Sharing to Improve Performance in Server CPUs

by

Abdelrahman S. Hussein

B.Sc., Mansoura University, Egypt, 2017

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

Declaration of Committee

Name:	**Abdelrahman S. Hussein**
Degree:	**Master of Science**
Thesis title:	**Exploiting TLB Sharing to Improve Performance in Server CPUs**
Committee:	**Chair:** Tianzheng Wang Assistant Professor, Computing Science
	Alaa Alameldeen Supervisor Associate Professor, Computing Science
	Arrvindh Shriraman Committee Member Associate Professor, Computing Science
	Prashant Nair Examiner Assistant Professor, Electrical and Computer Engineering University of British Columbia

Abstract

Simultaneous Multithreading (SMT) is a fundamental feature that improves the performance of modern CPUs via thread-level parallelism. SMT allows the CPU to issue instructions from different threads on the same core by sharing CPU resources among threads, therefore improving performance when these resources are under-utilized. However, SMT threads suffer from negative inter-thread interference where one thread's resource use degrades the performance of other threads, which affects the Quality of Service (QoS) in server CPUs. In this thesis, we observe that cooperative threads, i.e., threads from the same process that share a virtual address space, interfere positively, where one thread's performance can benefit from another thread sharing CPU resources. To exploit this observation, we propose simple architectural changes that enable effective sharing of the Translation Lookaside Buffer (TLB) by cooperative threads. The Operating System (OS) needs to notify the hardware whether the currently mapped threads are cooperative. Our architecture will take advantage of running cooperative threads by sharing the TLB instead of partitioning it to improve performance.

Keywords: Simultaneous Multithreading; Server CPUs; Multithreading

Dedication

To my beloved, Mariam, and my little angel, Layla,

I would've not gone that far if it had not been for your existence in my life. To my dear partner, wife, and best friend, Mariam. Thanks for having my back during the time of this work. You have been my main source of confidence and faith during the hard times. I owe you this moment and I hope one day I make amends to you. To my sweet baby girl, Layla, thanks for teaching me patience and becoming the reason I, literally, work day and night to finish this work. Your birth was the shining turning point in our life.

I dedicate this work to my Mariam and Layla.

Acknowledgements

I would like to begin by thanking God for enlightening my path and blessing me with enormous support and guidance throughout the time of this work. I am grateful to so many people who always cared about my success.

No words can describe how in debt I am to my advisor and mentor, Prof. Alaa Alameldeen, for his remarkable guidance and advice during this work. As I was newly learning about and doing research in Computer Architecture, he was always there answering my questions and putting me on the right track to be a better researcher. I look forward to more time spent under his supervision during my upcoming Ph.D. to learn more from him regarding work and life. I would like to express my sincere gratitude to my examining committee; Prof. Arrvindh Shriraman and Prof. Prashant Nair, and the defense chair, Prof. Tianzheng Wang, for serving on my thesis defense. I would like to extend my gratitude to our collaborators at the Intel Transformative Server Architecture (TSA) Project at Intel and the University of British Columbia for their feedback on this work. Special thanks to Adithya Ranganathan of Intel Labs for his support and help with this work.

To my parents, thanks for the continuous invaluable support throughout these whole years. I hope this makes you proud parents like you have always made me a proud son. To my sisters Nancy and Deena, thanks for all the fun times and being a source of laughter.

I would like to thank my all-the-time mentors and friends, Ahmed A. Metwally, Abdelrahman Hosny, and Khaled Diab for their valuable advice on how to become a successful researcher on both personal and professional levels. The Computer Architecture Student Association (CASA) is an outstanding community that supports computer architecture junior and senior research students. I would like to extend my thanks to Elba Garza and Tianqi Tang for their mentorship of me as a junior computer architecture researcher. I also would like to thank Mahmoud Khairy and Abdulrahman Mahmoud who helped me in a very early time with my first steps in the field.

Special thanks to my lab mates for making the lab a fun place to work at; Mahmoud AbuMandour, Marzieh Barkhordar, Brian Fu, Zahra Jamarani, Yonas Kelemework, Bishoy Sabry, Fateme Shokouhinia, and Parker Tian. I also would like to thank my friends for their time and support, which did not stop even when we are thousands of miles apart; Sherif Abdelhadi, Mina Magdy, Mohammed Sarg, Mohammed Elmanzlawy, Ahmed Elmessiry, Mohammed Mustapha, and Mohammed Ewis.

Table of Contents

Declaration of Committee **ii**

Abstract **iii**

Dedication **iv**

Acknowledgements **v**

Table of Contents **vi**

List of Tables **viii**

List of Figures **ix**

1 Introduction **1**

2 Background **5**

2.1 Simulatenous Multithreading 5

2.2 Resource Sharing and Contention 6

2.3 Translation Lookaside Buffer 8

2.4 Multithreading Scheduling in OS 9

3 Methodology **11**

3.1 Proposed System 11

3.1.1 Cooperative Threads in OS 11

3.1.2 TLB Sharing Policy 12

3.1.3 Proposed TLB Sharing Mechanism 13

3.2 Experimental Setup 13

3.2.1 The Simulator 13

3.2.2 Benchmarks 13

3.2.3 Challenges 15

3.2.4 Trace Generation 16

3.3 Evaluation Methodology 17

3.3.1 Modifications in SniperSim 17

3.3.2 Architecture Configuration . . . 17
3.3.3 Evaluated Configurations and Performance Metrics . . . 18
3.4 Limitations . . . 19

4 Results and Discussion **20**
4.1 Impact on Performance . . . 20
4.2 TLB Performance . . . 23
4.3 Sensitivity Analysis . . . 27
4.3.1 TLB Size . . . 27
4.3.2 TLB Associativity . . . 29
4.3.3 DTLB MPKI . . . 34
4.3.4 STLB Miss Penalty . . . 34
4.4 Extra Benefit in Caches . . . 35

5 Conclusion and Future Work **39**
5.1 Conclusion . . . 39
5.2 Future Work . . . 40
5.2.1 Other CPU Structures . . . 40
5.2.2 Extending to CMP . . . 40

Bibliography **41**

List of Tables

Table 3.1 Description of the benchmarks used from the DaCapo Benchmark Suite 14
Table 3.2 Description of the benchmarks used from the Renaissance Benchmark Suite 14
Table 3.3 Specifications of the baseline architecture used 18
Table 3.4 The sharing mechanism of resources in the baseline architecture . . . 18

List of Figures

Figure 4.1 Performance gain normalized to performance of the baseline configuration (Non-cooperative on private TLBs) for cooperative threads running on private and shared TLB (The higher the better) 21

Figure 4.2 Absolute values for the STLB MPKI for each benchmark for non-cooperative threads running on baseline architecture and cooperative threads running on the proposed architecture 21

Figure 4.3 Absolute values for the ITLB MPKI for each benchmark for non-cooperative threads running on baseline architecture and cooperative threads running on the proposed architecture 22

Figure 4.4 Absolute values for the DTLB MPKI for each benchmark for non-cooperative threads running on baseline architecture and cooperative threads running on the proposed architecture 23

Figure 4.5 Reduction achieved in the STLB MPKI due to co-locating cooperative threads on the same core with STLB shared (The higher the better) 24

Figure 4.6 Reduction achieved in the ITLB MPKI due to co-locating cooperative threads on the same core with ITLB shared (The higher the better) 24

Figure 4.7 Reduction achieved in the DTLB MPKI due to co-locating cooperative threads on the same core with DTLB shared (The higher the better) 25

Figure 4.8 The percentage of shared data page addresses among the threads of each benchmark normalized to all the pages accessed 26

Figure 4.9 The percentage of shared instruction page addresses among the threads of each benchmark normalized to all the pages accessed 26

Figure 4.10 Performance gain when cooperative threads run on **shared** TLBs where the size of the ITLB varies, normalized to the configuration with the smallest parameter value (The higher the better) 27

Figure 4.11 Performance gain when cooperative threads run on **private** TLBs where the size of the ITLB varies, normalized to the configuration with the smallest parameter value (The higher the better) 28

Figure 4.12 Performance gain when cooperative threads run on **shared** TLBs where the size of the DTLB varies, normalized to the configuration with the smallest parameter value (The higher the better) 28

Figure 4.13 Performance gain when cooperative threads run on **private** TLBs where the size of the DTLB varies, normalized to the configuration with the smallest parameter value (The higher the better) 29

Figure 4.14 Performance gain when cooperative threads run on **shared** TLBs where the size of the STLB varies, normalized to the configuration with the smallest parameter value (The higher the better) 30

Figure 4.15 Performance gain when cooperative threads run on **private** TLBs where the size of the STLB varies, normalized to the configuration with the smallest parameter value (The higher the better) 30

Figure 4.16 Performance gain when cooperative threads run on **shared** TLBs where the associativity of the ITLB varies, normalized to the configuration with the smallest parameter value (The higher the better) . 31

Figure 4.17 Performance gain when cooperative threads run on **private** TLBs where the associativity of the ITLB varies, normalized to the configuration with the smallest parameter value (The higher the better) . 31

Figure 4.18 Performance gain when cooperative threads run on **shared** TLBs where the associativity of the DTLB varies, normalized to the configuration with the smallest parameter value (The higher the better) 32

Figure 4.19 Performance gain when cooperative threads run on **private** TLBs where the associativity of the DTLB varies, normalized to the configuration with the smallest parameter value (The higher the better) 32

Figure 4.20 Performance gain when cooperative threads run on **shared** TLBs where the associativity of the STLB varies, normalized to the configuration with the smallest parameter value (The higher the better) 33

Figure 4.21 Performance gain when cooperative threads run on **private** TLBs where the associativity of the STLB varies, normalized to the configuration with the smallest parameter value (The higher the better) 33

Figure 4.22 Reduction in the DTLB MPKI when cooperative threads run on **shared** TLBs where the associativity of the DTLB varies, normalized to the configuration with the smallest parameter value (The higher the better) 34

Figure 4.23 Performance gain when cooperative threads run on **shared** TLBs where the miss penalty of the STLB varies, normalized to the configuration with the smallest parameter value (The higher the better) 35

Figure 4.24 Reduction achieved in the L1-D MPKI due to co-locating cooperative threads on the same core (The higher the better) 36

Figure 4.25 Reduction achieved in the L1-I MPKI due to co-locating cooperative threads on the same core (The higher the better) 36
Figure 4.26 Reduction achieved in the L2 MPKI due to co-locating cooperative threads on the same core (The higher the better) 37
Figure 4.27 Reduction achieved in the L3 MPKI due to co-locating cooperative threads on the same core (The higher the better) 37

Chapter 1

Introduction

Simultaneous Multithreading (SMT) is an important microarchitectural mechanism used to increase the efficiency of modern CPUs. Because it allows multiple threads to stream instructions to the pipeline in parallel, it increases the thread-level parallelism (TLP) that leads to filling the utilization gaps found in Out-of-Order superscalar processors [1]. SMT offers an efficient scheme to share the CPU resources among multiple threads at the same time, which represents a cheaper alternative to duplicating the number of cores like the Chip Multiprocessor (CMP) [2, 3]. Fundamentally, the SMT aims at logically duplicating the number of cores visible to the software at the cost of a minor increase in area and energy consumption. To maximize the efficiency of shared resources, threads assigned to the same physical core may use different sharing policies that reduce the interference effect or support security [4]. While SMT enables CPUs to execute instructions simultaneously from different threads, it also exhibits a negative side-effect of creating contention on the shared resources, such as caches, Reorder Buffer (ROB), functional units, etc., where the co-located threads compete for resources to the extent their overall performance degrades considerably.

SMT is enabled by major cloud services providers, such as Google Cloud, Amazon AWS, and Microsoft Azure [5]. Whether the workload running on a given server CPU is batch-oriented or latency-sensitive, companies like Google co-locate workloads of both types on SMT cores on the same physical core [6] to achieve better utilization. However, due to the aforementioned contention effect caused by resource sharing, the performance loss caused by SMT becomes more significant from a Quality-of-Service (QoS) perspective [7, 8].

Prior studies [9, 10, 11] highlighted that, while server workloads are heavily multi-threaded, they still exhibit under-utilization of CPU resources for many reasons, such as the low memory level parallelism and inefficient front-end (e.g., high instruction cache miss rate). Therefore, this is translated to a higher total cost of ownership (TCO) that companies afford due to this low efficiency. Given the previous reasons, there has become a significant need for more SMT threads to be added to the core to exploit the under-utilized resources. Yet, this creates a trade-off since adding more SMT cores promises better utilization, but also creates capacity issues due to the shared resources and the subsequent consequences of creating contention [12]. Thus, this option becomes even more infeasible because of the implicit design complexity and the resulting impact of growing the contention and stressing the caches among the threads.

To that end, there have been many proposals to optimize the co-location of threads from different perspectives, such as microarchitectural schemes for sharing resources and the co-location scheduling policy. While the main target is improving the workload performance, these proposals adopted different strategies at their core. Many researchers have proposed many mechanisms that make use of the fetch policy to achieve maximum throughput of the overall system but ignore the performance of individual threads, such as the work in [13, 14]. Another approach is to compromise the fairness of resource sharing among threads to optimize for the QoS of one critical thread by leveraging the unused resource of the other threads [6].

On the other hand, many efforts addressed the scheduling challenges of multithreaded workloads to alleviate the execution bottlenecks of the threads due to synchronization and communication. One of the early proposals on this topic was done by Snavely et. al [15] which initially co-locates threads and then adjusts the scheduling based on information collected from performance counters. Another work by Tam et. al. [16] took a similar approach where it implemented an OS-based mechanism that schedules threads based on performance data sampled using Performance Monitoring Units (PMUs) in the processing units. However, this later proposal was extended to CMPs. A more recent work [17] proposed a cooperative HW/SW mechanism to recognize the bottleneck regions in threads and then execute these bottlenecks on fast cores to alleviate their impact on performance. These proposals are discussed in-depth in Chapter 2.

While the aforementioned work promises significant performance gains, it does not take into consideration the relation among threads in terms of instruction and data sharing due to sharing the address space. This is a vital performance component that should have a profound impact on the performance of the hardware shared resources. Furthermore, while it remains intuitive that co-locating threads that share pages on the same core should result in overall better performance, the hardware remains oblivious to the relation among the threads that currently execute. This should be marked as a wasted window for optimization at a low cost since such information opens the space for more efficient mechanisms. An additional drawback can be observed in the context of the evaluation methodologies that exist in the literature. For the server CPUs, the challenge of evaluating prior mechanisms becomes more challenging due to the sophisticated components of the server workloads that require a range of features to be supported by the simulator used. The prior work compromises either the Instruction-Set Architecture (ISA) in use, the workloads, or the capacity of the research community to reproduce the results by using in-house simulators.

On the contrary to the previous work that took a fine-grained approach to analyze and identify execution patterns of the running threads, this thesis adopts a more coarse-grained approach to optimize for the co-location of threads based on *cooperative threads*. Cooperative threads are threads that are spawned from the same parent process and share the same virtual address space. Given this fact, this nature can be used to make threads interfere positively, instead of competing for shared resources. This can lead to a scenario where one thread would be prefetching shared data for the other thread and vice-versa if they are co-located on the same core. Unlike the previous proposals, this approach requires minor changes in the OS scheduler to favor co-locating cooperative threads on the same physical core. It is the responsibility of the OS then to pass the information to the hardware that the threads to be scheduled are cooperative. This information is to be used by the hardware to decide whether to share the microarchitectural resources in the presence of cooperative threads or to statically partition the resource when the threads are non-cooperative.

We apply this proposal to the Translation Lookaside Buffer (TLB) to study how much it benefits from this mechanism and analyze its impact on the overall performance of the workload.

The role of the TLB is critical in terms of reducing the cost per single memory access either for instruction or data. This is because it caches the virtual address translation to the physical address, eliminating the need to access the memory multiple times for page walking to access multilevel page tables. Accordingly, this saves hundreds of cycles that would have been spent in traversing the page table. For two or more cooperative threads that share the same virtual address space, the same virtual address in those threads is mapped to the same physical address in memory. In light of this scenario, assuming a statically partitioned TLB, the entry for a given virtual address that is accessed by several cooperative threads is replicated in the TLB, which reduces the utilization of the TLB. This thesis investigates the hypothesis that, if the threads are cooperative and the TLB is redesigned to a shared TLB, this should give the space for more entries to be cached in the TLB which results in fewer page walks and reduces execution time.

To increase the TLB utilization, we employ our observation that co-locating cooperative threads should be translated into positive interference that threads would *help* each other. This is aggressively needed especially in the case of server applications where workloads have a considerably large code footprint, which creates extra contention on the resources that eventually degrades the performance. To address this issue, we base the evaluation of our work on server workloads. More specifically, we make the following contribution:

- We develop a reliable evaluation environment for server CPU simulation and evaluation. This environment can be easily used with server workloads without facing complex setup challenges.

