Universidad Zaragoza
1542

Departamento de Informática e Ingeniería de Sistemas
Instituto de Investigación en Ingeniería de Aragón
Grupo de Arquitectura de Computadores (gaZ)

Contributions to High-Performance Memory Hierarchies: Program Characterization, Resource Control, Transactional Synchronization, and Hardware Prefetching

by

Agustín Navarro Torres

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy
in the Universidad de Zaragoza

January 31, 2023

Supervisors Pablo Enrique Ibáñez Marín
Jesús Alastruey Benedé

Abstract

The increase in the number of cores and threads per processor over the last 15 years has allowed continuous improvements in system performance to be maintained. This design trend has involved major changes in the memory hierarchy. This dissertation explores new approaches to improve the performance of a multicore processor's memory hierarchy. Specifically, we analyze the utilization of its shared resources and propose mechanisms to improve the management of these resources from different levels ranging from the hardware to the application.

First, memory hierarchy performance has been evaluated for two SPEC CPU suites, CPU2006 and CPU2017, on an Intel Xeon Skylake-SP. This characterization has provided us with interesting findings, such as, for example, the unequal use of cache space by different applications or the effectiveness of hardware prefetching to reduce cache misses and improve system performance. This information served as a basis for defining new concrete objectives.

Next, we characterize the relationship between cache occupation, hardware prefetch and memory bandwidth consumption to understand their interactions. From this characterization work, we have proposed Balancer, a mechanism that dynamically imposes limits on LLC space usage and memory traffic to specific applications. These constraints improve performance and/or fairness in the execution of multiprogrammed workloads compared to an uncontrolled system. Balancer requires no hardware or operating system modifications.

As observed in the previous characterizations, data prefetching is a crucial technique as it allows hiding long-latency memory accesses and improving performance on modern high-performance processors. However, these prefetchers load a large number of useless blocks. This results in an unnecessary increase in the consumption of shared and scarce resources such as cache space and memory bandwidth. We propose Berti, a lightweight, highly accurate, energy-efficient, and high-performing local delta prefetcher that outperforms state-of-the-art prefetchers. Berti is an L1D prefetcher that orchestrates its requests across the entire cache hierarchy. Thanks to its high accuracy, Berti neither pollutes the caches nor wastes memory hierarchy bandwidth.

Synchronization between threads of the same application is another context where there can also be a high demand for shared resources in the memory hierarchy as the number of cores per processor increases. This dissertation presents a comprehensive study on the scalability of the different strategies that have been used for implementing synchronization solutions. The main conclusions that can be drawn are 1) hardware transactional memory scales better than fine-grained and non-blocking locks as the number of threads increases; 2) hardware transactional memory adoption is easy in real-world scientific applications and obtains performance comparable to that of a highly optimized locking scheme; and 3) enabling simultaneous multithreading for applications that access large memory blocks within their critical sections significantly affects the hardware transactional memory commit rate. In this context, we propose a novel cache replacement algorithm that aims to mitigate the negative effects of simultaneous multithreading on the transactional capacity abort rate.

Resumen

El aumento del número de núcleos e hilos por procesador en los últimos 15 años ha permitido mantener mejoras continuas en el rendimiento de los sistemas. Esta tendencia de diseño ha implicado importantes cambios en la jerarquía de memoria. Esta tesis explora nuevos enfoques para mejorar el rendimiento de la jerarquía de memoria de un procesador multinúcleo. En concreto, analizamos la utilización de sus recursos compartidos y proponemos mecanismos para mejorar la gestión de estos recursos en distintos niveles que van desde el hardware hasta la aplicación.

En primer lugar, se ha evaluado el rendimiento de la jerarquía de memoria para dos suites de SPEC, CPU2006 y CPU2017, en un Intel Xeon Skylake-SP. Esta caracterización nos ha proporcionado hallazgos interesantes, como, por ejemplo, el uso desigual del espacio de cache por parte de distintas aplicaciones o la eficacia de la prebúsqueda hardware para reducir los fallos de cache y mejorar el rendimiento del sistema. Esta información sirvió de base para definir nuevos objetivos concretos.

A continuación, caracterizamos la relación entre la ocupación de la cache, la prebúsqueda hardware y el consumo de ancho de banda con memoria para comprender sus interacciones. A partir de este trabajo de caracterización, hemos propuesto Balancer, un mecanismo que impone dinámicamente límites en el uso del espacio de la LLC y el tráfico con memoria a aplicaciones específicas. Estas restricciones mejoran el rendimiento y/o la equidad en la ejecución de cargas de trabajo multiprogramadas en comparación con un sistema no controlado. Balancer no requiere modificaciones en el hardware ni en el sistema operativo.

Como se ha observado en las caracterizaciones anteriores, la prebúsqueda de datos es una técnica crucial, ya que permite ocultar los accesos a memoria de larga latencia y mejorar el rendimiento en los procesadores modernos de alto rendimiento. Sin embargo, estos prebuscadores cargan un gran número de bloques inútiles. Esto se traduce en un aumento innecesario del consumo de recursos compartidos y escasos, como el espacio de cache y el ancho de banda con memoria. Proponemos Berti, un prebuscador hardware ligero, muy preciso, eficiente energéticamente y de alto rendimiento basado en deltas locales

que supera a los prebuscadores que conforman el estado del arte actual. Berti es un prebuscador de L1D que organiza sus peticiones a lo largo de toda la jerarquía de caches. Gracias a su gran precisión, Berti no contamina las caches ni desperdicia ancho de banda de la jerarquía con memoria.

La sincronización entre hilos de una misma aplicación es otro contexto en el que también puede haber una gran demanda de recursos compartidos en la jerarquía de memoria a medida que aumenta el número de núcleos por procesador. Esta tesis presenta un estudio exhaustivo sobre la escalabilidad de las diferentes estrategias que se han utilizado para implementar soluciones de sincronización. Las principales conclusiones que se pueden extraer son: 1) la memoria transaccional hardware escala mejor que los *fine-grain locks* y algoritmos *lock-free* a medida que aumenta el número de hilos; 2) la adopción de la memoria transaccional hardware es fácil en aplicaciones científicas y obtiene un rendimiento comparable al de un esquema de *fine-grain locks* altamente optimizado; y 3) habilitar el multihilo simultáneo para aplicaciones que acceden a grandes bloques de memoria dentro de sus secciones críticas afecta significativamente a la tasa de retiro de la memoria transaccional hardware. En este contexto, proponemos un novedoso algoritmo de reemplazo de cache que pretende mitigar los efectos negativos del multithreading simultáneo sobre la tasa de abortos por capacidad.

Acknowledgments

I would like to thank my official supervisors, Pablo and Chus, and my "*third*" supervisor, Victor, for their help and patience during all these years, this thesis would not have seen the light without their dedication and support. To my parents and my sister for their support during all these years.

To the GazBees (Adrián, Angélica, Carlos and Alba) for being there all these years and to the rest of the GAZ for welcoming me.

To Maria from Huawei Reseach Center Zürich, as well as Alberto Ros, Sawan and the rest of the research group at the University of Murcia for opening their doors and made me feel like home.

To Biswa for helping me to *prefetch* my first MICRO and guiding me in using ChampSim.

To my friends, especially Bea and Alberto, for being there during all these years. To Elba for her company at conferences.

Publications

Part of this dissertation includes results already published or accepted for publication.

The list of publications in chronological order are:

1. **Agustín Navarro-Torres**, Jesús Alastruey-Benedé, Pablo Ibáñez-Marín, Víctor Viñals-Yúfera. *Memory Hierarchy Characterization of SPEC CPU2006 and SPEC CPU2017 on the Intel Xeon Skylake-SP*. PLOS ONE 14(8): e0220135, 2019. https://doi.org/10.1371/journal.pone.0220135. JCR 2019: Q2

2. **Agustín Navarro-Torres**, Maria Carpen-Amarie, Jesús Alastruey-Benedé, Pablo Ibáñez-Marín. *Synchronization Strategies on Many-Core SMT Systems*. 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2021), October 26-29, 2021, Belo Horizonte, Brasil. CORE B

3. **Agustín Navarro-Torres**, J. Alastruey-Benedé, Pablo Ibáñez, and Víctor Viñals-Yúfera. *Berti: an Accurate Local-Delta Data Prefetcher*. 55th ACM/IEEE International Symposium on Microarchitecture (MICRO 2022), October 1-5, 2022, Chicago, Illinois, USA. GGS 2(A). HiPEAC Paper Award given by the HiPEAC steering committee. HiPEAC is the High-Performance and Embedded Architecture and Compilation Network of Excellence (EC Contract No. 287759, http://www.hipeac.net)

4. **Agustín Navarro-Torres**, J. Alastruey-Benedé, Pablo Ibáñez, and Víctor Viñals-Yúfera. *BALANCER: Bandwidth Allocation and Cache Partitioning for Multicore Processors*. Accepted for publication at The Journal of Supercomputing, 2023. JCR 2021: Q2.

Abbreviations

CMP	**C**hip **M**ulticore **P**rocessor
SLLC	**S**hared **L**ast **L**evel **C**ache
HTM	**H**ardware **T**ransactional **M**emory
STM	**S**oftware **T**ransactional **M**emory
SMT	**S**imultaneous **M**ulti**T**hreading
SKL-SP	**Sk**y**l**ake **S**calable **P**rocessor
CCD	**C**ompute **C**ore **D**ie
CCX	**C**ore **C**omple**X**
Intel-RDT	**Intel R**esources **D**irector ex**T**ensions
Intel-CAT	**Intel C**ache **A**llocation **T**echnology
Intel-MBA	**Intel M**emory **B**andwidth **A**llocation
TSX	**T**ransactional **S**ynchronization E**x**tensions
HLE	Intel **H**ardware **L**ock **E**lision
RTM	Intel **R**estricted **T**ransactional **M**emory
AMD QoSE	**AMD**64 Technology Platform **Q**uality of **S**ervice **E**xtensions
L2A	L2 **D**ata **C**ache Spatial Prefetcher
L2P	L2 **D**ata **C**ache Streamer **P**refetcher
CPU2006	SPEC **CPU 2006**
CPU2017	SPEC **CPU 2017**
FP	**F**loating **P**oint
CPI	**C**ycles **P**er **I**nstruction
APKI	**A**ccesses **P**er **K**ilo **I**instruction
HPKI	**H**its **P**er **K**ilo **I**instruction
DMPKI	**D**emand **M**isses **P**er **K**ilo **I**nstruction
MPKI	**M**isses **P**er **K**ilo **I**nstruction
rBW	**r**ead main memory **B**and**W**idth
CCO	**C**ontrol of SLLC **O**ccupancy
CMT	**C**ontrol of **M**emory **T**raffic
CAS	**C**ompare-**a**nd-**S**wap
MOPS	**M**illion **O**perations **P**er **S**econd
IP	**I**nstruction **P**ointer

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Rationale

Over the past 50 years, the industry has enjoyed a continuous improvement in processor performance fueled by Moore's Law [106] and Dennard Scaling [31]. During the early years, this performance improvement was coupled with increasing processor frequency. However, when the power dissipation limit was reached, processor designers reduced the processors' frequency and resorted to multiple cores integration on the same chip to maintain performance growth over time. Figure 1.1 shows how the number of transistors (yellow triangle) has grown exponentially over the years (X-axis), while frequency (blue circles) and power (red diamonds) grew exponentially until 2005 when they stabilized, and the number of cores (black squares) began to increase.