- We show that when the OS scheduler is modified to co-locate cooperative threads on the same core, this would result in a performance gain of 10% on average. The reasons behind this improvement are analyzed.
- We incorporate an adaptive sharing mechanism of the TLB, such that the OS informs the underlying CPU architecture that the scheduled threads are cooperative. When this approach is applied besides the co-location of cooperative threads, this results in an extra 5% performance gain, resulting in a total 15% average performance gain.
- We study the sensitivity of the proposed system to the variation in the TLB parameters and show that our system can scale with larger TLBs.
- Furthermore, we quantify the sharing among the cooperative threads and highlight that the more the sharing quantity between the threads is, the more potential performance gain is expected to be achieved.

The rest of this thesis is organized as follows: Chapter 2 gives the background and discusses the prior research related to this work. Chapter 3 explains the proposed solution, the evaluation methodology and its related challenges, and the evaluation setup. Chapter 4 reports the results of this work and discuss their aspects. Chapter 5 concludes the work and highlights the future work for this study.

Chapter 2

Background

2.1 Simulatenous Multithreading

Simultaneous Multithreading [2, 3] is a computer architecture scheme that enables multiple hardware threads/contexts to stream their instructions at any cycle to the pipeline simultaneously, thus, these threads are executed in parallel. Fundamentally, SMT addresses the issue of resource under-utilization in Out-of-Order (OoO) superscalar processors, which happens if the currently executing context goes through pipeline stalls due to resource inefficiency, such as cache misses. In OoO superscalar processors, this phenomenon results in wasting the CPU resources horizontally or vertically. While the former happens due to the dependencies between instructions which leads to partial leveraging of the CPU issue bandwidth, the latter is experienced due to long latency events that block the entire execution, such as caches/TLB misses or page faults.

Unlike Coarse-Grain or Fine-Grain multithreading, the SMT Supports hardware contexts. This means that, unlike software context switching which requires hundreds of CPU cycles to store the state of the thread, it now takes much fewer cycles to maintain the status of the thread to be replaced.

While many proposals in the 90s [2, 3, 18, 19, 20] contributed to the features of the SMT we know today, it took a few years until 1999 when Digital Alpha announced the first commercial implementation. Still, the project was abandoned incomplete until Intel delivered the actual first implementation [21, 22] in 2002, represented in their Pentium 4 Xeon server processors. A few years later, Intel forwarded its attention to multicore processors to motivate simpler core design [1] at the cost of sacrificing work on SMT. Yet, in 2010, Intel was able to combine both SMT and CMP in Nehalem [23].

Before Tullsen et. al.'s work [2, 3] on SMT, Hirata et. al. [24] evaluated their architecture, where instructions are streamed from different threads to functional units to improve throughput, using ray-tracing workloads. However, they based their evaluation on many unrealistic specifications, such as excluding caches, TLBs, and branch predictors.

2.2 Resource Sharing and Contention

The basic idea of SMT is to share on-core resources among threads to improve utilization. Conventionally, in CMP, all resources are exclusively allocated to one thread at a time, except for the Last Level Cache (LLC). While this approach reduces the contention among threads and its control logic per core is easier to design, these resources are not fully utilized during the thread execution. Because this work discusses the proposal to share the TLB among concurrent threads, we discuss the literature on cache sharing and partitioning.

Early attempts by [25, 26, 27] discussed partitioning the issue bandwidth among the executing threads where only one thread issues instruction in a given cycle, such that each thread streams one instruction per cycle. However, this leads to the previously mentioned utilization problem since the resources are now exclusively allocated to the single thread running during this cycle.

Caches are one of the critical components that deeply affect the performance and it has been an open research problem on how to share/partition the cache amongst interleaving threads. One of the early works by Stone et. al. [28] looked into how to statically partition the cache memory by assuming the information on cache misses for one process is available to the other process(s), which can be hard to obtain as the application pattern changes depending on its input set. Since logical cores in SMT setup share resources, the work by Tullsen. et. al. [2] analyzed how sharing or partitioning the caches affects the performance. Their results highlighted that sharing both instruction and data cache achieves better performance compared to making either or both of them private. This conclusion was also supported by another work [29] that concluded that cache contention is not a problem as their work assumed a range of workloads can fit in the L2 cache of which size was 256 KB. However, workloads have grown much more memory-demanding, which requires re-consideration of the contention effect in caches.

This motivated the work by Suh et. al. [30, 31] to propose a dynamic partitioning mechanism for the cache to distribute it among threads that execute interleaving. Their mechanism used performance counters to measure the cache misses per thread and then re-distribute the cache space among the threads using the data sampled from these counters. Nevertheless, as the applications have evolved in complexity and heterogeneity, it is more demanding to study the impact of running applications ranging from single-threaded to virtual machines concurrently on CMP or SMT. An example of this is the work in [32] that introduced the concept of QoS in caches to overcome the variability of locality and latency sensitivity due to the different memory access patterns exhibited by each application. This paper proposed the procedure adopted in their framework (i.e., CQoS) to classify, assign, and enforce priority in cache management among threads. Besides, it also proposed the mechanisms to impose the priority, which include selective cache allocation, heterogeneous cache regions, and static/dynamic set partitioning.

Subsequent work by Qureshi and Patt [33] proposed a more effective cache partitioning mechanism, namely the utility-based cache partitioning (UCP) among applications that execute simultaneously, depending on the likelihood of cache misses a process would encounter for a defined cache resource allocated. Their Utility Monitoring (UMON) observes the performance of each running process and then passes the information collected to the actual partitioning algorithm. The UMON is based on the Dynamic Set Sampling concept [34] that is used to foresee the cache performance by sampling a few sets of the cache.

Another work by Chang and Sohi [35] proposed Cooperative Cache Partitioning (CCP) on CMPs which aimed at solving the cache contention problem by simply using time-sharing partitioning, which would increase the performance of thrashing threads.

Overall, there have been many proposals that altered the insertion and replacement policies for better cache sharing, as in [36, 37, 38, 39, 40]. In particular, the proposals in [37] and [38] use probabilistic mechanisms to partition the caches. Another work by El-sayed et. al. [41] proposed the Kpart, which samples information on the cache partitions of the co-located programs and then partitions the cache among clusters of processes/threads.

Other CPU structures have also been the focus of many proposals for efficiently sharing them among co-running threads. Recent work by Margaritob et. al. [6] proposed a mechanism for more efficient Reorder Buffer (ROB) sharing among threads on the CPU level to achieve better resource utilization among threads. Another work proposed DCRA [42], which proposed a dynamic sharing policy of the issue queue and register file entries by tracking how each thread uses these resources. Reorder Buffer was the focus of the study by Sharkey et. al. [43] that also proposed adaptive partitioning of the ROB. A third proposal by Choi et. al. [44] also discussed another dynamic partitioning mechanism by first learning about the most efficient resource via a hill-climbing-based framework. This can even be taken to a very complex level implementation like the proposal discussed in [45]. In this study, a highly configurable system was proposed where the core resources are sliced and then used to form many small cores on demand.

As this thesis is more concerned with server CPUs as their workload has different characteristics due to their large code footprint in the memory, it is important to study the architecture in the presence of server applications. A very early study by Ruan et. al. [46] evaluated how server workloads behave when run on SMT cores. Their study concluded that, by having SMT enabled, the DTLB miss rate increases which can be justified by the fact that each thread is accessing scattered regions in memory. Also, the branch misprediction rate increased by 50%.

Therefore, the resources sharing policy remains a challenging problem in the multi-threaded and multi-core architecture. It remains a crucial goal to find the balance between containing the contention effect and increasing the sharing degree such that the threads help each other.

2.3 Translation Lookaside Buffer

The Translation Lookaside Buffer (TLB) [47] is the on-core cache that stores the translation of the logical/virtual address to its associated physical address in the page-based virtual memory system. TLBs are a crucial component in modern processors because they are capable of saving hundreds of cycles that could've been consumed in accessing the OS page tables in which the virtual address translation is found. This is more valuable in multi-level paging, where N number of memory accesses are required for N-level memory pages. The number of these memory accesses can even grow to 24 accesses in virtualized setups, which represents, on average, more than 40% of the runtime [48]. The Memory Management Unit (MMU) in the processor uses the TLB to store the regularly used instruction and data virtual addresses, which obviates the need to access the memory multiple times per address translation. TLBs are accessed with the virtual address, TLBs entries are either tagged with core/thread ID, such as the shared data TLB (DTLB) in Intel processors, or each core in the CMP has its own data/instruction TLB [4, 21, 49].

Several works introduced more optimized TLB mechanisms that help to mitigate the high miss penalty of a TLB miss by either reducing the misses or hiding them. For example, Bhattacharjee et al. [50] introduced one of the early proposals of TLB prefetching in the multi-core setup where their proposal consisted of two TLB prefetchers. The first prefetcher takes advantage of the commonality in virtual page address between cores such that one core inserts the entries into other cores, whereas the second relies on the distance-predictable TLB misses among the cores. Another proposal [51] considered TLB prefetching as an expensive approach since each prefetch request/transaction requires multiple memory access to traverse, especially if the prefetcher has a lower accuracy. Based on that, their proposal exploited the locality in the last level of page tables by predicting which of their entries can be useful in the future and combining it with state-of-the-art prefetchers.

Other efforts focused on reducing the miss latency of a TLB miss. One approach is to deploy a large L3 TLB cache to be located in the system's main memory such that it requires only one memory access [52]. A more direct approach is to enhance the performance of the caches inside the MMU either on the software level or hardware level [53, 54]. More proposals [55, 48] adopted the hashed page tables, instead of the conventional radix page tables, which handle TLB misses faster and more efficiently. Seeking further optimization in the TLB performance, Bhattacharjee et. al. [56] proposed to redesign the STLB to be shared among cores in the CMP setup. While this work demonstrated promising results for both multi-programmed and multithreaded workloads, it did not consider the large-code footprint workloads, as server applications, that are the focus of this thesis.

Another mechanism to hide the latency in TLBs is by speculating the translation of a given virtual address when this virtual address has no entry in the TLB (i..e, TLB miss). Several proposals, such as [57, 58, 59], adopted this theme where, on a TLB miss, the CPU runs in a speculative mode that starts by predicting the translation. Concurrently, the page table is traversed until the associated physical address is found, which if turned to match the speculated address, the CPU would commit the work that has been done.

An additional challenge in the virtual memory system is the TLB shootdown. The TLB shootdown happens when one or more TLB page entries get altered in cases like changing permissions or changing the address translation. TLB shootdown is an expensive operations that happen in many contexts, such as memory management debugging [60] and concurrent garbage collection [61], and may take as long as 13.2 μs [62]. Many recent studies [63, 64, 65, 66, 67, 68, 69] considered further optimizations to alleviate the huge cost of the TLB shootdown. The main goal of these proposals is to optimize the TLB shootdown such that it happens faster so that the overall shootdown frequency increases [65]. One example is the UNITD proposal [68], which suggested that the TLBs should be included in the conventional cache coherence protocol which leads to a significant reduction in the shootdown overhead.

From the perspective of security, TLBs can be exploited to leak data, in the presence of state-of-the-art cache defenses. An example of this is the proposal of TLBleed [70], which used a machine learning-based approach to analyze the timing properties of memory accesses done by the victim. Another work by Wang et. al. [71] concluded that the TLBs can be exploited against the Intel Software Guard Extension (SGX). This can be done by enforcing the flushing of the TLB to invoke page walks which leads to not trapping the SGX enclaves.

2.4 Multithreading Scheduling in OS

Kumar et. al [72, 73] stated that the presence of heterogeneous multicore processors opens the horizon for more efficient processing. However, this is contingent on the fact that the OS scheduler can map each process/thread to the core that is the best fit for the thread's needs. One direct way to achieve this is by evaluating the Instruction-per-cycle (IPC) metric for each thread, such that it can be used by the scheduler to map the thread to the core that achieves the best IPC. However, despite being a convenient metric for single-threaded workloads, IPC is an unreliable metric for multithreaded workloads [74]. Furthermore, using IPC as a guide to assess the possible mappings consumes a significantly longer time, not to mention that the IPC is also affected by resource sharing and phase change [75].

Because the default Linux Completely Fair Scheduler (CFS) scheduler is not aware of the core capabilities, it only attempts to keep all the cores as utilized as possible and also balanced as per the number of tasks assigned to each core and not to move threads between cores as much as possible [76, 77]. Given the aforementioned drawbacks of using the IPC as a guiding metric for scheduling, many proposals suggested the usage of other metrics to optimize the OS scheduler.

For example, Saez et. al. [78] used ILP and miss rate of the LLC. Another work by Craeynest et. al. [79] established their model on CPI stack, ILP, and memory level parallelism (MLP). A third work [77] proposed another model to identify the performance counters that relate mostly to the performance of the thread(s) and then generate a linear model out of them that the scheduler can use to efficiently map the threads to the cores.

To facilitate the OS to schedule multithreaded programs efficiently on the CPU, many proposals looked at the challenge of identifying the bottlenecks and critical threads to make the system aware of these bottlenecks that can alter the scheduling for better efficiency. For example, one study [80] suggested accelerating the execution of the bottlenecks in code sections, such that the thread spends a shorter time processing shared data and keeping shared data in big core caches. This idea was later adopted by Joao et. al. [17, 81] in their proposed cooperative hardware-software mechanism to identify the bottlenecks in a code segment and then send the threads that cause the highest latency to the large cores.

In the modern Asymmetric Chip Multi-Processor (ACMP), the fairness of the scheduling also remains another challenge, given the different natures of the cores on the ACMP. By default, fair schedulers optimize their scheduling per the fairness of the processing time given to each thread, process, or process group [75]. This, however, can be optimized if the scheduler considers the core capabilities as proposed in [82], where the amount of tasks/load assigned to each core should take into consideration the processing power of the core. Another work [83] proposed equal progress scheduling, in which an estimate of the processing time to be provided by each core has been identified. The threads then are prioritized and scheduled so that they make the same progress. Colab scheduling [75], on the other hand, is more concerned with multithreaded multi-programmed applications, which considers a range of factors to collaboratively define all the factors that are directly consequential on the AMP scheduling, core affinity, and fairness of the scheduling.

Chapter 3

Methodology

3.1 Proposed System

This work looks closely into two aspects of the system; the OS scheduler, and the sharing techniques of the TLBs. We build our proposal on exploiting the cooperative threads of server workloads that run on the CPU. The following subsections demonstrate the proposed system.

3.1.1 Cooperative Threads in OS

Cooperative threads are the threads that belong to the same parent process, thus, they have the same virtual address space. Accordingly, from the perspective of memory access patterns, these threads most likely have very similar access patterns. Therefore, each thread can implicitly participate in hiding latency causes that another thread would encounter during its execution by prefetching the data blocks into the shared resources, such as the caches.

In the server workloads, nevertheless, many services employ multithreading and/or multiprocessing to handle the incoming requests from clients by spawning threads/sub-processes that parallelize the processing of each request, balance the load, etc. Because the nature of requests differs from one request to another, these threads have a larger code footprint, especially if their complex features are considered. However, these threads will still access the same memory pages, which means these threads are now cooperating.

Hence, assigning cooperative threads on different cores, even on the same die, can be translated into a wasted window of opportunity for extra hiding of latency causes. Interestingly, the processor is oblivious to the relationship between the threads allocated in terms of being cooperative or non-cooperative. This represents the basis of the first part of this thesis, which is to allow the OS to communicate such information to the processor.

Normally, when a program needs to spawn a new child thread, it invokes the syscall *clone* to create a thread and then this thread is scheduled for execution, following the OS scheduling scheme. While there have been many proposals in the literature that discuss the optimal approach to schedule diverse threads based on different parameters, our proposal adopts a simple technique that is to co-schedule cooperative threads on the logical cores of the same physical core.

In sync, the OS will inform the proposed architecture whether the scheduled set of threads is cooperative or not. This information can be used by the architecture to determine the best sharing/partitioning policy to be applied. This can be achieved by enabling the OS to write to a hardware control register. In the future work, we further discuss how this idea can be also extended to the CMPs.

As discussed in the following chapter, our evaluation shows that this mechanism achieves significant performance gain. It can also be conjoined with the existing multithreading scheduling schemes for extra benefit.

3.1.2 TLB Sharing Policy

As discussed in chapter 2, many of the existing resources in the x86 ISA are partitioned (i.e. each core/thread has a private chunk of the resource allocated). This design choice aims at achieving a set of goals, such as reducing negative interference in the given resource and maintaining security between threads. The TLB is no exception to this as it is statically partitioned or, at least, core-ID tagged. This is because the TLB is accessed by the virtual address, which means the address used is not unique across programs.

The TLB is split among the cores equally, as in the instruction TLB (ITLB), or shared but tagged with the core ID, as in the data TLB (DTLB). Assuming cooperative threads running on the SMT cores of the same physical core, this will likely lead to replicating entries in the TLB caches, such that each replica belongs to one core. This can be seen as a waste of resources because (1) it wastes precious storage that could've contained extra unique entries that one or more threads need and (2) it increases the TLB misses significantly because these replicas will lead to more conflict misses when other virtual addresses that have not been cached are accessed.

This work takes advantage of this observation and proposes a new architecture in which the TLB is redesigned to be shared whenever it is used by cooperative threads. This requires minimal modifications to the TLB structures such that it handles the virtual address to be translated correctly in each case. Our mechanism is enabled by the information that the OS shares with the architecture about whether the threads are cooperative or not. This should lead the cooperative threads in the server CPUs, when a heavily multithreaded workload is executed that has a large code footprint, to help each other. Consequentially, this should map into performance gain because threads will have more shared entries in the TLB, and threads will act as prefetchers for each other to fetch entries into the TLB for the other corresponding threads.

3.1.3 Proposed TLB Sharing Mechanism

Because TLBs are either split per core, as in the ITLB on Intel cores, or shared but the entries are tagged with the thread/core ID, as in the DTLB also on Intel cores, the proposed sharing mechanism requires simple modifications to the existing TLB implementations.

When the OS identifies cooperative threads, it schedules the cooperative threads on the SMT cores of the same physical core. The CPU is notified that these threads are cooperative through a control register whose bytes are set by the OS in the presence of cooperative threads. When these threads begin executing, the core uses this control register to dynamically decide to share the TLB among the threads. The sharing itself can be achieved by altering how the padding of the virtual address used to access the TLB happens. For example, in the case of the DTLB, when it is shared among the threads, padding each entry with the core/thread ID is ceased. When this happens, more TLB sets will become available for the threads. In the presence of non-cooperative threads, the TLBs will remain private per thread.

3.2 Experimental Setup

3.2.1 The Simulator

To evaluate this work, we model the proposed system architecture in the Sniper Simulator v8.0 [84]. SniperSim is a trace-driven multicore simulator that supports SMT. By default, Sniper uses Intel Pin Instrumentation Tool [85] as the front-end functional simulator, which is responsible for streaming the instructions of the workload on the fly to the back-end of the simulator (i.e., timing model) to evaluate the performance of the proposed architecture.

Because of that, Sniper is a user-space simulator that does not take into consideration the interaction between the workload and the OS that is represented in invoking different OS syscalls, which means that the syscalls are by default emulated and not taken into consideration in the final performance evaluation. Moreover, this means that Sniper limitations are strictly defined by the limitation of its front-end component (i.e., the Intel Pin tool), such that if Pin is incapable of instrumenting a given workload, Sniper cannot be used in that case.

3.2.2 Benchmarks

Because this thesis mainly targets server CPUs that are likely to run workloads with many threads and different memory access patterns, it is essential to carefully choose representative benchmarks that model this behavior. Server workloads are commonly developed in Java, such as Cassandra [86], Kafka [87], and H2 Database Engine [88]. This means that these server workloads have complex execution pattern that includes the usage of virtual machines, such as Java Virtual Machine (JVM), or container services, such as Docker. For such cause, many benchmark suites can mimic this complexity, such as CloudSuite [9], DaCapo [89], and Renaissance [90]. Because CloudSuite depends heavily on Docker containers to model the server/client model, this makes using it for simulation purposes more demanding and complex to achieve because the simulator used has to be equipped with many features to be able to run such a benchmark suite inside Docker containers.

Benchmark	**Description**
Avrora	AVR microcontroller simulation.
Batik	Scalable Vector Graphics (SVG) images generation based on Apache Batik.
Cassandra	The famous No-SQL database management system [86].
Eclipse	Non-gui performance tests execution for the Eclipse IDE.
Graphchi	Disk-based computation of large graphs with billions of edges [91].
h2	JDBCbench-like in-memory benchmark executing banking application transactions.
Kafka	high-throughput low-latency server for realtime data feeds [87].
pmd	Analyze Java classes to determine source code problems.
Spring	Executes PetClinic application on Spring microservice using h2 in-memory database.
Sunflow	Ray-tracing-based image rendering.
Xalan	XML-HTML file transformer.
Tomcat	Execute queries against the Tomcat server and then verify the retrieved results.
Tradebeans/Tradesoap	Day-trader benchmarking through Java Beans/SOAP with Apache GERONIMO backend and h2 DBMS.

Table 3.1: Description of the benchmarks used from the DaCapo Benchmark Suite

Benchmark	**Description**
db-shootout	Testing in-memory Java databases using concurrent shootouts.
finagle-chirper	Microblogging service simulation based on Twitter Finagle [92].
finagle-http	High server load simulation using Twitter Finagle [92] and Netty [93].
neo4j-analytics	Runs Neo4j graph analytical queries on a movie database.

Table 3.2: Description of the benchmarks used from the Renaissance Benchmark Suite

Therefore, we evaluate our work using benchmarks from the DaCapo and Renaissance suites. While our main focus is on explicit server workloads, we also include benchmarks from other domains, such as ray tracing, document transformation, Integrated Development Environment (IDE), and graph databases. Tables 3.1 and 3.2 provide more details on the benchmarks used from DaCapo and Renaissance suites, respectively.

3.2.3 Challenges

Server benchmarks are known to be complex to set up. This makes the majority of academic simulation tools incapable of running these workloads without major compromises unless executed on real hardware, which is infeasible in architecture research. There are many factors that contribute to this complexity. First, these workloads intensely use OS syscalls at a very high rate, unlike HPC applications. This is to achieve many tasks that can only be accessed in the privileged mode, such as disk I/O, networking, polling, etc. Furthermore, because they require an environment as complex as the JVM, this means that these workloads will exhibit a combination of forks and joins to spawn new processes and threads until the actual server is up and ready to receive requests. Hence, this requires full-system simulation because the kernel in this case is a fundamental component to be part of the simulation. Thirdly, for evaluation, a realistic ISA (e.g., x86, ARM, etc.) has to be used for accurate assessment because x86, for example, has an extra layer of complexity since it uses variable-length instructions architecture. The simulator should also support multi-core SMT simulation because this is what server CPUs are like today.

Whereas these requirements represent the foundation to evaluate our work, none of the available simulators support all of these requirements. For gem5 [94], despite supporting full-system simulation for x86, it does not support SMT in this mode due to the sophisticated interaction between the OS and the underlying hardware to enable SMT. Another Simulator is SimFlex [95] that supports OoO execution on SMT, however, it only supports this for SPARC ISA, which is not the focus of this thesis. SniperSim, on the other hand, supports multi-core simulation for x86 ISA and it also models the SMT architecture. Yet, because Sniper mainly depends on Pin, this forces critical limitations, such as the incapability to instrument and, hence, simulate syscalls, and the complexity of instrumenting such a complex environment like the JVM. Furthermore, with a large sequence of spawning sub-processes that become parents of extra threads that terminate at different points of the runtime, Pin cannot trace this tree of forks and clones. It is also worth mentioning that Sniper's timing simulator does not support self-modifying code which is used extensively in the JVM. Instead, it is partially supported in the Pin front-end component.

Therefore, we argue that no standalone simulator satisfies the requirements for evaluating this work. As discussed in chapter 2, prior work compromised at least one of the critical aspects of server CPU simulation. Instead of using the Just in Time (JIT) instrumentation by Pin tool to stream instructions to Sniper's timing model, we choose to generate the CPU traces of the benchmarks on a separate simulator and then feed the traces to Sniper.

3.2.4 Trace Generation

We built a compound flow that consists of two simulators; gem5 and SniperSim, to evaluate the proposed architecture. The gem5 simulator is used to generate the traces and then the traces are then exported to Sniper to evaluate the system under study.

For gem5, we use x86 full-system simulation to run the workloads. This is important because the workloads use the OS syscalls excessively, hence, the trace generation setup must be able to satisfy this requirement. This simulation runs on a dual-core configuration of the Atomic Simple CPU model for faster tracing and also for ease of implementation. Each benchmark is checkpointed after its initialization phase when the server is up, the dataset is created, and the client is ready to send requests. This represents the region of interest (RoI) of the benchmarks that should be traced. Other techniques, such as Simpoint [96, 97] and LoopPoint [98] are used to find the representative region of interest of a given workload. However, for Simpoint, it is limited to single-threaded workloads. Regarding LoopPoint, it assumes the availability of pinballs of the desired workloads, which implicitly assumes that the Pin tool is capable of instrumenting the workload, which is infeasible in this case.

For DaCapo, we use the large input data to stress the architecture, whereas Renaissance does not use the notion of data size, instead, it uses the number of iterations. Renaissance is configured to run into 2 iterations. We generate traces for 1 billion instructions per core (approximately 2 billion instructions aggregated across cores). While this approach may initially seem similar to the work done in [99, 100], our approach differs in two perspectives. First, we generate the entire traces of each benchmark and not just the memory traces. Moreover, we generate multiple trace files, such that each trace file represents one of the threads/cores.

We implement a writer agent in gem5 that collects the information of each instruction, with the help of Intel X86 Encoder Decoder (Intel XED) [101]. For trace dumping, the agent monitors the resetting of the Page Table Base Register (i.e., CR3 register in x86) as an indicator of context switching. Upon the recognition of the context switching, the instructions are then forwarded to a newer trace file that is associated with the new thread. Furthermore, gem5 also generates the address mapping of each thread (and trace file, accordingly) to be used to identify cooperative threads. Moreover, gem5 is also modified to generate a list of memory addresses that represent operands for store instructions to be used within the support of the self-modifying code. Lastly, to overcome the problem of scattered trace files due to context switching for interrupt handling, we developed a standalone tool to merge the trace files that run on the same core and belong to the same address space. For the actual trace writing, we import the writer agent from SniperSim into gem5 with extra modifications to support the self-modifying code.

3.3 Evaluation Methodology

This section discusses the actual procedure followed to evaluate the proposed architecture. We explain how SniperSim is modified to evaluate this work.

3.3.1 Modifications in SniperSim

We extend the support for cooperative threads in SniperSim by adding several knobs that control different aspects of the simulation. First, when Sniper is fed trace files, it assumes multi-programmed execution, thus each thread has different address translation mapping. This is modified to support cooperative threads, such that identical virtual addresses are mapped to identical physical addresses if the trace files are tagged to be cooperative. For Sniper to be able to distinguish cooperative from non-cooperative threads, a knob is implemented to send such information to Sniper. Furthermore, Sniper is fed the operand effective memory addresses of the store instructions at the beginning of the simulation to handle self-modifying code. Thus, Sniper handles this instruction correctly if an instruction address is found in this list (i.e., this instruction has been modified by an earlier store)

This work studies the effect of sharing the TLBs among threads. Thus, the TLBs are also modified to be either shared among cores/threads, or private for each thread. This is also controlled by a separate knob. Additional performance counters are also implemented to measure the performance of the modified TLB.

On the other hand, Sniper does not model the STLB access. It also assumes a single-page walk on every STLB miss. Since this thesis is concerned with TLBs, it is important to model as many aspects as possible of the TLBs. Thus, we extend Sniper's baseline architecture to model the STLB miss penalty, multi-level pages, and realistic page walk penalty.

3.3.2 Architecture Configuration

Our baseline architecture is similar to the Intel Sunny Cove specifications. The architectural configurations are described in table 3.3. Regarding the sharing mechanism of the baseline pipeline, Table 3.4 demonstrates how each resource is being shared in the baseline SMT architecture in Sniper, except for the TLBs, which we tune as needed for each experiment to be either shared or private.

Parameter	Specification
Core	x86 Single core, with 2 SMT cores, runs at 2.66 GHz
ROB Size	352 entries
Reservation Stations Size	140 entries
L1 Data Cache	48 KB, 12-ways, MSHR: 10
L1 Instruction Cache	32 KB, 8-ways
L2 Cache	512 KB, 8-ways
L3 Cache	2 MB, 16-ways
TLB	ITLB: 128 entries, 8-ways DTLB: 64 entries, 4-ways STLB: 2048 entries, 16-ways, Access time = 8 cycles
Branch Predictor	Pentium-M Model
Prefetcher	GHB
Memory Latency	72 ns (191 cycles)
Page walking	4 KB pages, 4-level pages, page walk penalty = 764 cycles

Table 3.3: Specifications of the baseline architecture used

Resource	Sharing
TLBs	Private or shared, as per the experiment
L1, L2, and L3 Cache	Shared
ROB	Statically Partitioned, but redistributed once threads wake up/go asleep
Reservation Station	Shared
Branch Predictor	Split

Table 3.4: The sharing mechanism of resources in the baseline architecture

3.3.3 Evaluated Configurations and Performance Metrics

To evaluate the proposed architecture, we report the performance of the three system models. The first combination is non-cooperative threads executed on baseline architecture where the TLB is private per SMT core. The second is cooperative threads that are also executed on the same baseline architecture to evaluate the impact on the performance of co-locating cooperative threads. The last model is the actual proposed model, which is to run cooperative threads on shared TLB.

For each model, we run 600 million instructions in total on the two SMT cores and report the performance (in cycles) for each model. Also, we look into the TLB performance in terms of Misses-Per-Kilo-Instructions (MPKI). Furthermore, we report the amount of sharing between the threads of each benchmark, in terms of instruction and data page addresses, that is exhibited during execution time.

Because cooperative threads have the same address mapping, this is expected to affect all the cache levels since they are shared by default in the baseline SMT architecture. Thus, we report the performance of each cache in terms of MPKI.

3.4 Limitations

While our evaluation environment models many aspects of the SMT cores, there are still many characteristics that cause simulation inaccuracy in SniperSim. First, Sniper models correct-path execution only, which means it does not model pipeline flushes due to branch misprediction, memory order violations, or prefetching side-effects of wrong-path execution. Furthermore, the TLB currently models a single page size (i.e., 4-KB pages) and does not model huge pages. Thirdly, Sniper comes with an outdated branch predictor, which compromises the prediction accuracy. We discuss potential extensions to SniperSim in the future work section in chapter 5.

Chapter 4

Results and Discussion

As discussed in Chapter 3, we evaluate our work using Sniper [84], a multi-core trace-driven simulator. The configuration of the baseline architecture is similar to the Sunny Cove processor. We study the impact of co-locating cooperative threads when it is applied to the baseline architecture configuration, where the TLBs are private/split. We also analyze the effect of sharing the TLBs among cooperative threads from various perspectives.

We choose different representative benchmarks from each suite. From the DaCapo suite, we use avrora, batik, cassandra, eclipse, graphchi, h2, kafka, pmd, spring, sunflow, tomcat, tradebeans, tradesoap, and xalan. From the Renaissance suite, we use db-shootout, finagle-chirper, finagle-http, and neo4j-analytics. For all the benchmarks, we measure the performance gain, impact on TLBs and caches MPKI, the amount of sharing between the threads of each benchmark, and the sensitivity to the TLBs parameters from the performance perspective.

4.1 Impact on Performance

We first measure the execution time in cycles for the three setups; baseline configuration with non-cooperative threads running on baseline architecture with private TLBs, cooperative threads running on the same baseline architecture to measure the impact of the scheduling policy, and the cooperative threads running on the proposed architecture where the TLBs are shared. Figure 4.1 shows the performance gain in each case normalized to the baseline system. All the benchmarks achieve performance gain in each case of the proposed modifications. On average, the performance gain is 10% when cooperative threads are co-located on the baseline architecture, whereas sharing the TLB offers nearly an extra 5% more. Thus, in total, the proposed system achieves 15% performance gain. We justify the performance gain by only co-locating cooperative threads on baseline architecture in section 4.4.