Today chip multicore processors (CMPs) integrate several tens of cores, sometimes grouped in clusters. Each core, that can run one or more threads, has private resources (e.g.: the pipeline) and resources that are shared with other cores e.g.: a major part of the memory hierarchy. The memory hierarchy is a common solution for hiding part of the costly off-chip main memory access latency and avoiding the power consumption associated with its use. It is organized into levels, which increase in size, energy consumption and access latency. Usually, the first cache levels, those closest to the core, are private, while the last level of cache (LLC[1]) is a shared resource. Likewise, the main memory access channels and the interconnection network that communicates cores, SLLC banks and memory channels are also shared resources.

This paradigm shift of processors, from a single fast core to many cores that share resources, has driven major changes in their design. Researchers and engineers do not have to achieve only the highest performance in an individual application, they must also improve the performance of the system as a whole and prevent applications from harming each other. Thus, the memory hierarchy, being the main shared resource, requires special attention in this context.

[1] In this dissertation LLC and SLLC are used indistinctly.

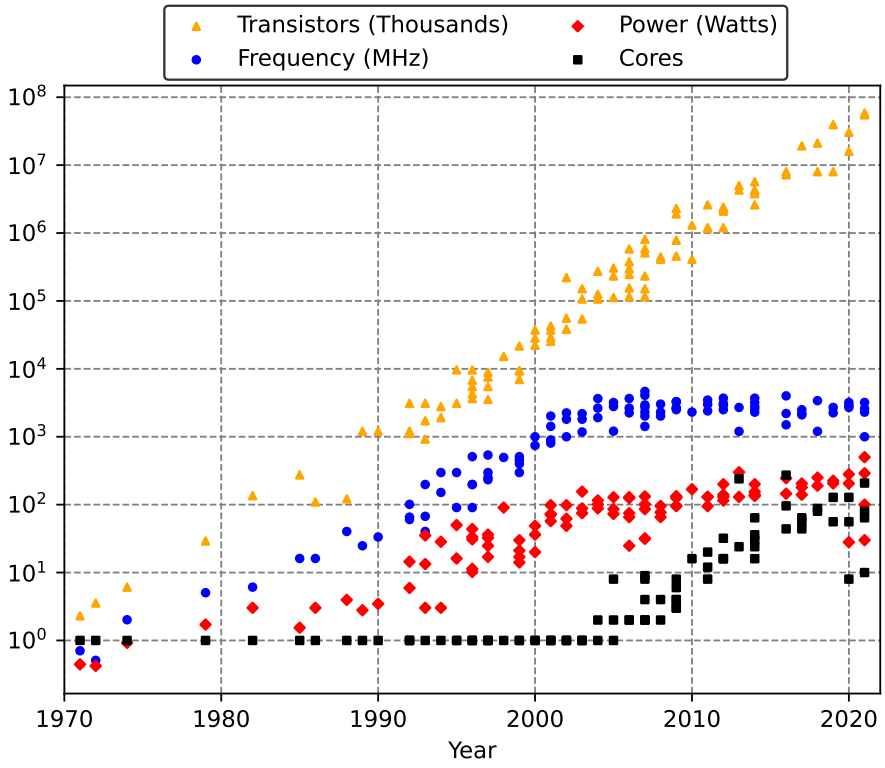


Figure 1.1: 50 years of microprocessor trend data [142].

Memory hierarchy design constantly demands new architectural solutions as new trade-offs arise due to the significant increase in the number of cores on the chip, and the increased pressure on the hierarchy that comes with it.

1.2 Contributions

The main objective of this thesis is to improve the efficiency in the use of shared resources in the memory hierarchy of a CMP. This goal can be achieved from different approaches, and by acting at different levels ranging from hardware to application. The contributions of the thesis attempt to improve our understanding of how applications use resources and to improve the way they use them. These contributions are four: 1) characterization of application use of shared resources such as LLC space and memory bandwidth, 2) shared resource control mechanisms in the memory hierarchy, 3) resource-efficient hardware data prefetcher, and 4) study of the use of shared resources by different synchronization mechanisms.

This section motivates the research lines, details the contributions, and

includes the results of the work.

1.2.1 Characterization of Application Use of Shared Resources

The experimental part of most research work in computer architecture consists of feeding a simulator or a real machine with programs representative of current or future software. There are a multitude of collections of such programs, each associated with a particular field of application [47, 45, 43, 120]. Understanding the behavior of each of these programs facilitates all experimentation tasks. It allows for example to select suitable samples to demonstrate the feasibility of an idea, or to analyze the results obtained when applying a technique taking into account the characteristics of the executed software. Consequently, application characterization is one of the first tasks of many research works and even the target of specific publications recurrently in the computer architecture community [87, 97, 15, 167].

At the beginning of the work of this thesis, several events occur that make the specific characterization work even more relevant: i) SPEC releases a new suite for general purpose computing, CPU2017, after eleven years since the previous version, CPU2006, ii) some vendors such as Intel significantly improve the tools for both information retrieval (hardware counters) and resource control (cache partitioning, prefetcher disconnection), and iii) Intel adopts for the first time the content management model of its LLC mostly in exclusion. Thus, the first objective raised in the thesis is the characterization of the new set of SPEC CPU2017 programs in the new Intel Xeon Skylake-SP memory hierarchy, and its comparison with the previous version, SPEC CPU2006. This characterization will provide information to define new concrete objectives in the rest of the work, and will facilitate all the experimentation work.

This characterization work is presented in **Chapter 2**. We evaluate the memory hierarchy behavior of the CPU2006 and CPU2017 SPEC single-thread benchmarks using different shared LLC sizes and prefetch configurations, on an Intel Xeon Skylake-SP. The evaluation is performed on a real machine, running the complete applications with all their reference inputs and using hardware counters to take the measures. This work has been published in *Plos One (2019)* [114]. All code and data are available at https://github.com/agusnt/Xeon-SP_Memory_Characterization_SPEC-CPU-2K6-2K17/.

1.2.2 Shared Resource Control Mechanisms in the Memory Hierarchy

Advances in the scale of integration allow processors to have more and more cores. This trend means that shared resources, especially space in the LLC and memory bandwidth, are under increasing pressure. On the other hand, the execution of workloads composed of independent applications generates uneven resource demands. For example, some programs may require a lot of

cache space while others may consume a lot of memory bandwidth. Shared resource management is key to efficiently running such workloads on today's processors. In this regard, many proposals have been published with hardware mechanisms to share SLLC space between cores or the applications they run. Most of these proposals require hardware modifications and have been evaluated with a reduced number of cores.

Recently, several commercial processors have incorporated technologies that allow 1) monitoring the use of cache space and memory bandwidth by different cores, and 2) imposing limits on the consumption of these shared resources. For example, starting with the Skylake family, Intel processors have Intel Cache Allocation Technology (CAT) [78], AMD EPYC processors have AMD64 Technology Platform Quality of Service Extensions [69]. These functionalities allow the design of software mechanisms that implement shared resource management policies and require no hardware or operating system modifications for their operation.

Since hardware support for memory traffic management is more recent, works for controlling this resource are scarcer. Specifically, we are aware of only three proposals that address memory bandwidth sharing [129, 171, 130]. Moreover, almost all previous works focus on Intel's memory hierarchies, where the LLC is shared among all cores, so there are no proposals specifically designed for a clustered LLC organization, such as the one used by AMD in its current EPYC processors.

The goal of this thesis in this line of research is to design a mechanism that improves system performance and fairness by dynamically sharing LLC cache space and main memory bandwidth among running applications. To this end, **Chapter 3** presents a detailed characterization of the shared cache and memory bandwidth usage of an AMD Rome processor running multiprogrammed workloads. Then, it proposes Balancer, a set of mechanisms that control the use of these shared resources to improve system performance and fairness. Balancer requires no hardware or operating system modifications.

Balancer has been designed and evaluated in a clustered LLC such as that of the AMD Rome. Hence, it can make different decisions in each cluster in a decentralized manner, in response to their particular cache utilization and bandwidth consumption.

This work has been accepted for publication in the *Journal of Supercomputing (2023)*. All code and data are available at https://github.com/agusnt/BALANCER.

1.2.3 Resource-efficient Hardware Data Prefetcher

Prefetching has proven to be a powerful mechanism for reducing the memory access penalty. Hardware prefetchers learn memory access patterns and use those patterns to load data into the cache hierarchy before the processor requests it. In this way, future processor accesses to memory will fetch the data from

nearby caches, with low latency. Prefetching techniques can be employed in the first-level private data cache (L1D), second-level cache (L2) or last-level shared cache (SLLC). Recently proposed prefetchers target both the L2 cache (Bingo [9]) and the L1 data cache (IPCP [122]). A common problem with all of them is their low accuracy. Often, the prefetchers make mistakes and load blocks that will never be used, causing pollution in the caches and extra consumption of memory bandwidth. This problem becomes more relevant as the number of cores increases as it causes an inefficient use of increasingly saturated shared resources. Ultimately, this inappropriate use results in a decrease in overall system performance and an increase in energy consumption.