For server benchmarks, such as Cassandra, Kafka, Finagle-Chirper, and Finagle-HTTP, they achieve the highest performance gain among all benchmarks. This is because they have

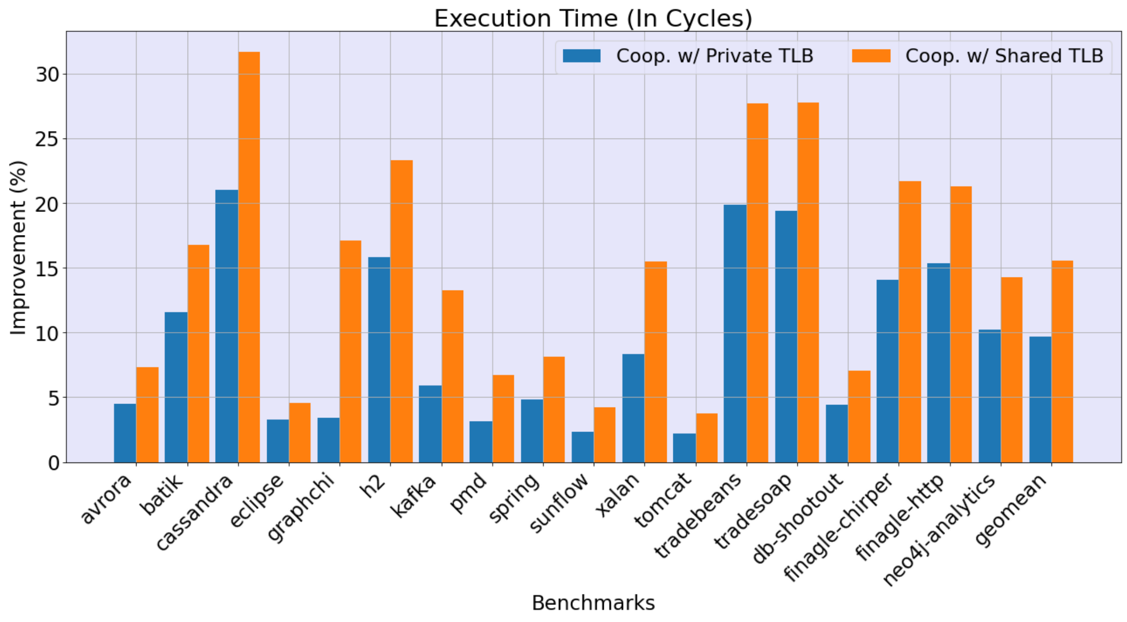

Figure 4.1: Performance gain normalized to performance of the baseline configuration (Non-cooperative on private TLBs) for cooperative threads running on private and shared TLB (The higher the better)

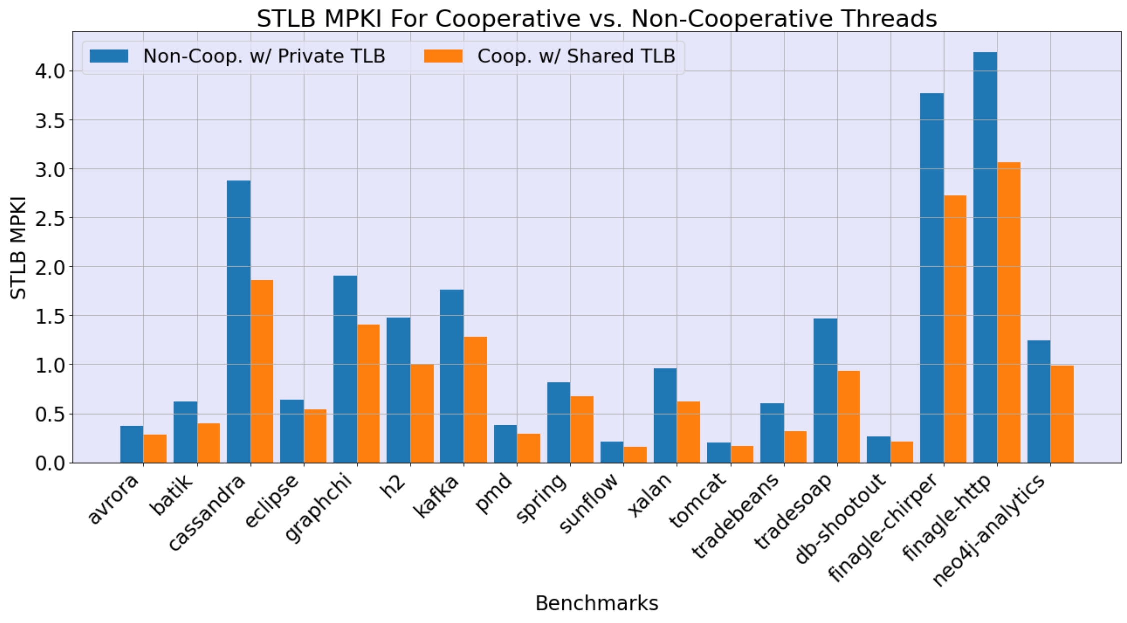

Figure 4.2: Absolute values for the STLB MPKI for each benchmark for non-cooperative threads running on baseline architecture and cooperative threads running on the proposed architecture

the largest STLB Miss-per-kilo-instruction (MPKI). The misses in STLB are more critical than the other levels of the TLB. This is because one miss in the STLB initiates a multi-level page walk that results in multiple memory accesses. Thus, by applying our proposed system, given that it reduces the misses in the STLB, this will accordingly result in fewer page walks, which enhances the performance significantly. Figure 4.2 shows the absolute values of the STLB MPKI for each benchmark.

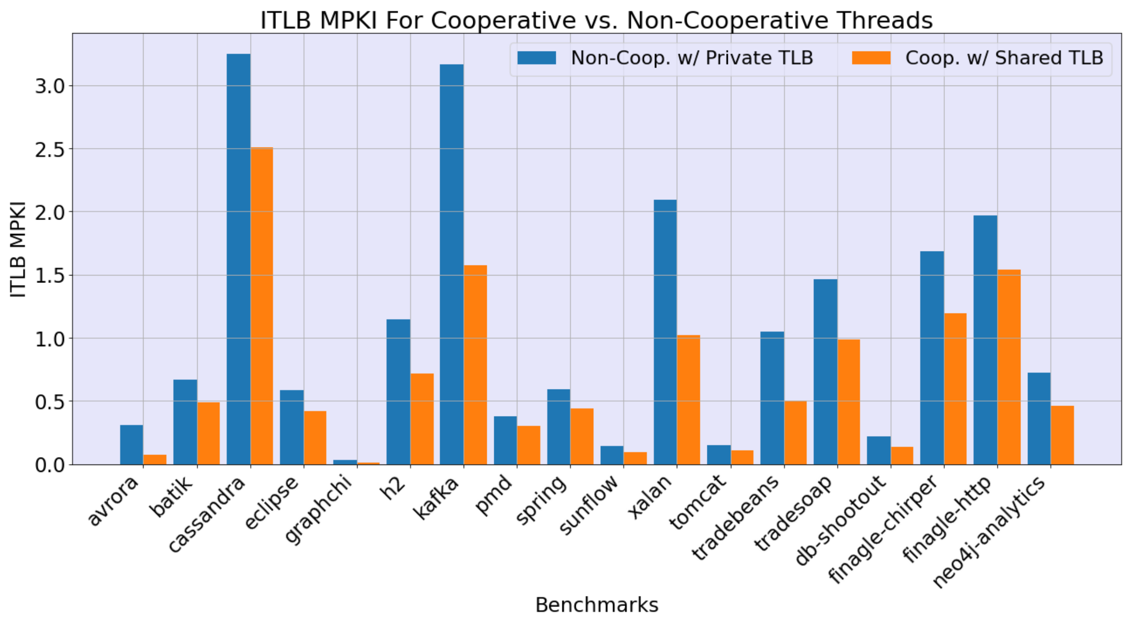

Figure 4.3: Absolute values for the ITLB MPKI for each benchmark for non-cooperative threads running on baseline architecture and cooperative threads running on the proposed architecture

Other benchmarks that exhibited less performance gain can also be justified by their low STLB MPKI. Thus, the benefit gained by sharing the TLBs is expected to be less than this achieved large-code footprint server benchmarks. The Tradebeans benchmark demonstrates performance gain, despite its low STLB MPKI. This is because it has relatively greater ITLB MPKI than most of its counterpart benchmarks, as shown in Figure 4.3, with low STLB MPKI. Thus, this benchmark may experience greater ITLB misses that are less likely to hit in the STLB. Unlike DTLB where its misses may have a relatively less severe impact on the pipeline, ITLB misses do not have this advantage and, thus, the entire pipeline gets stalled for hundreds of cycles until the instruction page address gets translated after accessing the memory multiple times. Figure 4.4 shows the absolute values of the DTLB MPKI.

Thus, in general, server workloads achieve speedups when they run on the proposed system, either with the TLBs private by just co-locating cooperative threads, or when the TLBs become shared among the cooperative threads.

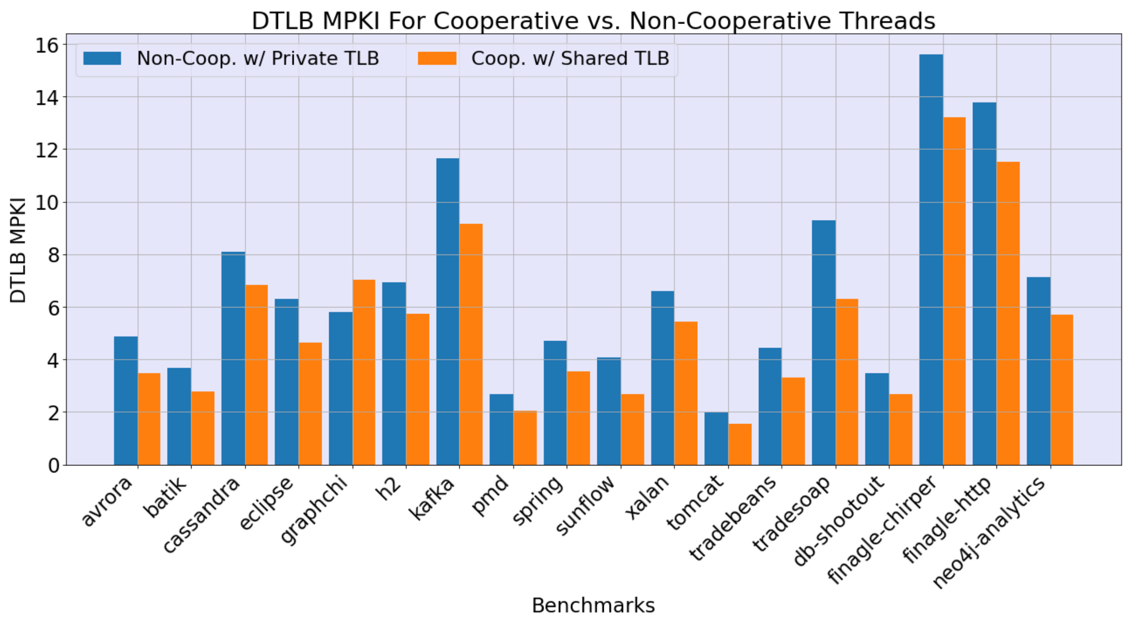

Figure 4.4: Absolute values for the DTLB MPKI for each benchmark for non-cooperative threads running on baseline architecture and cooperative threads running on the proposed architecture

4.2 TLB Performance

We take a closer look at the actual performance of the TLBs when they become shared. We consider the MPKI of each TLB stage since this is the metric that measures the direct impact of the misses on the execution time. In general,

As depicted by Figure 4.5, all the benchmarks achieve better STLB performance in terms of the MPKI. We care more about the STLB MPKI because its miss penalty is significant. The average gain for the benchmarks is 30%. Some of the benchmarks, such as tomcat, sunflow, and tradebeans, exhibit significantly more reductions in their STLB MPKI. This is because their absolute STLB MPKI value is low, thus, the smallest change in the MPKI would result in a significant gain.

We also consider the impact on the ITLB and DTLB MPKI. Figures 4.6 and 4.7 show the reduction in their MPKI, respectively. It is observed that, besides the significant reduction in the STLB MPKI, the benchmarks also exhibit a considerable reduction in the ITLB/DTLB MPKI. However, this does not have a similar impact on the performance since a miss in the L1 TLB (i.e., ITLB and DTLB) is usually backed up by the STLB, thus. Analogous to the observation in the STLB, some benchmarks also manifest noticeably larger reductions in the ITLB MPKI due to their very low absolute value of their ITLB MPKI, such as graphchi and avrora, as depicted in Figure 4.3. A third observation is related to the DTLB MPKI

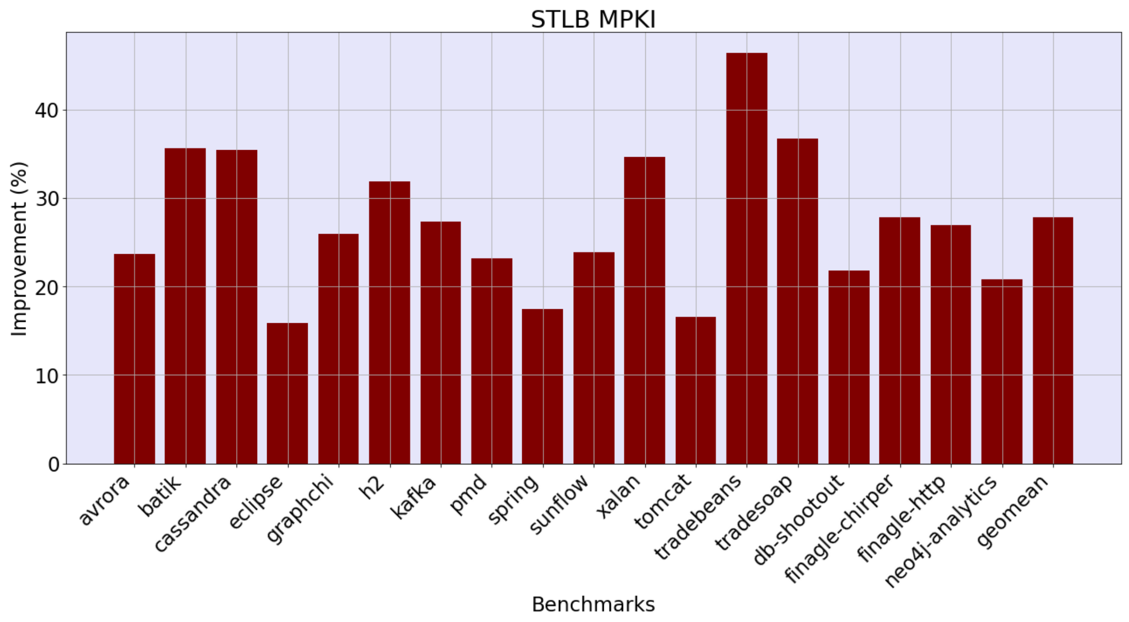

Figure 4.5: Reduction achieved in the STLB MPKI due to co-locating cooperative threads on the same core with STLB shared (The higher the better)

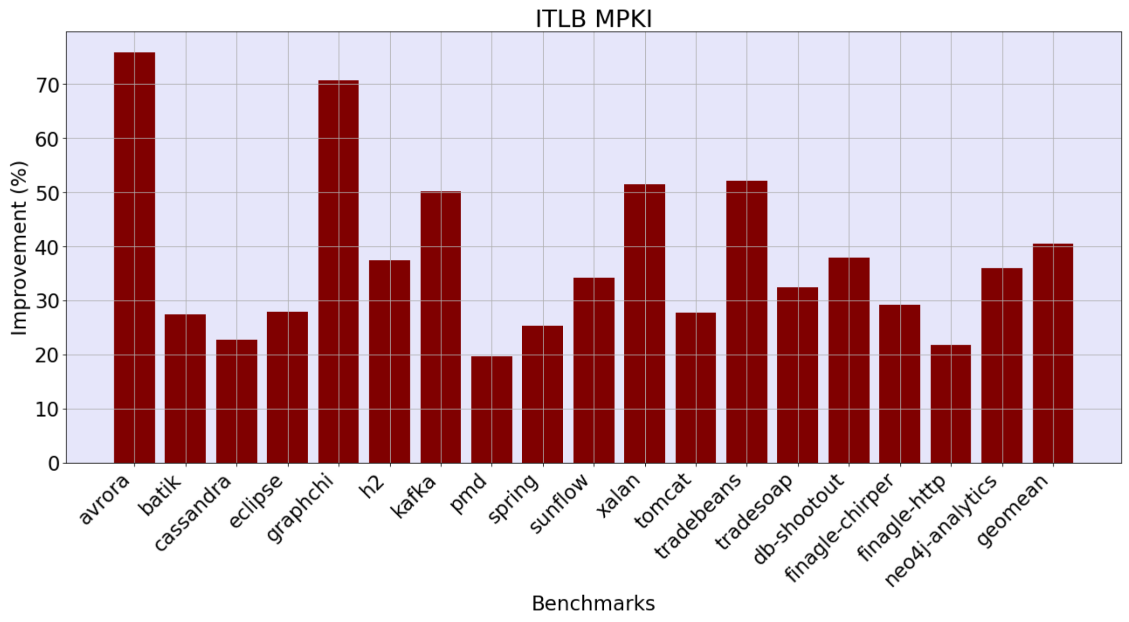

Figure 4.6: Reduction achieved in the ITLB MPKI due to co-locating cooperative threads on the same core with ITLB shared (The higher the better)

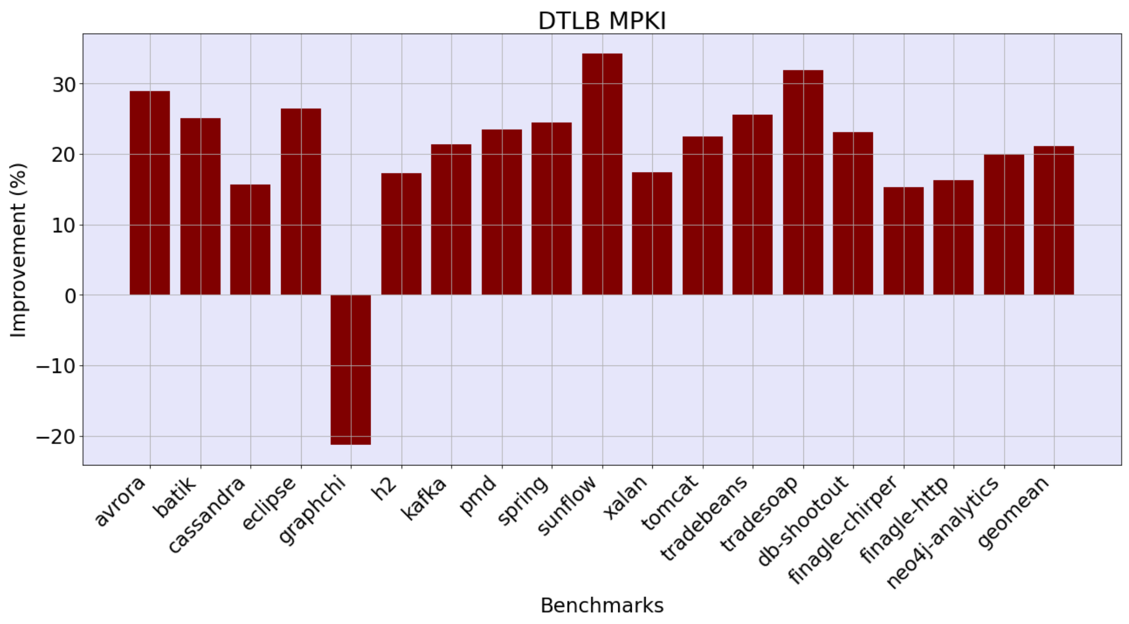

Figure 4.7: Reduction achieved in the DTLB MPKI due to co-locating cooperative threads on the same core with DTLB shared (The higher the better)

increase in graphchi benchmark when the TLBs become shared, which is an anomaly to the expected outcomes of the proposed system. We study this anomaly in-depth in section 4.3.3.

Overall, the reduction in the TLBs MPKI goes along with the expected outcome of the hypothesis of this study, if a certain feature of the benchmarks is considered. Figures 4.8 and 4.9 explain this outcome from the perspective of quantifying the sharing between the threads of each benchmark, for instructions and data, respectively. We measure the percentage of pages that were accessed by both threads/cores at any given moment during the execution time. This reveals a valuable observation which is the amount of sharing between the threads is significant that, if a resource is shared, it is expected to achieve a considerable performance gain. Moreover, the multithreading paradigm in itself aims at distributing the work among threads that execute in parallel, which means these threads are likely to be worker threads that do similar operations but on different data.

The Figures 4.8 and 4.9 confirm this claim. While the average amount of the sharing of instruction pages is more than 50%, the data sharing on average is way less than that (30%). This also agrees with the nature of each benchmark. For example, because sunflow and xalan benchmarks perform processing of different files, we see that they share many instruction pages but relatively fewer data pages than the other benchmarks. Also, because server benchmarks like Cassandra and h2 have worker threads that receive different requests from clients, their instruction page sharing is relatively higher, unlike their data page sharing.

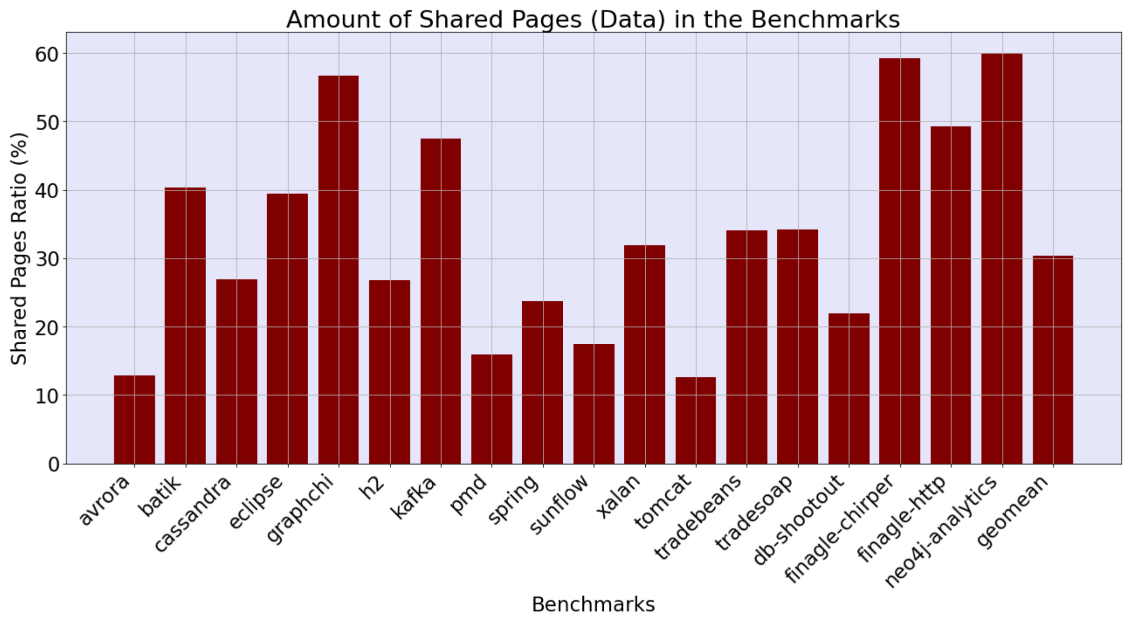

Figure 4.8: The percentage of shared data page addresses among the threads of each benchmark normalized to all the pages accessed

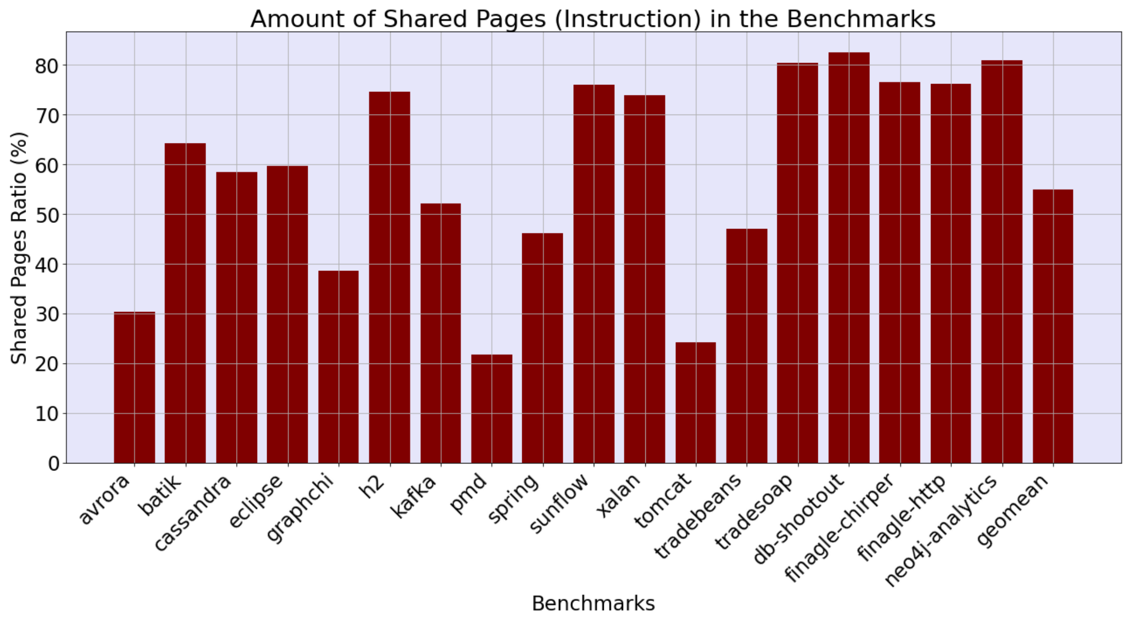

Figure 4.9: The percentage of shared instruction page addresses among the threads of each benchmark normalized to all the pages accessed

4.3 Sensitivity Analysis

To study the scalability of each configuration of the proposed system (i.e., cooperative threads running on shared and private TLBs), we study their sensitivity to the different sets of configurations for each of the TLBs; ITLB, DTLB, STLB, in terms of the number of entries and associativity. The results in each plot are normalized to the configuration with the smallest parameter value.

4.3.1 TLB Size

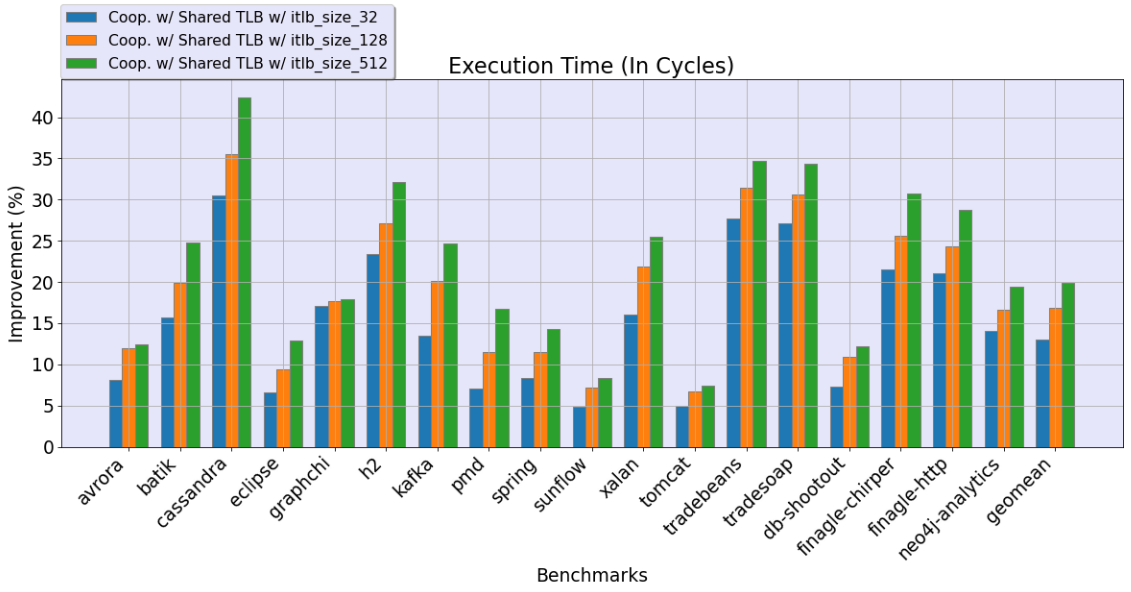

Figure 4.10: Performance gain when cooperative threads run on **shared** TLBs where the size of the ITLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

We analyze the sensitivity of the proposed system to the variation in each TLB size. Figures 4.10 and 4.11 show proportional relation between the ITLB size and the improvement in the execution time. This is because it allows more entries to be allocated into the ITLB without evicting other lines. Also, these figures concur that, for cooperative threads, sharing the TLBs provides more performance gain than when the TLBs are private. As the ITLB size is increased 4 times, the benefit gained is not more than 5% when the TLBs are shared or private. Yet, shared TLBs provide better overall performance gain.

Unlike the ITLB, the DTLB size variation gives a better improvement rate in the performance for every increase in the DTLB size. Figures 4.12 and 4.13 show that, for every 4 times increase in the DTLB size, 8% performance gained achieved. However, this ratio reduces significantly when the size increases from 64 entries to 256 entries (i.e., 5-6%). Yet,

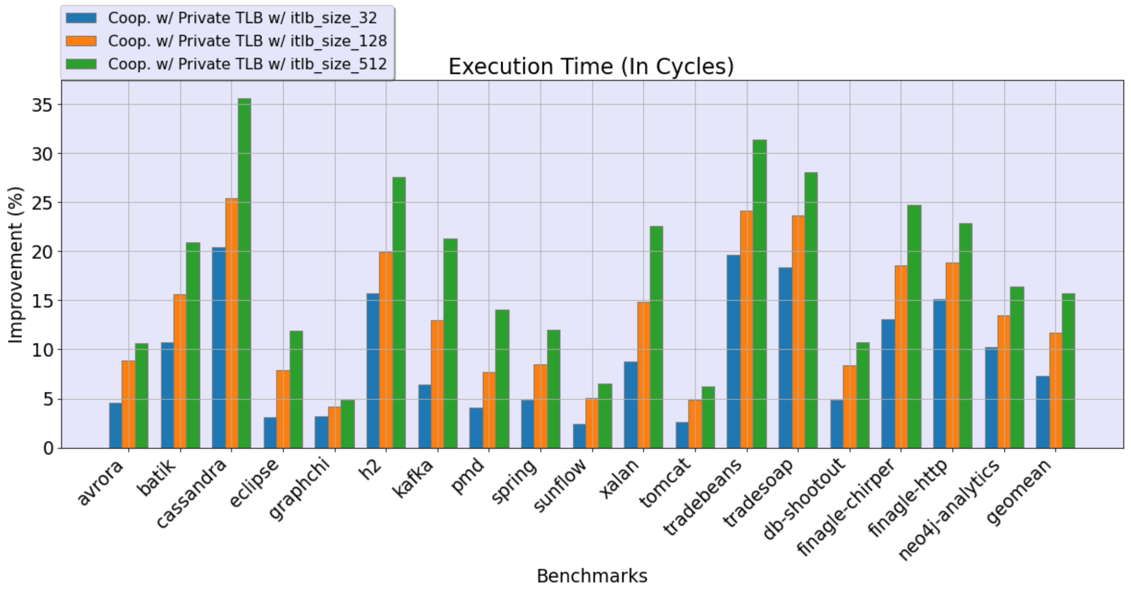

Figure 4.11: Performance gain when cooperative threads run on **private** TLBs where the size of the ITLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

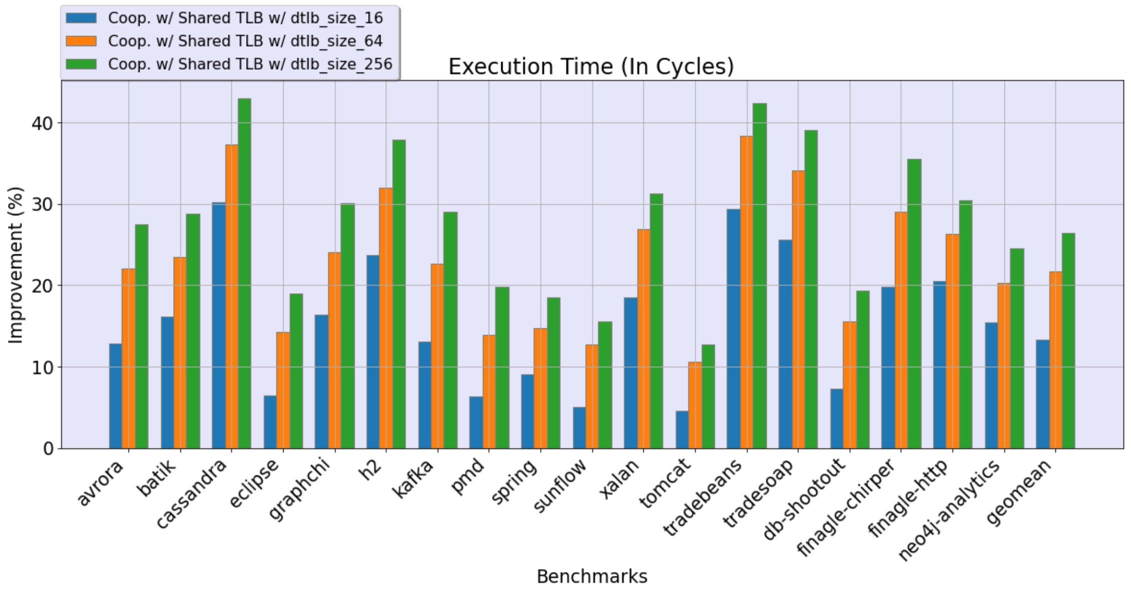

Figure 4.12: Performance gain when cooperative threads run on **shared** TLBs where the size of the DTLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

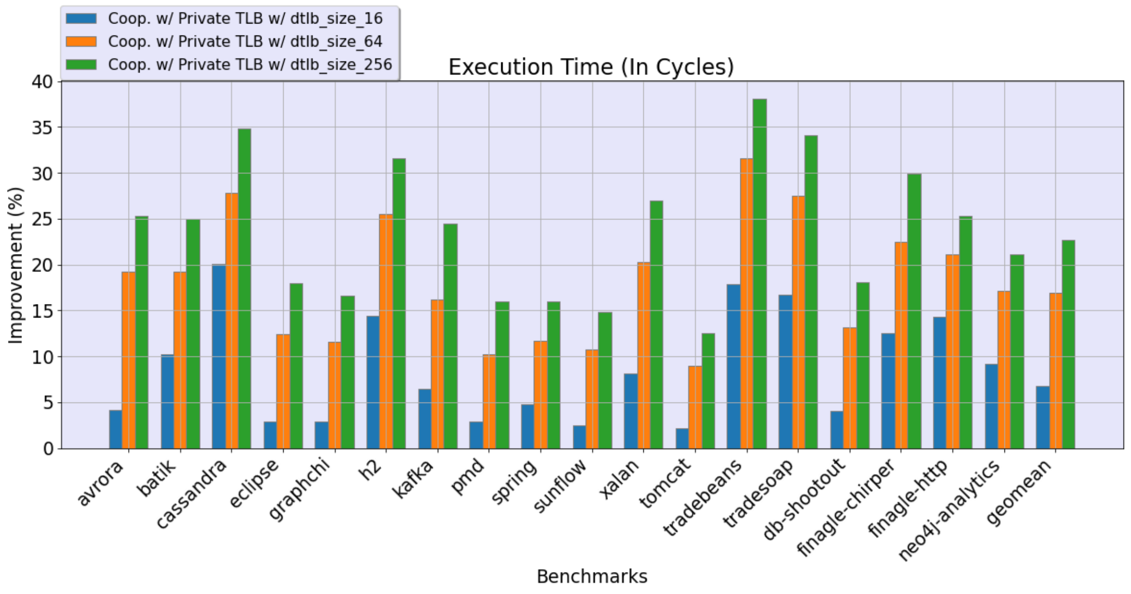

Figure 4.13: Performance gain when cooperative threads run on **private** TLBs where the size of the DTLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

more gain is achieved in this case than in the case of increasing the ITLB size because the data page addresses exhibit less sharing, thus, a bigger DTLB would serve the private entries that are accessed by only one thread/core.