Several mechanisms have been proposed to reduce the polution of prefetchers. For example, the Perceptron-based prefetch filter (PPF) is a filter that decides whether prefetched blocks are loaded in L2 or not [13]. Similar to PPF, there are proposals that control the aggressiveness of prefetchers by controlling their prefetch degree and prefetch distance, or decide whether to prefetch in L2 or in LLC [6, 42, 66, 125, 123, 124, 155]. These techniques achieve partially improved accuracy but entail additional storage.

The goal of this thesis with respect to prefetching is to propose a prefetch mechanism with very high accuracy, so that it minimizes the wasteful consumption of shared multicore processor resources such as shared cache space and bandwidth with main memory. Berti is proposed in **Chapter 4**, a new first-level data prefetcher that orchestrates prefetch requests to the memory hierarchy. Berti learns the best deltas per IP by a mechanism that calculates the coverage of each delta and selects those deltas that provide the highest coverage. Berti pushes further the limits of hardware prefetchers with minimal space storage requirements and high accuracy. This work has been published in the *IEEE/ACM International Symposium on Microarchitecture (2022)* [113]. All code and data are available at the artifact which received three badges at MICRO 2022: Artifacts Available, Artifacts Evaluated—Functional, Results Reproduced) https://github.com/agusnt/Berti-Artifact.

1.2.4 Study of the Use of Shared Resources by Different Synchronization Mechanisms

Previous contributions in this thesis are focused on improving the performance of systems running multiprogrammed workloads, i.e, a set of independent programs running simultaneously in the same computer. However, it is also important to improve the execution of parallel workloads, i.e., those composed of simultaneously running threads that collaborate to solve the same problem.

The stagnation in single-threaded performance improvement and its resulting shift toward multithreaded execution has changed how applications are developed. An application that wishes to take full advantage of the potential of a multicore processor should use as many threads as possible. This implies the use of several execution threads per core, or simultaneous multithreading

(*SMT*), which increases the pressure on the private and shared resource of the memory hierarchy.

The use of a larger number of threads implies more concurrency, making synchronization mechanisms more important. The use of classical mechanisms such as fine-grain locks or lock-free mechanisms are complex, require in-depth knowledge of the program, and are error-prone. Transactional memory was developed to make this synchronization easier and improve its performance. In transactional memory, data conflicts are detected and handled by the memory hierarchy, making content management within the memory hierarchy even more important. With the appearance of the first hardware transactional memory implementations on commercial processors [98, 162, 83], numerous mechanisms have appeared that evaluate and compare their performance with classical synchronization mechanisms [18, 128, 143, 176].

However, to the best of our knowledge, there is no study that evaluates the scalability of all synchronization mechanisms together on many-core systems and that takes into account the impact of simultaneous multithreading (SMT). Furthermore, while SMT is recognized as a limiting factor for HTM performance [164], its impact is not clearly quantified and no solution has been proposed to mitigate it.

Chapter 5 focuses on evaluating the performance and operation latency of the most widely-used synchronization mechanisms: hardware transactional memory (HTM), software transactional memory (SMT), locks and lock-free. This chapter also study the impact of simultaneous multithreading (SMT) technology on HTM performance and propose a mechanism to reduce SMT-inducted aborts. This work has been published in the *IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD (2021)* [115]. All code and data are available at https://github.com/agusnt/Synchronization-Strategies-on-Many-Core-SMT-Systems.

1.3 Dissertation Overview

This dissertation is organized as follows. **Chapter 2** characterizes SPEC CPU2006 and CPU2017 and its interaction with the Intel Xeon Skylake-SP memory hierarchy. In **Chapter 3** presents Balancer a software mechanism that improve the system's performance and fairness. Then **Chapter 4** presents Berti our proposal for a new L1D prefetcher.**Chapter 5** shows an evaluation of different thread synchronization mechanisms in terms of performance an operation latency. Finally in **Chapter 6** concludes the dissertation summarizing the work done and discusses about the new research lines.

A variety of methodologies and tools have been used during the completion of this doctoral thesis. In Chapter 2, a characterization work has been performed on a real system equipped with an Intel Skylake processor using hardware tools for resource monitoring and control (counters, model-specific registers, Intel CAT). Chapter 3 presents an experimentation and evaluation work on

a real system, this time on a computer with an AMD EPYC processor using the hardware support of the platform (hardware counters, AMD QoSE). In Chapter 4 the proposal has been evaluated on a processor modeled with the ChampSim simulator. Chapter 5 includes experiments with real hardware (hardware transactional memory on Intel Cascade Lake) and simulation (Gem5). Therefore, the experimental environment and methodology are presented in each chapter since there is no common framework.

1.4 Thesis project framework

This thesis has been developed at the Grupo de Arquitectura de Computadores de la Universidad de Zaragoza (gaZ), in the Departamento de Informática e Ingeniería de Sistemas (DIIS) and Instituto de Investigación en Ingeniería de Aragón (I3A).

The thesis and its research work has been funded by:

- Grant (BES-2017-079790) para la Formación de Personal Investigador (FPI) from the Spanish Ministry of Economy and Competitive.
- Project PID2019-105660RB-C21: Jerarquía de memoria, gestión de tareas y optimización de aplicaciones, from the Agencia Estatal de Investigación and TIN2016-76635-C2-1-R: Arquitectura y programación de computadores escalables de alto rendimiento y bajo consumo, from the Spanish Ministry of Economy and Competitive. Both projects are in collaboration with the University of Cantabria which led to a fruitful collaboration in this thesis.
- The Aragón Government has partially funded the work through the Research group recognition: T58_20R research group from Aragón Government and European Social Fund, and (3) 2014-2020 "Construyendo Europa desde Aragón" from European Regional Development Fund.

I have developed two research internships. One of them at the Huawei research lab in Zürich (Switzerland) under the supervision of Maria Carpen-Amarie, funded by Huawei, and a second one at Grupo de Arquitectura de Computadores y Sistemas Paralelos (CAPS) of the Universidad de Murcia under the supervision of Alberto Ros Bardisa, thanks to a competitive grant for short stays associated to the FPI.

Chapter 2

Memory Hierarchy Characterization of SPEC CPU2006 and CPU2017

SPEC CPU is one of the most common benchmark suites used in computer architecture research. In this chapter we present a detailed evaluation of the memory hierarchy performance for the CPU2006 and single-threaded CPU2017 benchmarks. The experiments were executed on an Intel Xeon Skylake-SP, which is the first Intel processor to implement a mostly non-inclusive SLLC. First, we present a classification of the benchmarks according to their memory pressure and analyze the performance impact of different SLLC sizes. Then, we test all the hardware prefetchers showing that they improve performance in most of the benchmarks. After a comprehensive experimentation, we can highlight the following conclusions: i) almost half of SPEC CPU benchmarks have very low miss ratios in the second and third level caches, even with small SLLC sizes and without hardware prefetching, ii) overall, the SPEC CPU2017 benchmarks demand even less memory hierarchy resources than the SPEC CPU2006 ones, iii) hardware prefetching is very effective in reducing SLLC misses for most benchmarks, even with the smallest SLLC size, iv) the SLLC utilization is uneven among applications, and v) from the memory hierarchy standpoint the methodologies commonly used to select benchmarks or simulation points do not guarantee representative workloads.

2.1 Introduction

The majority of experimental research in computer architecture is based on feeding a simulator or a real machine with benchmarks that are representative of current or future software in a certain application field. The benchmark characterization is one of the first tasks to be carried out by the computer architecture community. Among its goals, we can highlight the classification of applications according to certain characteristics, the selection of samples for simulation, or the detection of non-optimal behaviors to detect flaws in the designs and improve it. These benchmarks suites can be developed by diverse corporations

(e.g. SPEC [120]), research groups (e.g. CloudSuite [47]), communities (e.g. TACLeBench [45]) and even certain companies (e.g. EEMBC [43]) propose benchmark suites composed of a number of applications focused on specific fields such as general purpose computing, cloud computing, real time or embedded processing, respectively. According to SPEC Corporation, SPEC CPU2017 contains a collection of next-generation and industry-standardized benchmarks aimed at stressing the processor, memory subsystem and compiler [119].

Implementing a new hardware concept in a real system is unfeasible in most cases, due to its high cost or the impossibility of its subsequent modification. An alternative is to use a simulator which models at the desired level of detail (e.g. cycle-level) the behavior of a complex system such as a multicore processor with a multi-level memory hierarchy and an interconnection network. However, the complete execution of a benchmark in these simulators may require months or even years. Thus, sampling techniques are used to identify small sections of a benchmark that approximate the behavior of the full application [165, 131].

Benchmarks with certain characteristics are selected to evaluate the performance of new proposals. For example, research on shared cache replacement algorithms frequently selects a mix of benchmarks with different degrees of pressure on the memory hierarchy: some of them show high cache utilization while others do the opposite [86, 2].

In this chapter, we characterize the interaction of the SPEC CPU2006 and CPU2017 suites with the Intel Xeon Skylake-SP memory hierarchy. The analysis of CPU2017 is of special interest since it is a recent suite [126]. Regarding the processors, it also brings relevance to this study because the Intel Xeon Skylake-SP family has been released in July 2017 and incorporates significant changes in the memory hierarchy: the private L2 cache size has quadrupled and the SLLC, unlike all previous Intel processors, has been designed following a mostly non-inclusive policy. AMD chose similar policies since its inception, namely strict exclusion between the private cache levels, and mostly-exclusion between private cache levels and the SLLC. So we think non-inclusive content policies seem to be a consolidating trend worth focusing.

The remaining of this chapter is organized as follow. Section 2.2 describes the state of the art. Section 2.3 explains the hardware framework used in this chapter. Section 2 presents the detailed characterization. And section 2.5 summarizes the conclusion remarks.

2.2 State of the Art

The characterization of a new benchmark suite is a recurrent research activity in the computer architecture community. In this section, we present the state of the art in two areas: benchmark characterization and selection of simulation intervals.

2.2.1 Benchmark Characterization Methodologies

Benchmark characterization may be carried out through simulation or by using hardware counters. On the one hand, simulation provides a flexible experimentation framework that allows the evaluation of different memory hierarchy configurations, such as cache sizes or replacement policies. Unfortunately, hardware components in recent processors, such as prefetchers, can not be accurately modelled since their implementation details are not fully disclosed. Moreover, a complete benchmark simulation may require weeks or months, so studies based on simulation typically characterize only a small section of the selected benchmarks.

On the other hand, real execution is able to capture the behavior of state-of-the-art hardware components. Performance monitoring support included in commercial systems collects execution events that can be used to obtain metrics that characterize benchmark behavior during their real execution with a very low overhead. However, using real execution makes it difficult to perform design space exploration, since hardware configuration capabilities are limited.

There are many papers devoted to the SPEC CPU2006 characterization. For instance, Jaleel et al. characterize the behavior of the suite with different cache sizes on a simulator [87], Korn et al. study its performance according to page size [97].

Regarding SPEC CPU2017, we have only found two characterization studies: Limaye et al. [100] and Panda et al. [126]. They analyze the behavior of benchmarks on an Intel processor from the Haswell family. Hardware counters are used to collect the amount and type of executed instructions, memory footprint, and cache misses at all levels of the memory hierarchy. Both papers end up presenting a methodology to classify benchmarks. Regarding the characterization of the use of the memory hierarchy, they show some limitations. Namely, characterization is performed on old systems, local miss ratios are used as performance metric instead of MPKI, and the sensitivity to cache size or hardware prefetching is not studied.

With respect to the content management in shared LLCs, many processors use an inclusive policy (LLC content is a superset of all private caches), however, using instead a mostly non-inclusive policy (LLC acts as a victim cache which may, or may not, evict cache lines on hits) seems to gain momentum through more elaborated coherence protocols. AMD calls the same policy "mostly-exclusive", and started using it in its first processor with shared LLC, the 2007 4-core Opteron Barcelona [27]. All the following AMD processors, such as the 6-core Istanbul (2009), the 12-core Magny Cours (2010), the 16-core Bulldozer (2011) or, recently the 4-core Zen Core Complex (2016) evolved in cache sizes, coherency protocols and core features, but all have maintained the same mostly-exclusive contents policy. So we think characterizing benchmarks through the Intel Skylake-SP fits well with this trend.

In this work, hardware counters have been used to obtain metrics about the

execution of the benchmarks on a Skylake-SP processor. Intel's Model Specific Registers (MSR) allow us to independently enable or disable the different hardware prefetchers. The *Intel Cache Allocation Technology (CAT)* allows us to vary the LLC space occupied by an application modifying its number of allocated ways. In this way, we characterize the behavior of the entire benchmark with different hardware configurations of the memory hierarchy.

2.2.2 Selection of Benchmarks and Simulation Intervals

SPEC CPU2006 and CPU2017 are composed of several applications, some of them with different inputs, resulting in multiple application-input combinations. We define *benchmark* as an application-input pair. For example, there are 29 applications and 55 benchmarks (application-input pairs) in the CPU2006 suite. The execution time of a complete benchmark on a simulator may last weeks or even months, which makes the simulation of a suite unfeasible. To reduce this time, a two-level sampling can be carried out. First, a subset of benchmarks is selected. Second, one or more fragments of the complete execution representing the overall behavior are chosen as simulation intervals. In the literature, several successful sampling techniques have been proposed, such as *Hierarchical Clustering* [100] for benchmark selection, and *SimFlex* [165] or *SimPoint* [131] for intervals selection.

Hierarchical Clustering [100]. The *Hierarchical Clustering* methodology is applied in three steps: i) execution of all benchmarks to obtain 20 metrics through hardware counters, ii) analysis of the main components to reduce the number of metrics to 4, and iii) clustering of similar benchmarks.

In the first step, the authors select microarchitecture-independent metrics which are related to the type of instructions executed and their proportions. Therefore, this methodology does not consider the behavior of the memory hierarchy as a parameter to guide sampling.

SimFlex [165]. The *SimFlex* methodology uses statistical sampling theory to select simulation intervals. It identifies numerous small-sized intervals that are distributed throughout all the application execution, ensuring that they are representative of the benchmark.

However, *SimFlex* has an important drawback when it is applied to cache memory hierarchy research: the simulation intervals do not have enough extension to provide accurate data without a previous cache warm-up. Warming memory structures has an unacceptable overhead when simulating large caches or a large number of intervals.

SimPoint [131]. *SimPoint* is one of the most used methodologies to select simulation intervals. First, it splits up the execution of a benchmark into intervals of equal number of instructions. For each interval, *Simpoint* calculates a signature that contains the number of executions of each basic block. Then, *SimPoint* executes the *k-Means* algorithm to group different intervals into clusters called phases. The intervals of a given phase execute similar code and

therefore are expected to exhibit a similar behavior in the system (misses in the memory hierarchy, CPI ...). Finally, the centroid is selected as the most representative interval of each phase.

Although *SimPoint* offers several simulation intervals for each benchmark, most research related to memory hierarchy design uses only the most representative interval.

One of the contributions of this chapter is to assess the representativeness of the intervals selected by *SimPoint* regarding the interaction with the memory hierarchy. Towards that end we analyze the temporal evolution of different metrics, namely CPI, MPKI2 and MPKI3, across the whole application execution. By plotting such metrics as a function of time and superposing the first three intervals selected by *SimPoint*, we will see how well they match the memory hierarchy dynamics.

2.3 Experimental Framework

2.3.1 Intel SKL-SP Memory Hierarchy

Intel launched in 2017 its new family of processors targeting high-performance servers, the Intel Xeon Skylake-SP (Skylake Scalable Performance, SKL-SP for short). Specifically we use an *Intel Xeon Gold 5120*. The processor integrates 14 cores. Each core contains a split first level cache (32 KiB for instructions and 32 KiB for data) with two data hardware prefetchers, a 1 MiB unified second level cache with 16 ways, with another two hardware prefetchers. All cores share a 19.25 MiB mostly non-inclusive SLLC with 11 ways that works as a victim cache. On a demand miss, a demand memory or L2 prefetch request fills only the private levels, but not the SLLC. When a block is evicted from L2, a reuse filter decides whether the block is filled into SLLC or not. In all of its previous cache organizations, Intel used an inclusion policy, which enforces that all the private caches (L1 and L2) content of all the cores are also stored in the SLLC. Inclusion leads to designs with relatively small private caches and a large SLLC. By contrast, in non-inclusive hierarchies, the SLLC content is largely independent of the private caches content. Changing the policy from inclusive to mostly non-inclusive design has allowed Intel to redistribute the cache chip area, by enlarging private caches (from 256 KiB to 1 MiB) and diminishing the SLLC (from 2.5 MiB to 1.375 MiB per core). However, since now SLLC stores less replicated content, reducing its size does not necessarily imply lowering its effective capacity. According to D. Kanter, *"the new cache design reduces the processor, reducing L2 miss rate by about 40% on average for the SPECint_rate2006 suite, whereas the L3 miss rate barely increases"* [28].

SKL-SP processors have four hardware prefetchers associated with the first and second cache levels which can be selectively enabled or disabled by the user through the 0x1A4 *Model Specific Register* (MSR) [77].

2.3.2 Runtime Environment

In order to ensure the reproducibility of the experiments each execution thread is pinned to the first core and TurboBoost has been disabled.

The system runs a CentOS 7 Linux with the 3.10 kernel. The system specifications are shown in Table 2.1.

Table 2.1: SKL-SP system specifications.

Processor	Intel Xeon Gold 5120 (Skylake-SP)
Cores × Threads	14 × 2
L1 I-Cache	32 KiB, 64 B line size, 8 ways
L1 D-Cache	32 KiB, 64 B line size, 8 ways
L2C	1 MiB, 64 B line size, 16 ways
LLC	19.25 MiB, 64 B line size, 11 ways
Main Memory	96 GiB DDR4 Nominal peak BW: 115.2 GB/s
TurboBoost	Disabled
Hyperthreading	Disabled (1 thread/core)
OS	CentOS 7, kernel: 3.10

2.3.3 Control Tools

We use *Intel Cache Allocation Technology* (Intel-CAT for short) [78], a tool included in the *Intel Resource Director*, to limit the amount of SLLC, from 1.375 MiB (1 way) to 19.25 MiB (11 ways) in steps of 1.375 MiB (1 way) available for each thread. The available space is configured by writing a binary mask in a per-thread MSR register. Each bit of the mask represents a way, a fraction of the space, of SLLC that can be used. Therefore, changing the SLLC size implies a change in SLLC associativity for that thread. Several threads can use the same fraction, which implies a competitive sharing of the same subset of SLLC.

2.3.4 Monitoring and Instrumentation Tools

We use hardware counters to analyze the behavior and interaction of the applications executed on a real system. Hardware counters allow us to record events that occur during the execution of an application. For example, we can measure retired instructions, elapsed cycles, or misses experienced in the SLLC. The use of hardware counters allows us to analyze applications on a real systems. In Figure 2.1 we outline the relevant hardware components and raw measures we use in our methodology.

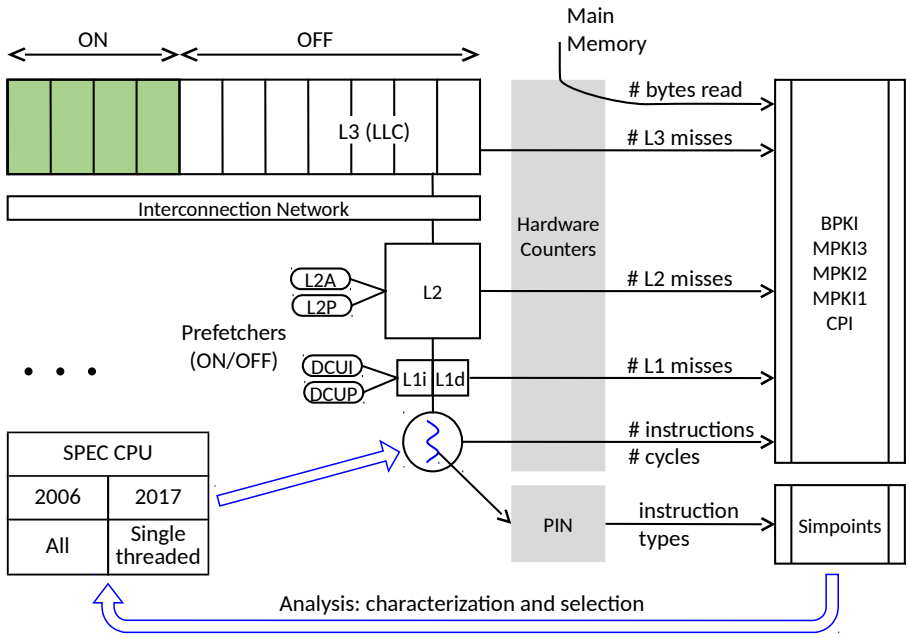


Figure 2.1: Outline of the methodology.