Similar to the DTLB, increasing the STLB size also gives a proportional increase in performance improvement. Yet, the same improvement is achieved by increasing the STLB size 2 times, unlike the DTLB where the same benefit was achieved by increasing the DTLB size 4 times. Thus, the system is more sensitive to the STLB size than the ITLB and the DTLB sizes. This is because increasing the cache size reduces the capacity misses, which alleviates the miss penalty impact on the performance because the miss penalty in the STLB has a more detrimental impact on the performance, compared to the prior TLB levels.

4.3.2 TLB Associativity

Concerning the variation in the associativity, we study the impact of increasing the associativity in each TLB. Figures 4.16 and 4.17 show that a 4-time increase in the ITLB associativity maps to a little to no improvement overall. Since the data sharing between the threads of each benchmark is lower relative to the sharing in instructions, increasing the DTLB associativity results in a noticeable expected performance gain. This is highlighted by Figures 4.18 and 4.19 where an average gain of 2% is achieved when the associativity is doubled.

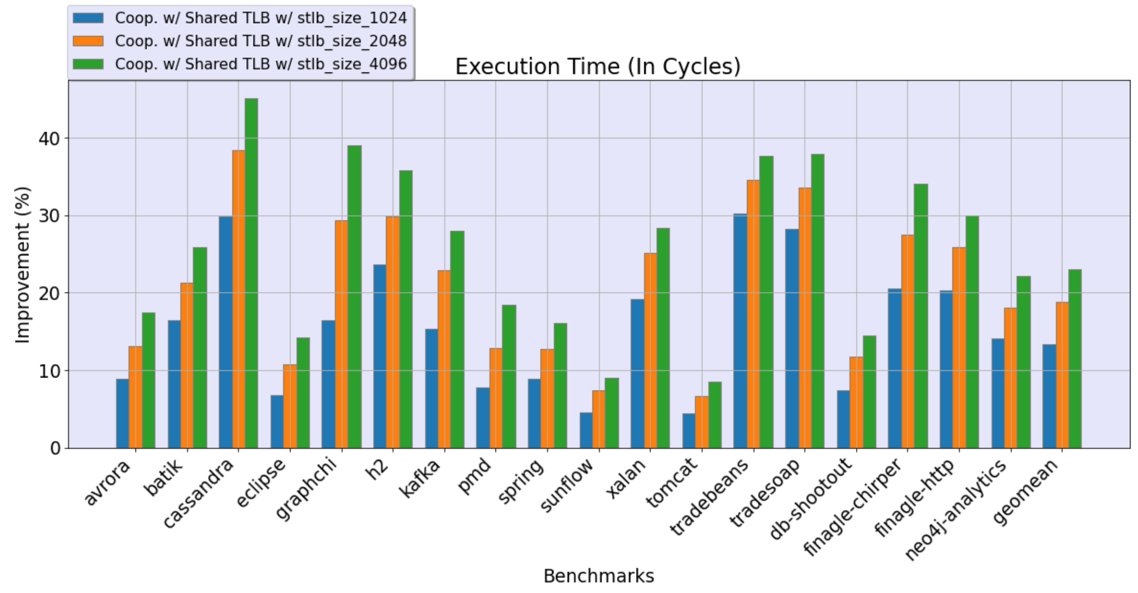

Figure 4.14: Performance gain when cooperative threads run on **shared** TLBs where the size of the STLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

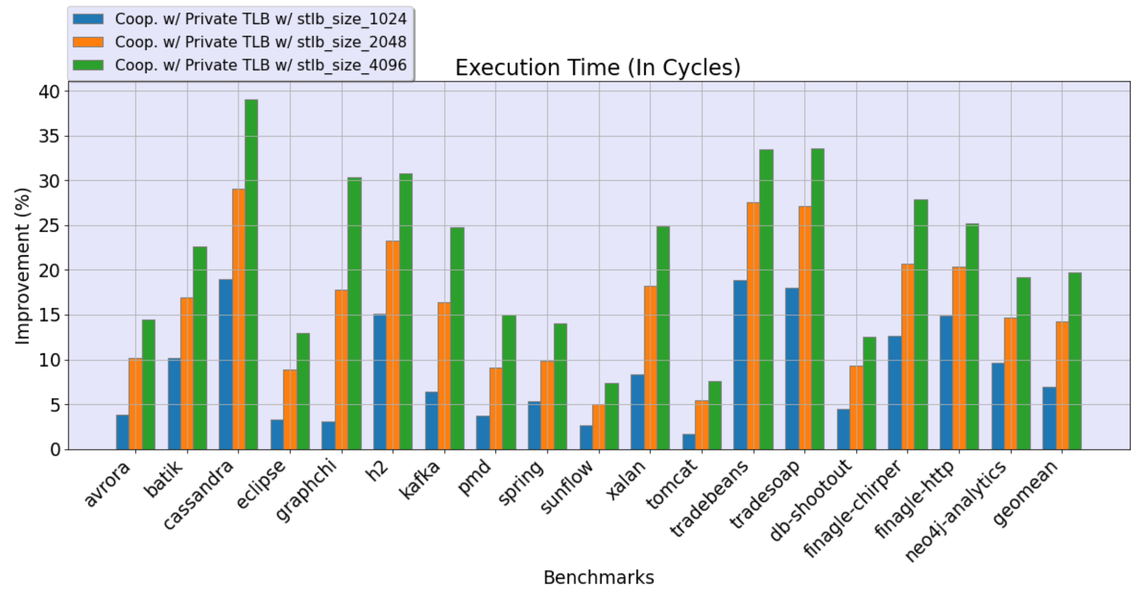

Figure 4.15: Performance gain when cooperative threads run on **private** TLBs where the size of the STLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

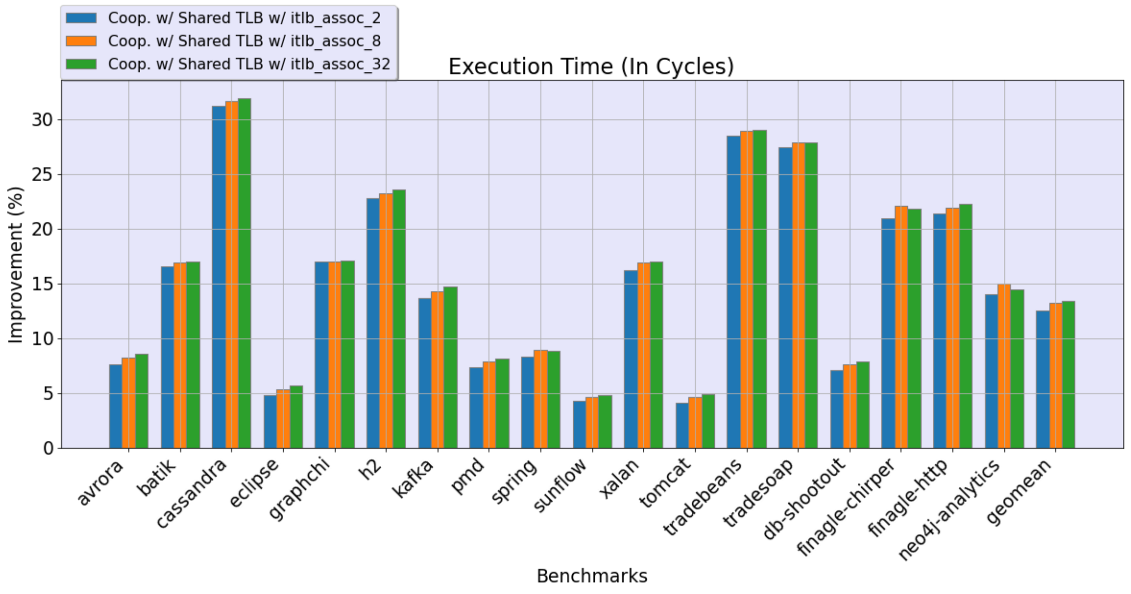

Figure 4.16: Performance gain when cooperative threads run on **shared** TLBs where the associativity of the ITLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

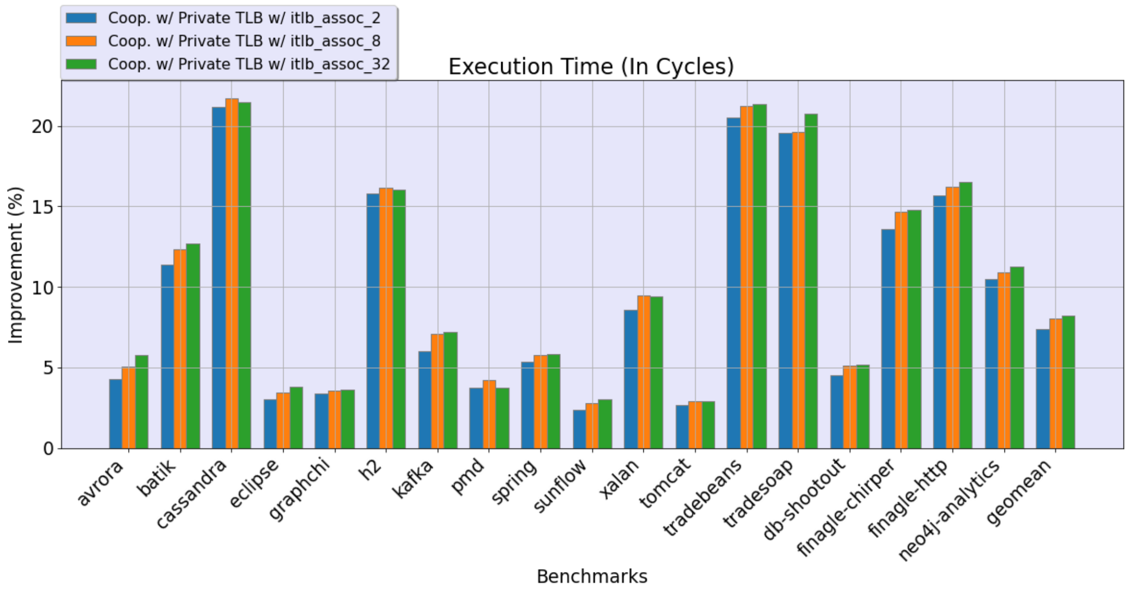

Figure 4.17: Performance gain when cooperative threads run on **private** TLBs where the associativity of the ITLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

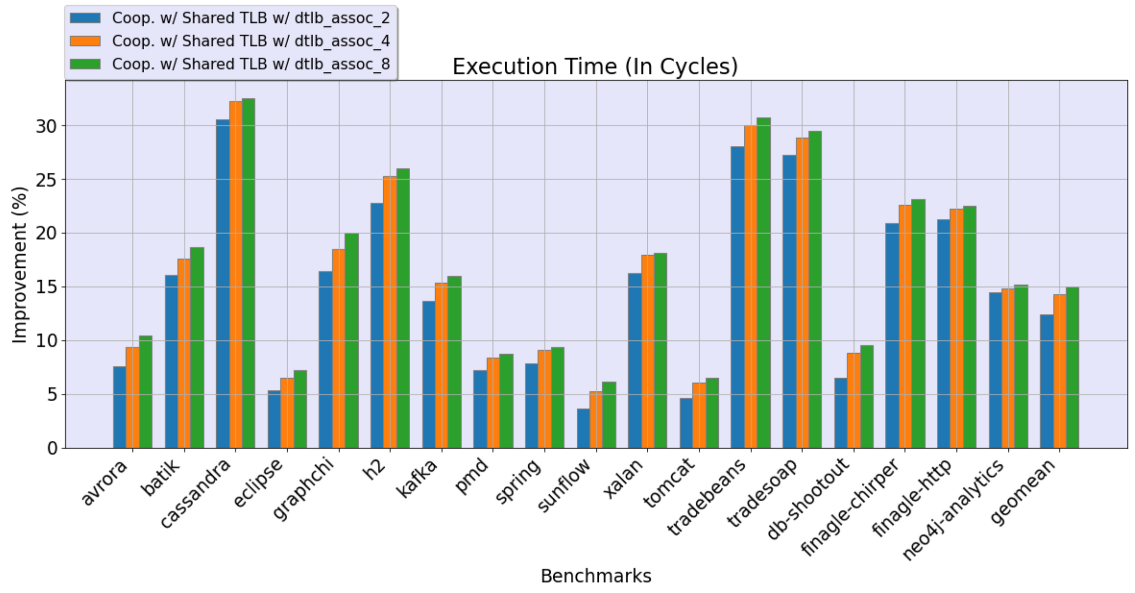

Figure 4.18: Performance gain when cooperative threads run on **shared** TLBs where the associativity of the DTLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

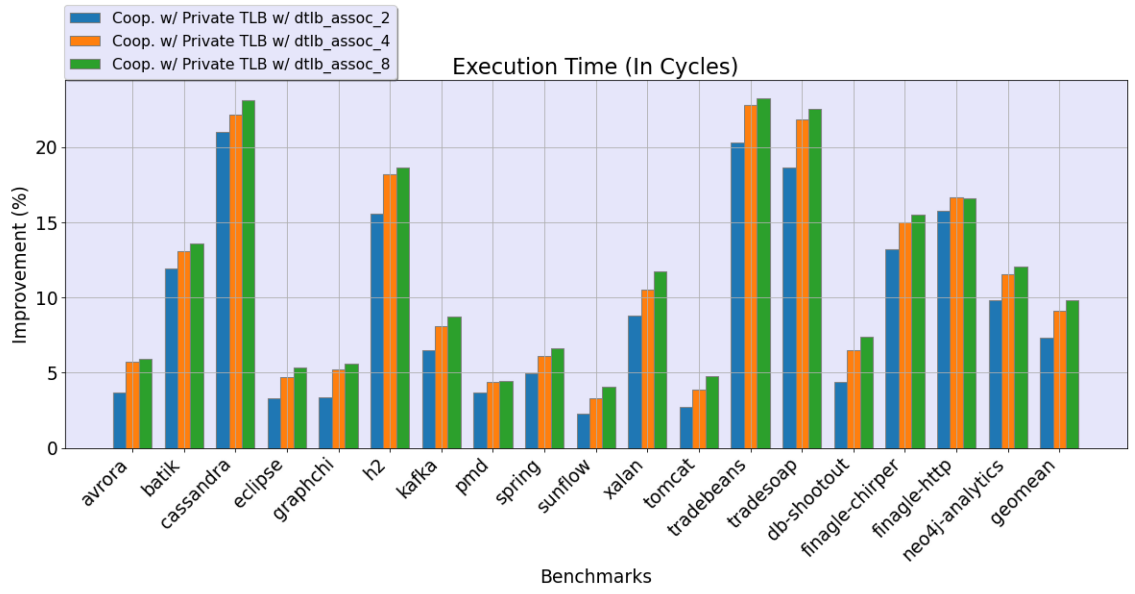

Figure 4.19: Performance gain when cooperative threads run on **private** TLBs where the associativity of the DTLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

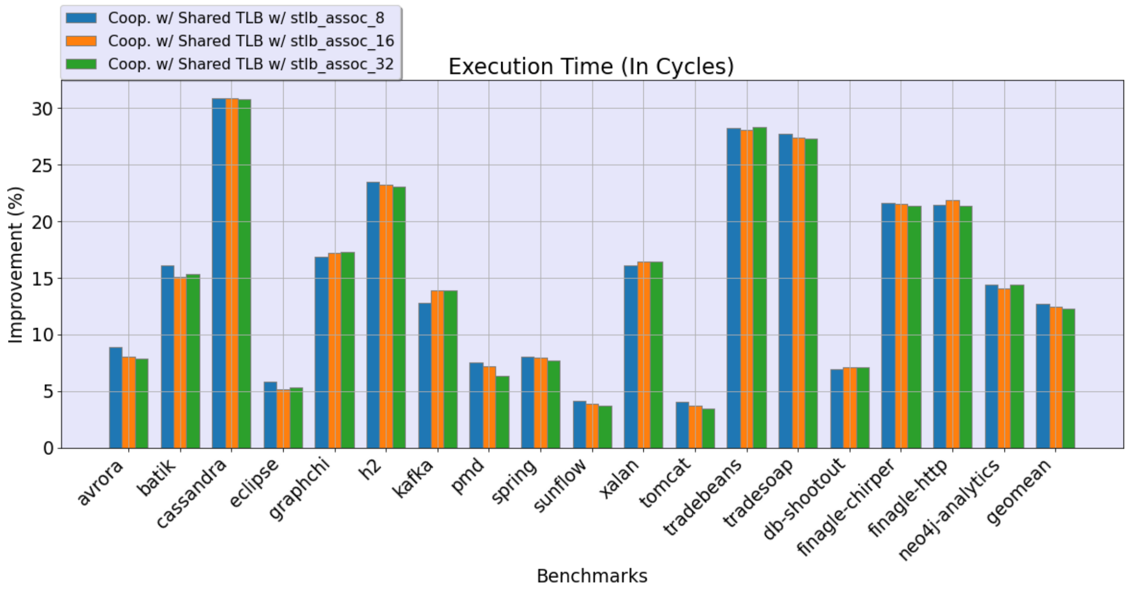

Figure 4.20: Performance gain when cooperative threads run on **shared** TLBs where the associativity of the STLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