All measures of hardware events have been taken from hardware counters driven by `perf`, a GNU/Linux profiler tool [30]. Table 2.2 shows the correspondence between monitored hardware counters and the symbolic events of `perf`.

Table 2.2: `Perf` symbolic event types used.

	Symbolic event types
Cycles	cycles
Retired Instructions	instructions
L1D Demand Misses	r08d1
L2 Demand Load Misses	LLC-load
L2 Demand Store Misses	LLC-store
LLC Demand Load Misses	LLC-load-misses
LLC Demand Store Misses	LLC-store-misses
Memory Read Access	uncore_imc_{0-5}/event=0x4,umask=0xC/
Memory Write Access	uncore_imc_{0-5}/event=0x4,umask=0x3/

To measure the evolution of an event with respect to the number of occurrences of another event over time (e.g.: a graph with Y-axis the number of failures in SLLC and X-axis the number of instructions) we have developed

`perf++` [112]. `Perf++` is a new tool that allows us to measure the occurrences of an event every x occurrences of another event.

Moreover, we also resort to the instrumentation tool PIN to analyze instruction operation codes [103], and feed them into the simpoint package to obtain representative simulation intervals according to that proposal [131].

2.3.5 Workloads

In this chapter we use the SPEC CPU2006 [118] (CPU2006 for short) and single-thread CPU2017 [119] (CPU2017 for short) benchmark suites. SPEC CPU benchmark suites are one of the most widely used suites in the computer architecture research. SPEC CPU2017 benchmarks was released in 2017 in order to replace the previous version (CPU2006) According to SPEC Corporation: *"The SPEC CPU 2017 benchmark package contains SPEC's next-generation, industry-standardized, CPU intensive suites for measuring and comparing compute intensive performance, stressing a system's processor, memory subsystem and compiler"* [119].

Both suites have been compiled following the official documentation provided by SPEC [153, 154]. CPU2006 has been compiled with `gcc 4.9.2` and the options `-O3 -fno-strict-aliasing`. CPU2017 has been compiled with `gcc 6.3.1` and the base flags. `-DBIG_MEMORY` has been used for `deepsjeng` and `-m64` when required.

We do not consider the multi-threaded applications, CPU2017 speed versions of `xz` and the floating point (fp), because their characterization requires a different methodology. Therefore, all CPU2006 and single-threaded CPU2017 applications have been executed with all the "reference" inputs (one or more input data sets representative of real behavior).

Table 2.3 shows the 106 benchmarks tested across the two suites, from a total of 43 applications, 17 integer and 26 floating point. Some applications appear only in one suite, as `astar` (SPEC CPU2006 int) or `blender` (SPEC CPU2017 fp), while some others have evolved and are in both suites, as `gcc`. Moreover, some CPU2017 applications have both speed and rate versions. For instance, the integer application `mcf` has one CPU2006 version and two CPU2017 versions, one to produce the SPECrate metric (_r), and the other one the SPECspeed metric (_s). For each benchmark, the table specifies an input identifier (#), the input name (Input) and the measured instruction count (Inst.).

As a general rule, we can see a significant increase of individual instruction counts in the CPU2017 benchmarks with respect to the CPU2006 ones, even though if we take into account the aggregate figures the difference flattens.

2.3.6 Metrics

The most relevant metric for measuring the performance of an application running on a system is execution time. Any other metric helps to understand

Table 2.3: Benchmarks tested, divided between integer (int column) and floating point (fp column). Filled cells in columns “2006" and “2017" mean the benchmark appears in the corresponding suite. The columns labeled “_r” and “_s” refer to the application versions producing SPECrate and SPECspeed metrics, respectively. Columns “Inst." and "#" show instruction count (x10^{12}) and input identifier, respectively.

2006			int	2017			
Inst.	*Input*	#	*Name*	#	*Input*	*Inst.*	
						_r	_s
0.36	BigLakes	1	**astar**				
0.75	rivers	2					
0.41	source	1	**bzip2**				
0.17	chicken.jpg	2					
0.29	liberty.jpg	3					
0.53	program	4					
0.58	text.html	5					
0.33	combined	6					
			deepsjeng	1	ref	1.87	2.18
			exchange2	1	6	2.91	2.91
0.07	166	1	**gcc**	1	pp.c -O3	0.20	
0.14	200.00	2		2	pp.c -O2	0.23	
0.12	c-typeck	3		3	small.c -O3	0.23	
0.09	cp-decl	4		4	ref32.c -O5	0.19	
0.10	expr	5		5	ref32.c -O3	0.26	
0.14	expr2	6		1	-fipa-pta		1.21
0.17	g23	7		2	-fin=1000		0.52
0.15	s04	8		3	-fin=24000		0.50
0.05	scilab	9					
0.02	13x13	1	**gobmk**				
0.06	nngs	2					
0.03	score2	3					
0.02	trevorc	4					
0.03	trevord	5					
0.50	baseline	1	**h264ref**				
0.32	main	2					
2.89	sss_main	3					
0.86	nph3	1	**hmmer**				
1.82	retro	2					
			leela	1	ref	2.11	2.11
1.65	1397	1	**libquantum**				
0.32	inp	1	**mcf**	1	inp	0.92	1.65
0.54	omnetpp	1	**omnetpp**	1	General	1.09	1.06
1.05	checkspam	1	**perlbench**	1	checkspam	1.22	1.22
0.36	diffmail	2		2	diffmail	0.70	0.70
0.66	splitmail	3		3	splitmail	0.67	0.67
2.26	ref	1	**sjeng**				
			x264	1	-pass 1	0.52	0.52
				2	-pass 2	1.96	1.96
				3	-seek 500	1.99	1.99
0.99	t5	1	**xalancbmk**	1	t5	1.27	1.27
			xz	1	cld	0.40	
				2	cpu2006	1.04	
				3	combined	0.57	

Total
18.84

AVG
0.54

Total	**Total**
20.34	20.46

AVG	**AVG**
1.02	1.36

2006			fp	2017		
Inst.	*Input*	#	*Name*	#	*Input*	*Inst.*
						_r
			blender	1	sh3	1.73
2.56	*None*	1	**bwaves**	1	bwaves_1	1.30
				2	bwaves_2	1.57
				3	bwaves_3	1.40
				4	bwaves_4	1.83
			cactuBSSN	1	spec_ref	1.11
2.73	benchADM	1	**cactusADM**			
4.25	hypervis	1	**calculix**			
			cam4	1	*None*	2.69
1.65	23	1	**dealII**			
			fotonik3d	1	*None*	1.95
0.11	cytosine	1	**gamess**			
0.09	h2ocu2+	2				
0.37	triazolium	3				
1.72	*None*	1	**gemsFDTD**			
1.95	gromacs	1	**gromacs**			
			imagick	1	refrate	4.59
1.24	reference	1	**lbm**	1	reference	1.28
0.18	leslie3d	1	**leslie3d**			
0.10	su3imp	1	**milc**			
			nab	1	1am0	2.09
2.28	namd	1	**namd**	1	apoa1	1.78
			parest	1	ref	3.39
0.94	SPEC-ref	1	**povray**	1	SPEC-ref	3.31
			roms	1	ocean2	2.71
0.34	pds-50	1	**soplex**			
0.35	ref	2				
3.39	ctlfile	1	**sphinx3**			
3.44	*None*	1	**tonto**			
2.93	*None*	1	**wrf**	1	*None*	4.20
1.92	*None*	1	**zeusmp**			

Total
32.51

AVG
1.63

Total
36.91

AVG
2.31

the execution time and the influence of some subsystem on it. Therefore, a performance metric is only useful if it gives insight into execution time variations.

When measuring performance of an application running on a system, the ultimate metric is execution time. Any other metric is intended to help understand why an execution time is obtained or to show the influence of some subsystem on it. Therefore, a performance metric is only useful if it gives insight into execution time variations. Since we are interested in characterizing the memory hierarchy as a performance enabler or limiter, we measure miss ratios and memory bandwidth consumption in addition to execution time. Specifi-

cally, we consider number of cycles per instruction (CPI), number of demand misses per thousand instructions in the different cache levels (demand misses per kilo instruction, DMPKI), and number of bytes read and written from/to main memory per thousand instructions (bytes per kilo instruction, BPKI), as performance metrics. BPKI accounts for both blocks fetched on demand and blocks fetched by prefetch, and it can be used to show the extra traffic induced by the hardware prefetchers, if we compare the same application running with and without hardware prefetching. Table 2.4 summarizes the main metrics used in this chapter.

Table 2.4: Used metrics.

Name	**Acronym**	**Definition**
Cycles per Instruction	*CPI*	Cycles / Retired Instructions
Demand Misses per Kilo Instruction	*DMPKI*	Demand Misses / (Retired Instructions ÷ 1000)
Access per Kilo Instruction	*APKI*	Access / (Retired Instructions ÷ 1000)
Hit per Kilo Instruction	*HPKI*	Hit / (Retired Instructions ÷ 1000)
Bytes per Kilo Instruction	*BPKI*	Bytes / (Retired Instructions ÷ 1000)

Previous works often use the SLLC local miss ratio (#SLLC misses / #SLLC accesses) instead of DMPKI [100]. However, the SLLC local miss ratio does not consider how often the SLLC is accessed and thus, it does not correlate well with execution time. For instance, a large reduction in the local SLLC miss ratio may not decrease the execution time if the average number of SLLC accesses per instruction is very low. Conversely, a slight SLLC local miss ratio reduction can significantly reduce the execution time if the number of SLLC accesses per instruction is high. On the other hand, DMPKI is a metric that correlates much better with execution time. DMPKI is a global metric, since it is relative to the number of instructions executed. Few misses per instruction should imply little penalty in time and vice versa, although the effect of DMPKI can be influenced by instruction parallelism.