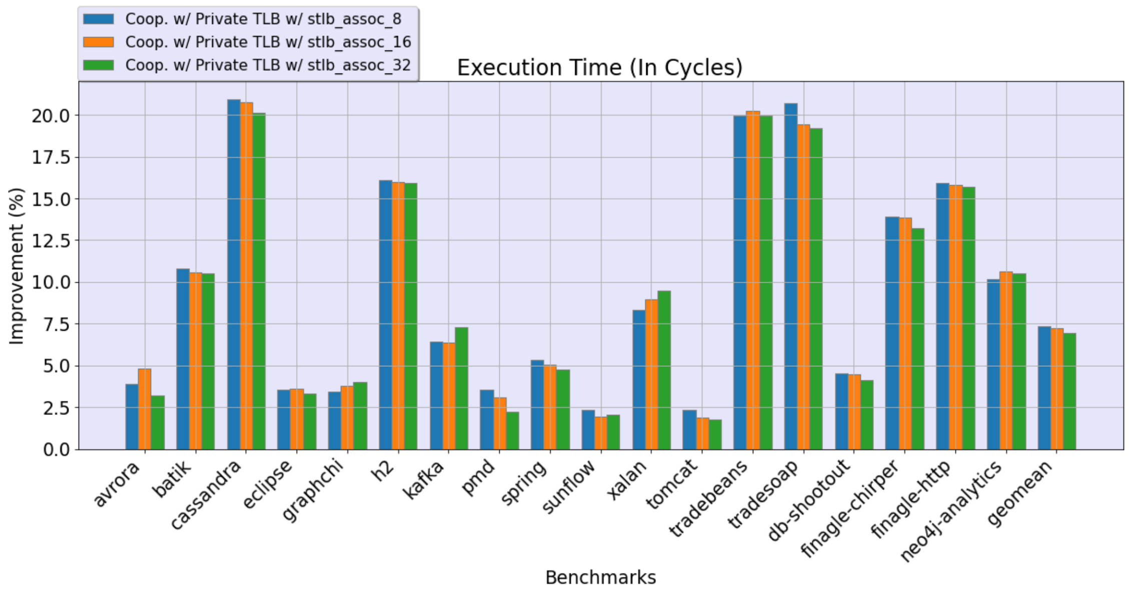

Figure 4.21: Performance gain when cooperative threads run on **private** TLBs where the associativity of the STLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

On the other hand, as observed in Figures 4.20 and 4.21, doubling the STLB associativity almost has no impact on the performance.

4.3.3 DTLB MPKI

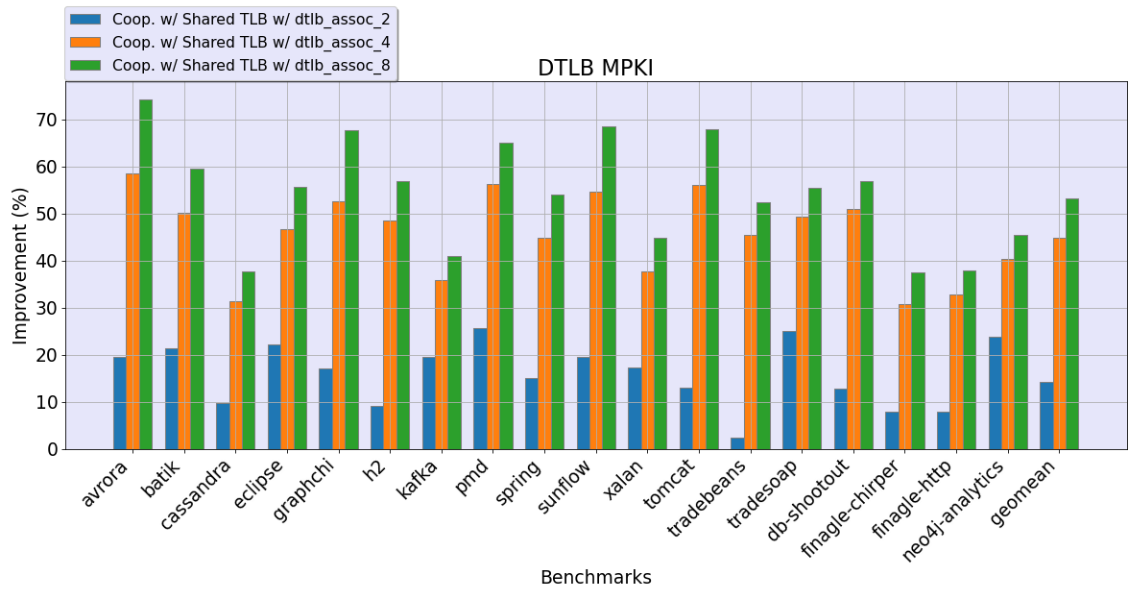

Figure 4.22: Reduction in the DTLB MPKI when cooperative threads run on **shared** TLBs where the associativity of the DTLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

We only consider the change in the DTLB MPKI to analyze the anomaly in Figure 4.7 where graphchi benchmark exhibited an increase in its MPKI when the DTLB is shared between the cooperative threads. Figure 4.22 shows the variation in the DTLB MPKI as its associativity increases for cooperative threads when they run on shared TLBs. This can be explained by that when the DTLB is shared between the threads of the graphchi benchmark, more entries are mapped to the same set, resulting in more conflict misses. By increasing the TLB associativity, more entries can fit into the same set, which eliminates this phenomenon.

4.3.4 STLB Miss Penalty

Besides the size and associativity, we also study how our proposed system behaves when the STLB miss penalty varies. This is crucial because, conventionally, the number of memory accesses required for page walking varies, depending on the page table and its presence in the cache hierarchy. Hence, assuming a constant number of memory accesses per page walk represents an unrealistic assumption. Therefore, we study how the overall performance

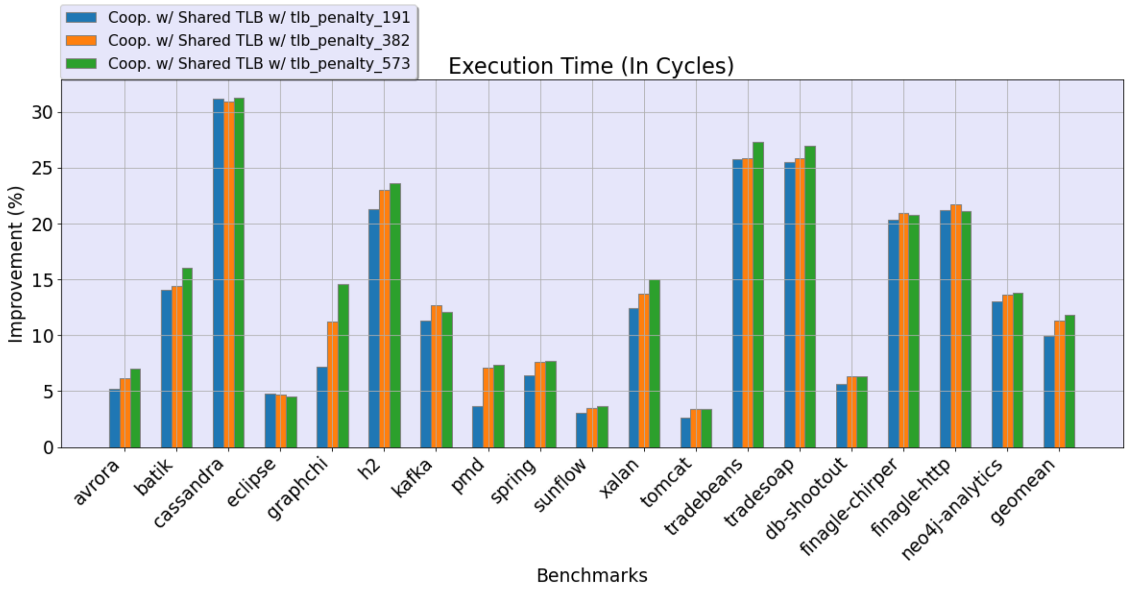

Figure 4.23: Performance gain when cooperative threads run on **shared** TLBs where the miss penalty of the STLB varies, normalized to the configuration with the smallest parameter value (The higher the better)

gain varies with respect to the variation in the STLB miss penalty. Figure 4.23 shows this variation.

4.4 Extra Benefit in Caches

While this study focuses mainly on analyzing the related aspects of the proposed architecture to co-locate cooperative threads and share the TLBs among them, it is still important to study the other components that are affected by the proposed system. For that case, we consider the memory caches, namely L1 instruction/data cache, L2 cache, and L3 cache. The difference between the memory cache and the TLB is that TLBs are accessed by the virtual address, whereas the caches are accessed using the physical address.

Conventionally, the SMT cores share all the cache levels because then the threads will be running on the same physical core. One observation is that if two threads of the same address space share the cache where a common virtual address is translated into the same physical memory location, this means that each thread will be fetching the same block into the cache. Thus, one thread could help the other thread in that case.

Hence, by just co-locating the cooperative threads on the same physical core, it is expected to observe performance gain in the cache hierarchy. Figures 4.24, 4.25, 4.26, and 4.27 show this benefit in terms of the reduction that happened in the MPKI of each cache.

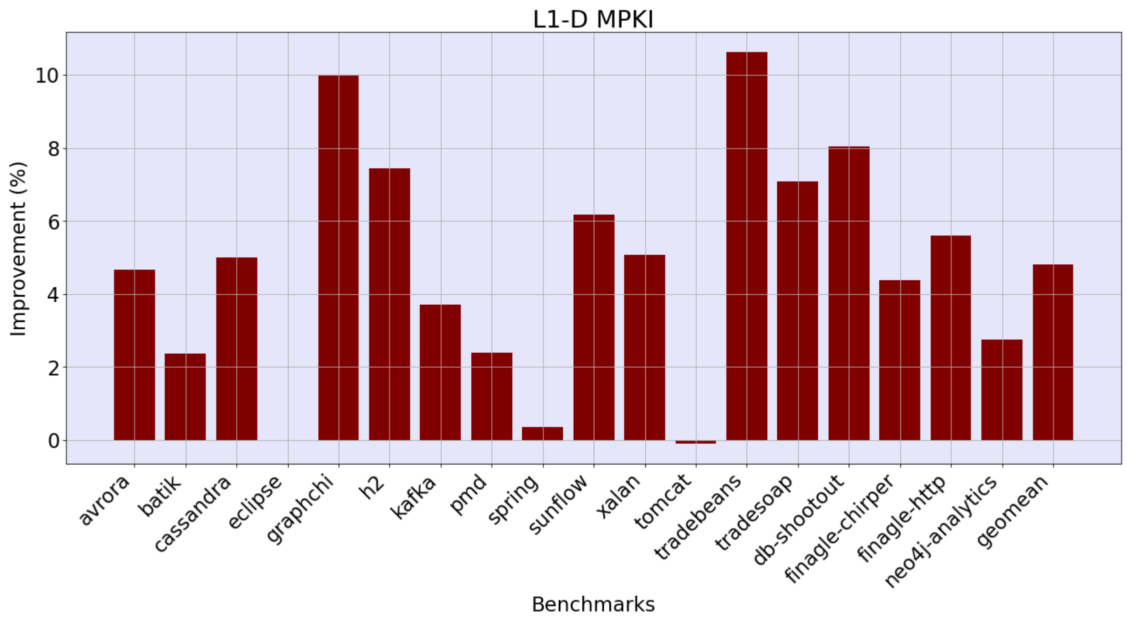

Figure 4.24: Reduction achieved in the L1-D MPKI due to co-locating cooperative threads on the same core (The higher the better)

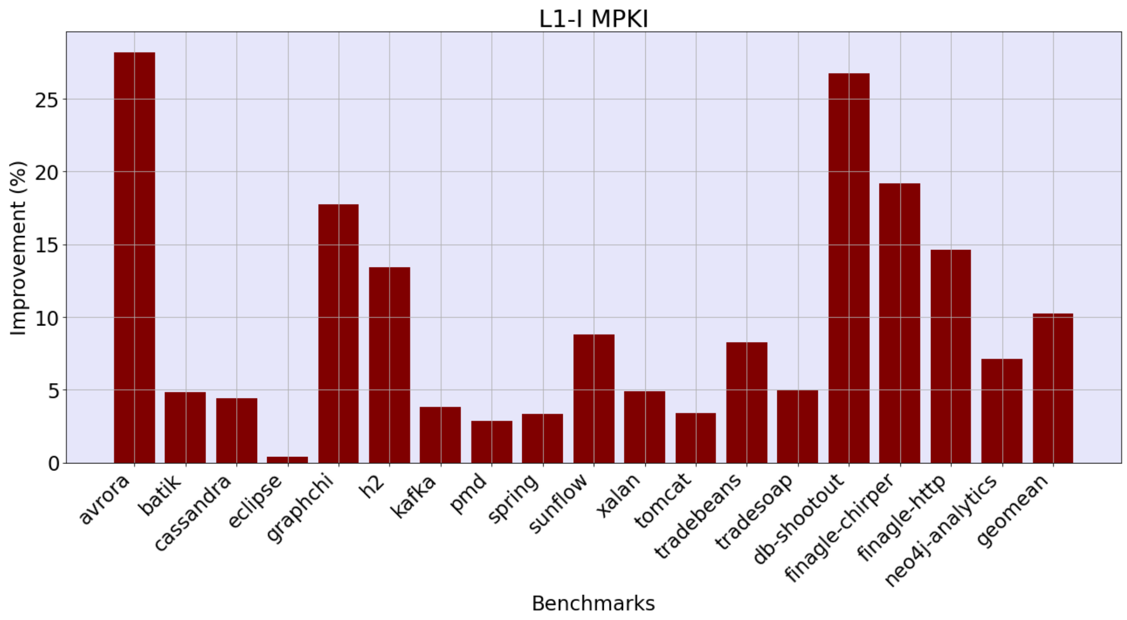

Figure 4.25: Reduction achieved in the L1-I MPKI due to co-locating cooperative threads on the same core (The higher the better)

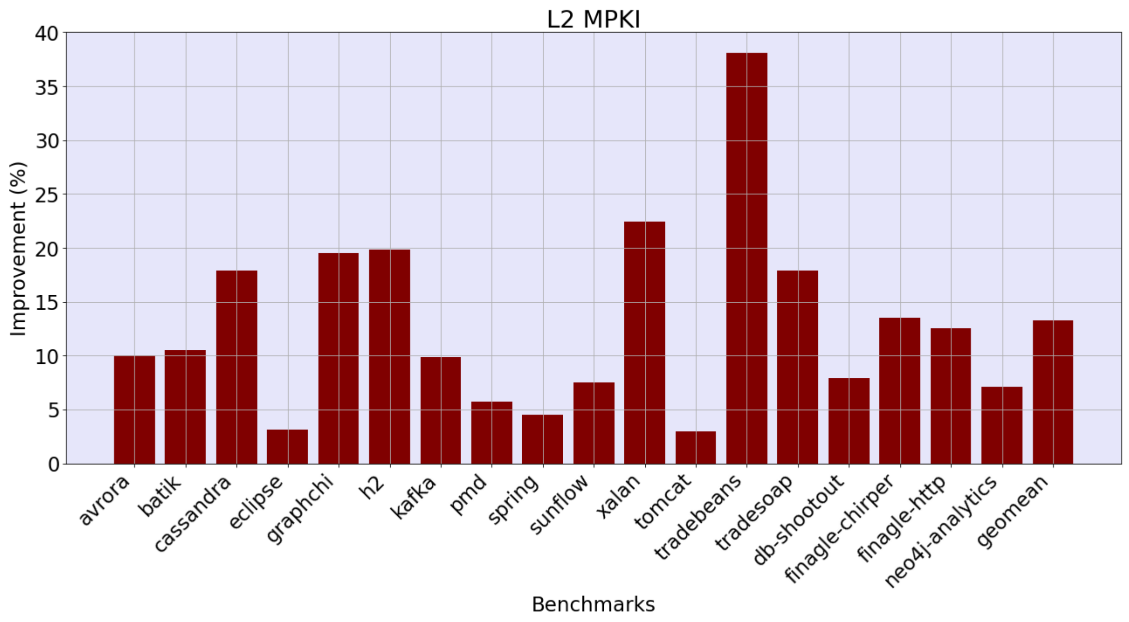

Figure 4.26: Reduction achieved in the L2 MPKI due to co-locating cooperative threads on the same core (The higher the better)