Regarding main memory bandwidth consumption, we use BPKI instead of bytes per time (i.e. bytes per kilo cycle, BPKC) because we want to quantify prefetching overhead. As we will see in section 2.4, prefetching usually results in a significant decrease in the execution time, which in turns causes a BPKI increment, even when the number of bytes read from memory is not increased. BPKI, on the other hand, measures bandwidth consumption per unit of work performed. The value of BPKI in the system without prefetching can be considered a minimum. Any increase in this metric when prefetching is enabled indicates a waste of the memory bandwidth resource due to inaccurate prefetching.

2.4 Evaluation

2.4.1 Identification of Memory Intensive Benchmarks

All the CPU2006 benchmarks (55 from 29 applications) and the CPU2017 single-threaded benchmarks (51 from 23 applications) have been executed first with limited memory resources, disabling all hardware prefetchers and using the minimum SLLC size available, 1.75 MiB, resulting from enabling only one of the eleven SLLC ways.

For each benchmark, we have measured its miss ratios in the three data cache levels (DMPKI1, DMPKI2, DMPKI3). These metrics are shown in Figures 2.2 and 2.3 for SPEC CPU2006 and CPU2017, respectively.

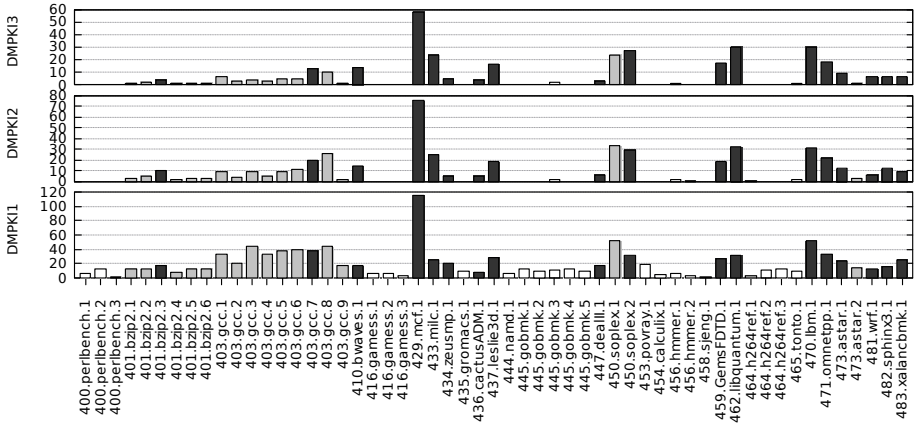


Figure 2.2: DMPKI1, DMPKI2 and DMPKI3 for all SPEC CPU2006 benchmarks, sorted by benchmark number.

If we select for each benchmark the input set with the highest DMPKI3, the average value of DMPKI1 is similar for CPU2006 and CPU2017 (21.9 and 21.8, respectively). However, the average values of DMPKI2 and DMPKI3 are clearly higher in CPU2006 (12.4 and 10.2) than in CPU2017 (8.1 and 6.6). Therefore, a first conclusion is that SPEC CPU2017 does not put more pressure on the memory hierarchy, rather the opposite.

In order to select a set of memory-intensive benchmarks we proceed as follows. Firstly, we identify benchmarks with very low DMPKI2 and DMPKI3 ratios, namely (both below 1.0). Under these circumstances the SKL-SP private caches are sufficient to meet the storage needs and we assume that the SLLC behavior has no interest. Thus, we propose to leave out from further analysis the 43 benchmarks indicated by white-filled bars in the DMPKI1 axes (22 out of 55 in CPU2006, and 21 out of 51 in CPU2017). Secondly, for all the remaining benchmarks we select a single representative for each application, the one with the highest DMPKI2-3 miss ratios (black-filled bars in the figure). Notice that in some cases, the specific application selection extends to both CPU2006 and

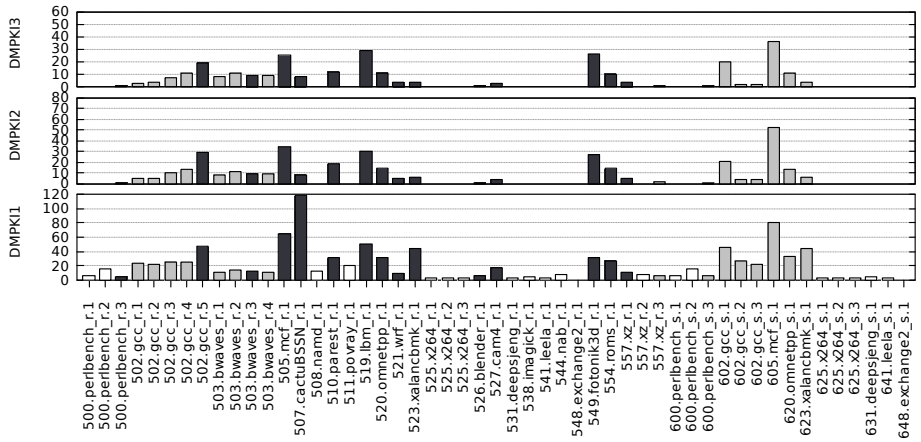


Figure 2.3: DMPKI1, DMPKI2 and DMPKI3 for all SPEC CPU2017 single-threaded benchmarks, sorted by benchmark number.

Table 2.5: Selected benchmarks and their performance metrics for minimum SLLC size and no prefetching.

2006				
Benchmark	*DMPKI1*	*DMPKI2*	*DMPKI3*	*CPI*
401.bzip2.3	18.4	10.2	4.5	0.88
403.gcc.7	38.1	20.6	13.0	1.99
410.bwaves.1	17.5	14.8	14.6	0.96
429.mcf.1	116.3	75.9	59.0	5.08
433.milc.1	26.2	25.1	24.4	1.92
434.zeusmp.1	21.5	5.4	5.0	0.78
436.cactusADM.1	8.0	5.1	4.5	0.91
437.leslie3d.1	29.0	18.3	17.0	1.16
447.dealII.1	18.7	6.6	3.8	0.72
450.soplex.2	31.9	29.1	27.6	2.35
459.GemsFDTD.1	27.1	18.8	17.5	1.39
462.libquantum.1	32.8	32.2	30.5	1.22
470.lbm.1	52.3	31.6	30.2	1.42
471.omnetpp.1	34.6	22.9	18.4	1.52
473.astar.1	24.8	12.4	9.8	1.08
481.wrf.1	12.2	6.4	6.0	0.82
482.sphinx3.1	16.3	12.4	6.9	0.68
483.xalancbmk.1	26.3	9.1	6.1	0.64

2017				
Benchmark	*DMPKI1*	*DMPKI2*	*DMPKI3*	*CPI*
500.perlbench_r.3	5.8	1.9	1.5	0.71
502.gcc_r.5	47.2	29.3	19.1	2.16
503.bwaves_r.3	12.5	9.9	9.7	0.75
505.mcf_r.1	65.8	34.2	25.7	1.97
507.cactuBSSN_r.1	118.3	8.8	8.4	1.02
510.parest_r.1	31.5	18.7	12.5	1.34
519.lbm_r.1	50.9	30.7	29.0	1.59
520.omnetpp_r.1	32.3	14.1	11.3	1.84
521.wrf_r.1	10.9	5.6	4.7	1.26
523.xalancbmk_r.1	44.2	6.7	4.4	1.11
526.blender_r.1	7.5	1.9	1.5	0.68
527.cam4_r.1	18.5	4.4	3.0	0.73
549.fotonik3d_r.1	32.3	27.8	27.0	2.06
554.roms_r.1	27.6	14.4	10.9	1.07
557.xz_r.1	11.9	5.4	4.3	1.34

2017, as is the case, for example, of `mcf`. Table 2.5 shows the resulting 33 benchmarks (18 from CPU2006 and 15 from CPU2017) that form our selected workload of memory-intensive benchmarks which will be analyzed in depth in the next subsections.

Characterization highlights:

- SPEC CPU2017 puts less pressure than CPU2006 on the memory hierarchy.
- Less than 50% of the SPEC CPU2006 and CPU2017 benchmarks

show DMPKI2 and DMPKI3 ratios above one, even with hardware prefetching disabled and with the minimum allocatable size of SLLC.

2.4.2 Sensitivity to LLC Size and Hardware Prefetching

In this experiment, we study the sensitivity of the memory intensive benchmarks to SLLC size and hardware prefetching. The benchmarks selected in the previous section have been executed with five SLLC sizes. For each SLLC size, two executions have been performed: all prefetchers enabled and all disabled.

The evaluated SLLC sizes are: 1.75 MiB, 3.5 MiB, 7 MiB, 14 MiB and 19.25 MiB, corresponding to associativities of 1, 2, 4, 8 and 11, respectively.

Figures 2.4 and 2.5 show the DMPKI3 of the selected benchmarks for the different SLLC sizes, with and without hardware prefetching.

Without hardware prefetching, increasing the SLLC size results in an DMPKI3 reduction for almost all benchmarks of both suites, with the exception of `410.bwaves`, `434.zeusmp`, and `459.GemsFDTD` in CPU2006, and `503.bwaves` in CPU2017.

With hardware prefetching, the DMPKI3 improvement achieved by increasing the SLLC size is considerably reduced for six CPU2006 benchmarks (`433.milc`, `437.leslie3d`, `447.dealII`, `450.soplex.2`, `462.libquantum` and `481.wrf`), and for five CPU2017 benchmarks (`510.parest`, `519.lbm`, `521.wrf`, `549.fotonik3d` and `554.roms`).

Both hardware prefetching and increasing SLLC size have the same goal, improving performance by decreasing cache misses. Therefore, when either one of these two techniques acts effectively, it removes part of the problem and reduces the need for the other one. Hardware prefetching is very effective for many applications. Even with the minimum SLLC size (1.75 MiB in our system), DMPKI3 is very low for many benchmarks running with prefetching activated. In all these cases, increasing the SLLC size does not provide any benefit. For instance, 510.parest_r.1 clearly shows this behavior. Without prefetching, DMPKI3 decreases from 12.5 to 0.6 by increasing the SLLC size from 1.75 to 14 MiB. However, when enabling the prefetchers, the DMPKI3 with 1.75 MiB is already 1.0, so any further increase in the SLLC size provides very little benefit. This reduction of DMPKI3 happens even though the SKL-SP hardware prefetching only fill the prefetch request into L1D and L2. DMPKI3 can decrease with hardware prefetch, because a block prefetched into private levels can eliminate a future demand request, therefore eliminating the SLLC demand miss.