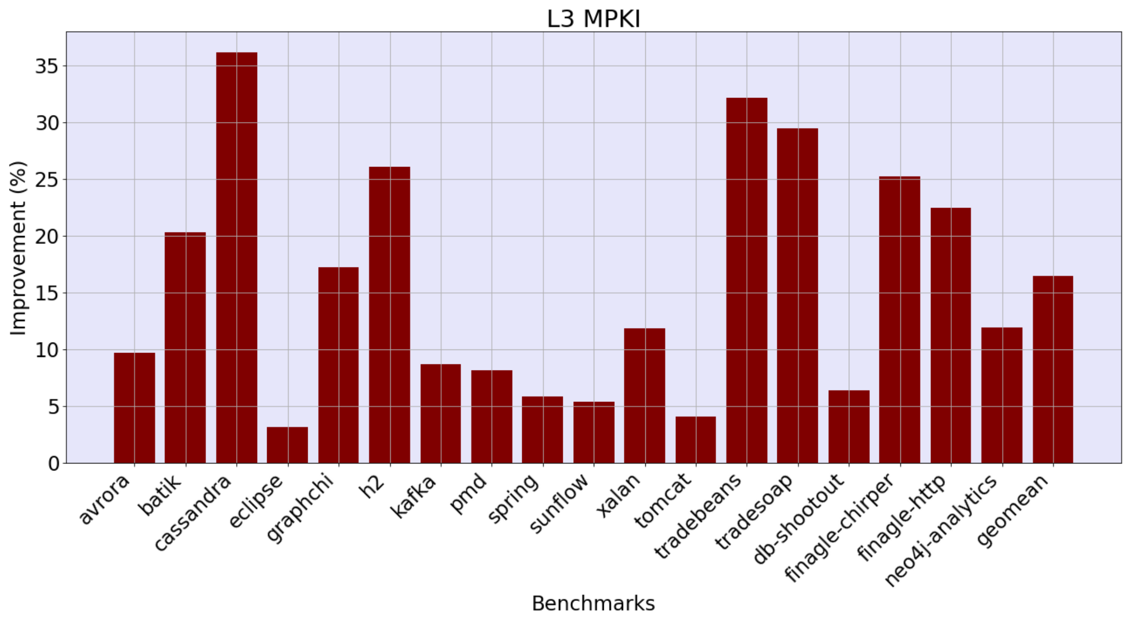

Figure 4.27: Reduction achieved in the L3 MPKI due to co-locating cooperative threads on the same core (The higher the better)

Because the amount of sharing between the threads is more significant in the case of instruction pages than data pages, the L1 instruction cache achieves more reduction in its MPKI than the L1 data cache.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this work, we showed that cooperative threads can significantly help each other if co-located on logical cores of the same physical core, such that they share CPU resources. In server workloads, where the code footprint is huge, one thread can fetch data that the other threads need, which hides the miss latency any of these threads may encounter.

By only co-locating cooperative threads on the same core with no further modification in the microarchitecture, an average performance gain of 10% can be achieved, which results in better QoS. This can be justified by that, because cooperative threads share data and instruction pages, they bring

We showed that the reason behind this performance gain was the shared cache hierarchy whose MPKI reduced significantly. The L1 instruction cache exhibited an average of 10% reduction in the MPKI, whereas the L1 data cache demonstrated less reduction (i.e., more than 4%). Also, the L2 and L3 caches showed an average MPKI reduction of approximately 15%.

More importantly, we proposed our modified system architecture where the TLBs can be shared among SMT threads if they are identified to be cooperative. With this proposed architecture, an extra 5% performance gain can be achieved, leading to a total of 15% on average when the cooperative threads run on the proposed architecture. We showed that, by sharing all the TLB levels, the MPKI of the ITLB, DTLB, and STLB is reduced by 40%, 20%, and 30% on average, respectively.

We attribute this performance gain to the amount of sharing between the threads of each benchmark, which is analyzed in this thesis from the perspective of the instruction and data pages. For large-code-footprint workloads, it can be seen that their cooperative threads can share 30% of their data pages and more than 50% of their instruction pages, on average.

Lastly, we study the scalability and sensitivity of our system when the TLB parameters vary with each of our proposed configurations. By increasing the size of, at least, one of the

TLBs, the performance gain increases proportionally. On the other hand, when the I/DTLB associativity increases, this results in a relatively lower performance gain. However, varying the STLB associativity has almost no effect on the performance.

5.2 Future Work

5.2.1 Other CPU Structures

In future work, we plan to improve the accuracy of the simulation in Sniper by adding support for more realistic components, such as adding support for large page sizes in the TLB, and implementing a state-of-the-art branch predictor and prefetchers. Also, it is essential to extend the proposal to share other resources among threads when they are known to be cooperative. In particular, we will analyze the consequences of sharing other structures, such as branch predictors, and micro-op caches.

5.2.2 Extending to CMP

Even though this proposal mainly targets SMT cores, it may have the potential to be extended to the CMPs. The SMT and CMPs are different ways to achieve parallelism. The former achieves it through sharing the on-core resources, whereas the latter achieves it by increasing the number of on-chip cores. Thus, in a mechanism similar to sharing the last-level cache, the STLB can also be shared among the CMP cores, if they execute cooperative threads. The same logic that controls the sharing policy of the STLB may thus be extended to operate similarly in the CMP setup.

Yet, other factors have to be studied in this case. For example, if the latency caused by locating the STLB off-core outweighs the benefit, it remains a more feasible option to keep the STLB local to each core. Therefore, a more thorough study of this approach is needed.

Hence, we shall analyze how relocating the STLB off-core will impact the performance while considering the aforementioned factors.

Bibliography

[1] M. Nemirovsky and D. Tullsen, *Multithreading architecture.* Springer Nature, 2022.

[2] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd annual international symposium on Computer architecture*, 1995, pp. 392–403.

[3] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proceedings of the 23rd annual international symposium on Computer architecture*, 1996, pp. 191–202.

[4] M. Taram, X. Ren, A. Venkat, and D. Tullsen, "{SecSMT}: Securing {SMT} processors against {Contention-Based} covert channels," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3165–3182.

[5] W. Jia, J. Shan, T. O. Li, X. Shang, H. Cui, and X. Ding, "{vSMT-IO}: Improving {I/O} performance and efficiency on {SMT} processors in virtualized clouds," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 449–463.

[6] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot, "Stretch: Balancing qos and throughput for colocated server workloads on smt cores," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2019, pp. 15–27.

[7] X. Yang, S. M. Blackburn, and K. S. McKinley, "Elfen scheduling:{Fine-Grain} principled borrowing from {Latency-Critical} workloads using simultaneous multithreading," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 309–322.

[8] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 450–462.

[9] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *Acm sigplan notices*, vol. 47, no. 4, pp. 37–48, 2012.

[10] A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Enhancing server efficiency in the face of killer microseconds," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2019, pp. 185–198.

[11] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 406–418.

[12] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 158–169.

[13] F. J. Cazorla, E. Fernandez, A. Ramírez, and M. Valero, "Improving memory latency aware fetch policies for smt processors," in *High Performance Computing: 5th International Symposium, ISHPC 2003, Tokyo-Odaiba, Japan, October 20-22, 2003. Proceedings 13 5*. Springer, 2003, pp. 70–85.

[14] S. Everman and L. Eeckhout, "A memory-level parallelism aware fetch policy for smt processors," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007, pp. 240–249.

[15] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, 2000, pp. 234–244.

[16] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 47–58, 2007.

[17] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 223–234, 2012.

[18] M. Gulati and N. Bagherzadeh, "Performance study of a multithreaded superscalar microprocessor," in *Proceedings. Second International Symposium on High-Performance Computer Architecture*. IEEE, 1996, pp. 291–301.

[19] W. Yamamoto and M. Nemirovsky, "Increasing superscalar performance through multistreaming." in *PaCT*, vol. 95, 1995, pp. 49–58.

[20] W. Yamamoto, M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirosky, "Performance estimation of multistreamed, superscalar processors," in *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, vol. 1. IEEE, 1994, pp. 195–204.

[21] D. Koufaty and D. T. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 56–65, 2003.

[22] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture." *Intel Technology Journal*, vol. 6, no. 1, 2002.

[23] M. Dixon, P. Hammarlund, S. Jourdan, and R. Singhal, "The next-generation intel core microarchitecture." *Intel Technology Journal*, vol. 14, no. 3, 2010.

[24] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, "An elementary processor architecture with simultaneous instruction issuing from multiple threads," in *Proceedings of the 19th annual international symposium on Computer architecture*, 1992, pp. 136–145.

[25] R. Govindarajan, S. S. Nemawarkar, and P. LeNir, "Design and performance evaluation of a multithreaded architecture," in *Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture.* IEEE, 1995, pp. 298–307.

[26] B. K. Gunther, "Superscalar performance in a multi threaded microprocessor," Ph.D. dissertation, University of Tasmania, 1993.

[27] C. J. Beckmann and C. D. Polychronopoulos, "Microarchitecture support for dynamic scheduling of acyclic task graphs," *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 140–148, 1992.

[28] H. S. Stone, J. Turek, and J. L. Wolf, "Optimal partitioning of cache memory," *IEEE Transactions on computers*, vol. 41, no. 09, pp. 1054–1068, 1992.

[29] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers, "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 3, pp. 322–354, 1997.

[30] 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.* IEEE, 2002, pp. 117–128.

[31] G. Suh, L. Rudolph, and S. Devadas, "Dynamic cache partitioning for cmp/smt systems," *Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, 2004.

[32] R. Iyer, "Cqos: a framework for enabling qos in shared caches of cmp platforms," in *Proceedings of the 18th annual international conference on Supercomputing*, 2004, pp. 257–266.

[33] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06).* IEEE, 2006, pp. 423–432.

[34] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 167–178, 2006.

[35] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, 2007, pp. 402–412.

[36] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (prism)," *ACM SIGARCH computer architecture news*, vol. 40, no. 3, pp. 428–439, 2012.

[37] D. Sanchez and C. Kozyrakis, “Vantage: Scalable and efficient fine-grain cache partitioning,” in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 57–68.

[38] R. Wang and L. Chen, “Futility scaling: High-associativity cache partitioning,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 356–367.

[39] C.-J. Wu and M. Martonosi, “A comparison of capacity management schemes for shared cmp caches,” in *Proc. of the 7th Workshop on Duplicating, Deconstructing, and Debunking*, vol. 15. Citeseer, 2008, pp. 50–52.

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

[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 *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 104–117.

[42] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernández, “Dynamically controlled resource allocation in smt processors,” in *37th International Symposium on Microarchitecture (MICRO-37’04)*. IEEE, 2004, pp. 171–182.

[43] J. Sharkey, D. Balkan, and D. Ponomarev, “Adaptive reorder buffers for smt processors,” in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, 2006, pp. 244–253.

[44] S. Choi and D. Yeung, “Learning-based smt processor resource distribution via hill-climbing,” *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 239–251, 2006.

[45] Y. Zhou and D. Wentzlaff, “The sharing architecture: sub-core configurability for iaas clouds,” *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 559–574, 2014.

[46] Y. Ruan, V. S. Pai, E. Nahum, and J. M. Tracey, “Evaluating the impact of simultaneous multithreading on network servers using real hardware,” in *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2005, pp. 315–326.

[47] J. F. Couleur and E. L. Glaser, “Shared-access data processing system,” Nov. 19 1968, uS Patent 3,412,382.

[48] I. Yaniv and D. Tsafrir, “Hash, don’t cache (the page table),” *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1, pp. 337–350, 2016.

[49] A. Bhattacharjee and M. Martonosi, “Inter-core cooperative tlb for chip multiprocessors,” *ACM Sigplan Notices*, vol. 45, no. 3, pp. 359–370, 2010.

[50] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, “Accelerating two-dimensional page walks for virtualized systems,” in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 26–35.

[51] G. Vavouliotis, L. Alvarez, V. Karakostas, K. Nikas, N. Koziris, D. A. Jiménez, and M. Casas, “Exploiting page table locality for agile tlb prefetching,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 85–98.

[52] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, “Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb,” *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 469–480, 2017.

[53] A. Bhattacharjee, “Large-reach memory management unit caches,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 383–394.

[54] T. W. Barr, A. L. Cox, and S. Rixner, “Translation caching: skip, don’t walk (the page table),” *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 48–59, 2010.

[55] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, “Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1093–1108.

[56] A. Bhattacharjee, D. Lustig, and M. Martonosi, “Shared last-level tlbs for chip multiprocessors,” in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 62–63.

[57] T. W. Barr, A. L. Cox, and S. Rixner, “Spectlb: A mechanism for speculative address translation,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, pp. 307–318, 2011.

[58] S. Haria, M. D. Hill, and M. M. Swift, “Devirtualizing memory in heterogeneous systems,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 637–650.

[59] B. Pham, J. Veselỳ, G. H. Loh, and A. Bhattacharjee, “Large pages and lightweight memory management in virtualized environments: Can you have it both ways?” in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 1–12.

[60] D. Dhurjati and V. Adve, “Efficiently detecting all dangling pointer uses in production servers,” in *International Conference on Dependable Systems and Networks (DSN’06)*. IEEE, 2006, pp. 269–280.

[61] P. Cheng and G. E. Blelloch, “A parallel, real-time garbage collector,” in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001, pp. 125–136.

[62] M. Oskin and G. H. Loh, "A software-managed approach to die-stacked dram," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 188–200.

[63] N. Amit, "Optimizing the {TLB} shootdown algorithm with page access tracking," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 27–39.

[64] N. Amit, A. Tai, and M. Wei, "Don't shoot down tlb shootdowns!" in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–14.

[65] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh, "Avoiding tlb shootdowns through self-invalidating tlb entries," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 273–287.

[66] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 29–44.

[67] M. K. Kumar, S. Maass, S. Kashyap, J. Veselỳ, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "Latr: Lazy translation coherence," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 651–664.

[68] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "Unified instruction/translation/data (unitd) coherence: One protocol to rule them all," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 2010, pp. 1–12.

[69] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 340–349.

[70] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 955–972.

[71] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2421–2434.

[72] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 81–92.

[73] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2, p. 64, 2004.

[74] A. R. Alameldeen and D. A. Wood, "Ipc considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, pp. 8–17, 2006.

[75] T. Yu, P. Petoumenos, V. Janjic, H. Leather, and J. Thomson, "Colab: a collaborative multi-factor scheduler for asymmetric multicore processors," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 268–279.

[76] T. Li, D. Baumberger, and S. Hahn, "Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 65–74, 2009.

[77] I. Jibaja, T. Cao, S. M. Blackburn, and K. S. McKinley, "Portable performance on asymmetric multicore processors," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, 2016, pp. 24–35.

[78] J. C. Saez, A. Fedorova, D. Koufaty, and M. Prieto, "Leveraging core specialization via os scheduling to improve performance on asymmetric multicore systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 2, pp. 1–38, 2012.

[79] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 213–224, 2012.

[80] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, pp. 253–264, 2009.

[81] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Utility-based acceleration of multithreaded applications on asymmetric cmps," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 154–165, 2013.

[82] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007, pp. 1–11.

[83] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-isa heterogeneous multi-cores," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 2013, pp. 177–187.

[84] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[85] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[86] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS operating systems review*, vol. 44, no. 2, pp. 35–40, 2010.

[87] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a replicated logging system with apache kafka," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1654–1655, 2015.

[88] "H2 database engine," https://www.h2database.com Accessed: 2024-03-15.

[89] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications.* New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[90] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon *et al.*, "Renaissance: Benchmarking suite for parallel applications on the jvm," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 31–47.

[91] A. Kyrola, G. Blelloch, and C. Guestrin, "{GraphChi}:{Large-Scale} graph computation on just a {PC}," in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, 2012, pp. 31–46.

[92] "Open-source twitter finagle repository at github," https://github.com/twitter/finagle.

[93] "Open-source netty repository at github," https://github.com/netty/netty.

[94] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[95] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 31–34, 2004.

[96] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 318–319, 2003.

[97] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[98] A. Sabu, H. Patil, W. Heirman, and T. E. Carlson, "Looppoint: Checkpoint-driven sampled simulation for multi-threaded applications," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE, 2022, pp. 604–618.

[99] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "I-spy: Context-driven conditional instruction prefetching with coalescing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 2020, pp. 146–159.

[100] T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "Ripple: Profile-guided instruction cache replacement for data center applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 2021, pp. 734–747.

[101] "Intel xed," https://intelxed.github.io/ Accessed: 2024-03-29.