It also reduces DMPKI3 for *small* SLLC sizes in other four CPU2006 benchmarks (`401.bzip2`, `473.astar`, `482.sphinx3` and `483.xalancbmk`) and two CPU2017 ones (`510.parest` and `523.xalancbmk`). For one benchmark,

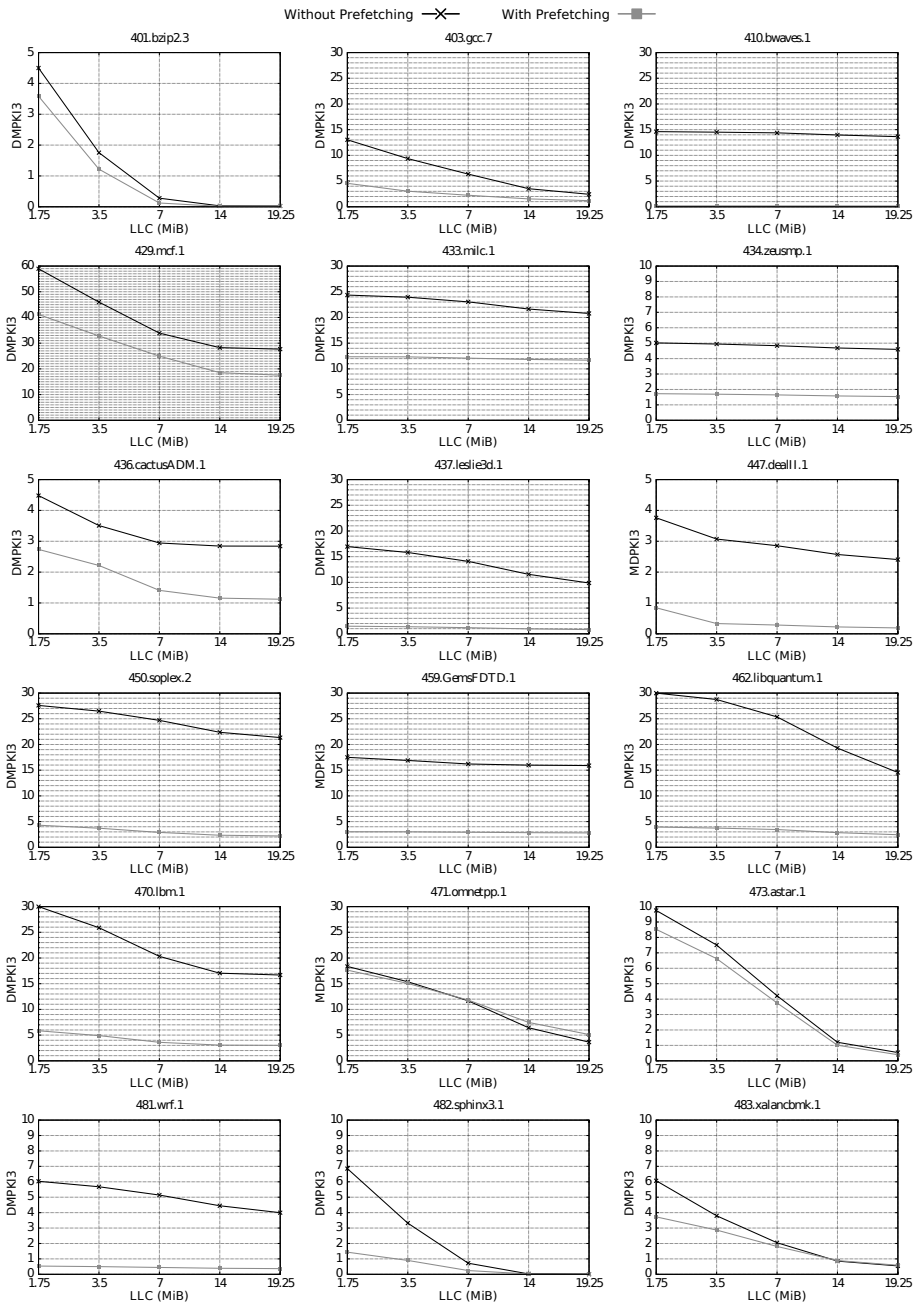


Figure 2.4: DMPKI3 vs. SLLC size for the selected CPU2006 benchmarks, with and without prefetching.

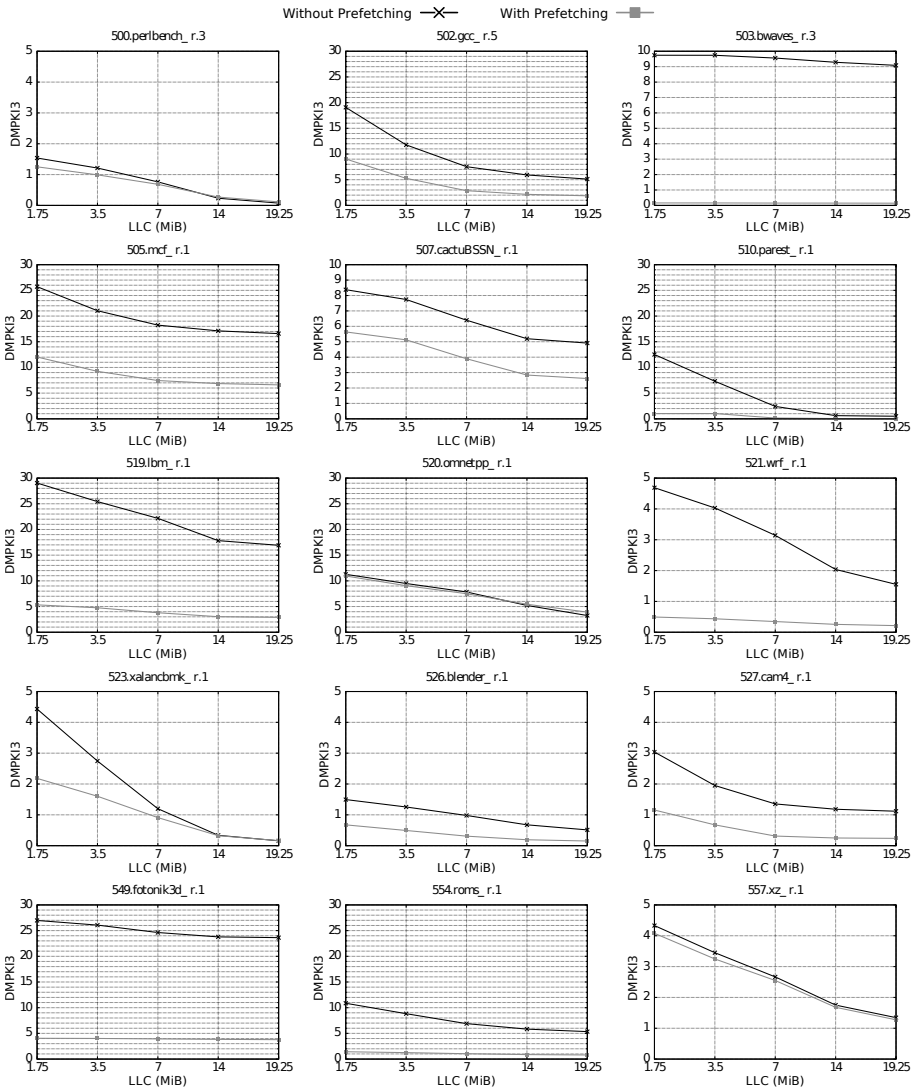


Figure 2.5: DMPKI3 vs. SLLC size for the selected CPU2017 benchmarks, with and without prefetching.

`omnetpp`, which is included in both suites, SLLC misses are not reduced for any SLLC size, and even are slightly increased with the largest SLLC size.

DMPKI3 can decrease with hardware prefetching even if the prefetched blocks are not filled into SLLC (SKL-SP hardware prefetch requests are filled into L1 or L2). This is because a block prefetched that is only filled into the private levels can eliminate a future demand request, therefore eliminating the SLLC demand miss.

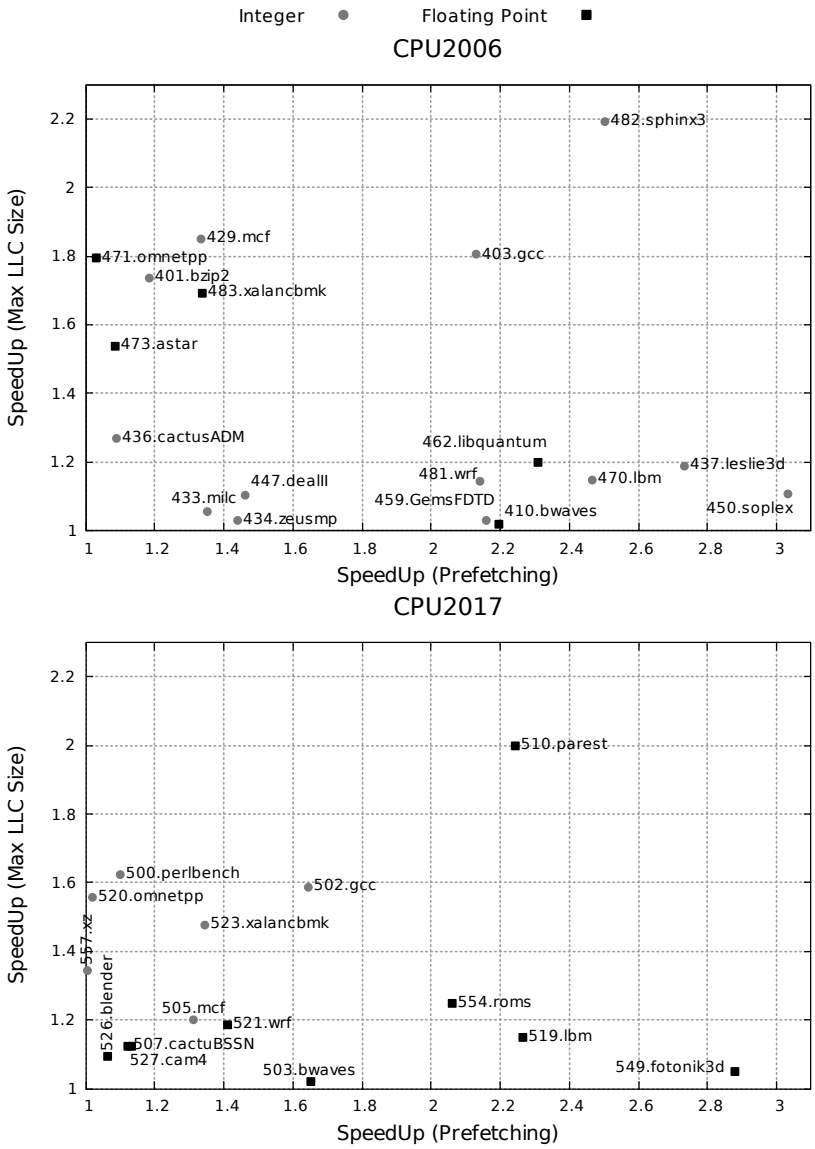


Figure 2.6: Speed-ups enabled either by hardware prefetching, with the minimum cache size (X axis) or maximum SLLC size, without prefetching (Y axis) over a baseline configuration without prefetching and minimum SLLC size for the selected CPU2006 and CPU2017 benchmarks. Integer and floating point benchmarks are represented by gray circles and black squares, respectively.

To summarize, Figure 2.6 shows the speedups of benchmarks when prefetching is enabled with the minimum cache size (X axis), and when the SLLC size is

increased up to 19.25 MiB without prefetching (Y axis) with respect to a baseline system with the minimum SLLC size and without prefetching. Figure 2.6 facilitates the classification of benchmarks according to their sensitivity to both parameters. Integer and floating point benchmarks are plotted with gray circles and black squares, respectively. For instance, we can see a group of CPU2006 benchmarks that are very sensitive to hardware prefetching but show little sensitivity to SLLC size increase (`462.libquantum`, `481.wrf`, `459.GemsFDTD`, `410.bwaves`, `470.lbm`, `437.leslie3d` and `450.soplex`).

Figure 2.6 also allows us to analyze whether the clustering of applications made by other proposals is in agreement with the sensitivity of these applications with respect to the prefetch and the increase in SLLC size. As an example, Limaye et al. [100] classify `510.parest` and `503.bwaves` of SPEC CPU2017 as very similar benchmarks. However, in our classification we can see that `510.parest` is the CPU2017 benchmark which is most sensitive to the SLLC size while `503.bwaves` is the least sensitive one.

Characterization highlights:

- Hardware prefetching is very effective in reducing DMPKI3 for *all* SLLC sizes in 14 and 10 CPU2006 and CPU2017 benchmarks, respectively. This is because prefetching, regardless of the size of the cache, detects the right patterns and goes ahead of the memory reference stream correctly. Therefore, in those benchmarks where DMPKI3 is high for all SLLC sizes, hardware prefetching is effective in reducing the DMPKI3 also for all SLLC sizes.
- The utilization of SLLC is uneven among the applications. Some applications take advantage of a larger SLLC size, and others are cache size-independent.

Correlation between DMPKI3 and CPI

Figures 2.7 and 2.8 show the correlation between DMPKI3 (X axis) and CPI (Y axis) for the different SLLC sizes, with (square marks) and without (x marks) prefetching. The slope of the CPI/DMPKI3 linear interpolation gives thousand of cycles per SLLC miss: (cycles/instruction)/(SLLC misses/Kinstruction). For instance, the slope of `401.bzip2.3` in Figure 2.7 is 0.084 Kcycles/SLLC_miss. For the sake of clarity, inside the figures we write instead 84 cycles/SLLC_miss. This number represents the average SLLC miss penalty for the benchmark.

As can be seen in Figures 2.7 and 2.8, the linear relationship between DMPKI3 and CPI is strong, although the DMPKI3 increase has different impacts on the benchmarks' CPI. The slope of the interpolation line varies between 30 cycles per miss for several benchmarks in both suites and 187

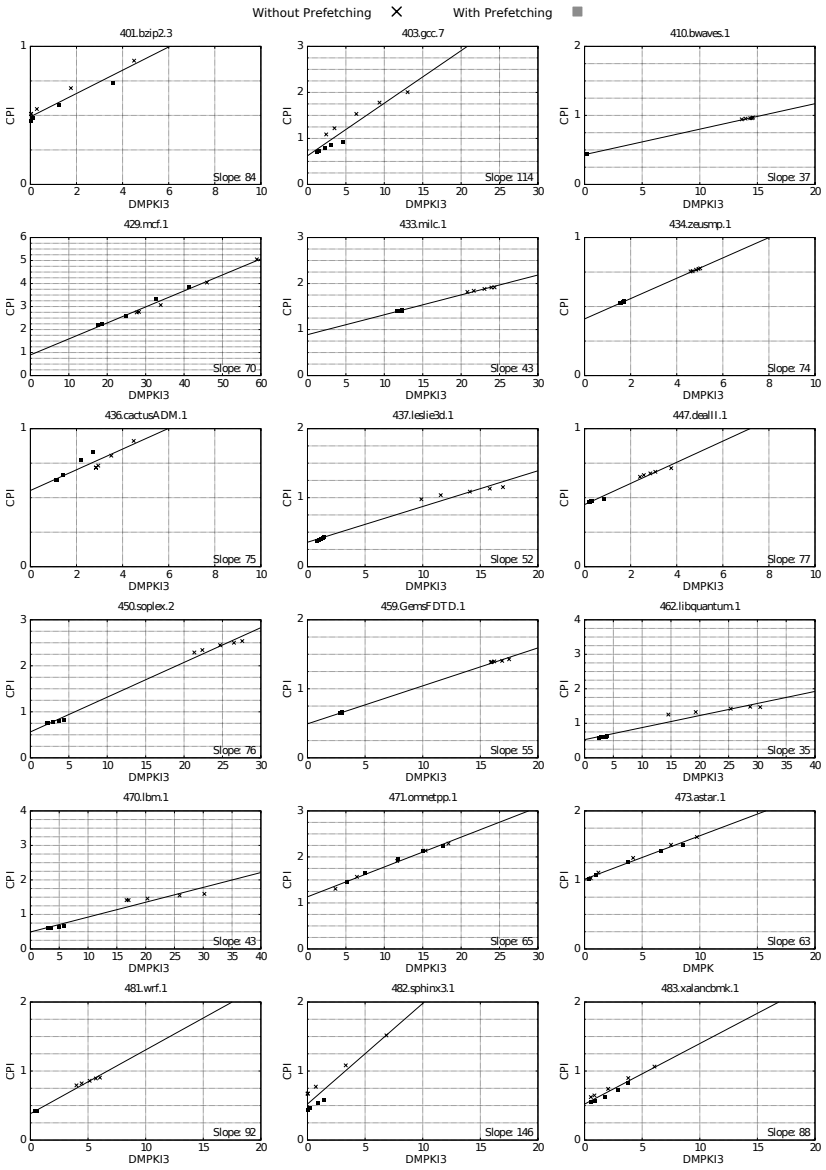


Figure 2.7: CPI vs. DMPKI3 for the selected CPU2006 benchmarks, varying SLLC size and with prefetching (square marks) and without prefetching (x marks). Slope units are cycles/miss. Slopes are comparable in all graphs because the ratio between X and Y scales is constant (10:1).

cycles per miss for `500.perlbench_r.3`. The slope value provides another criteria to classify benchmarks according to the amount of instruction-level and memory level-parallelisms [68]. For instance, a large slope value means a

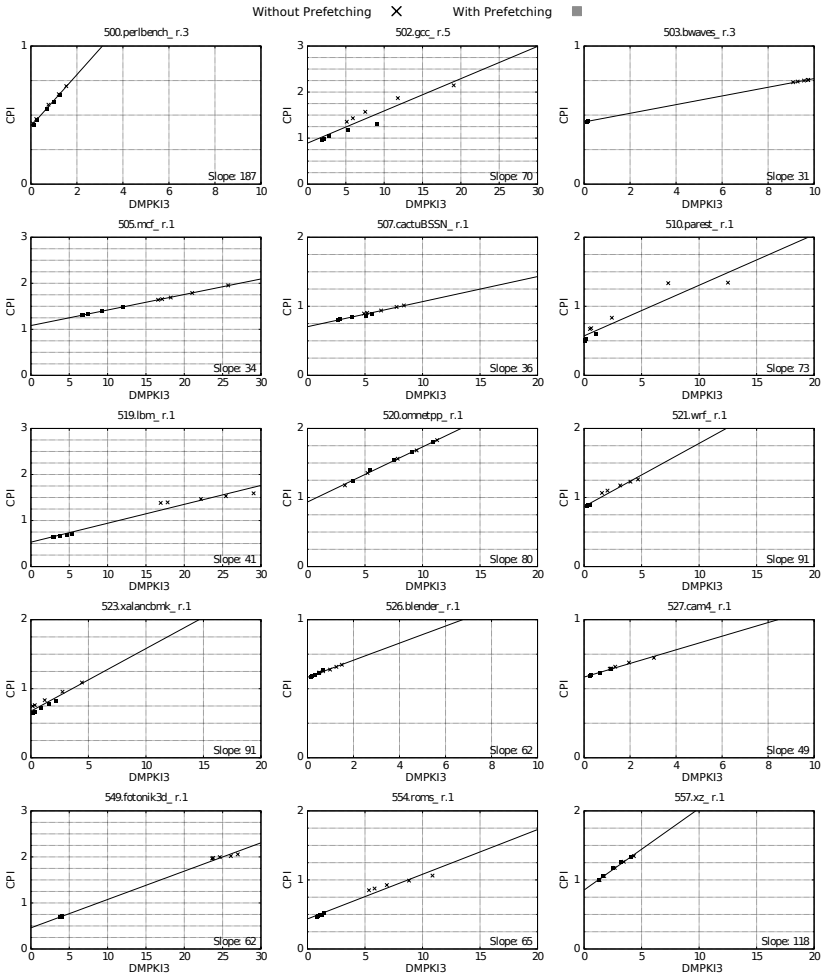


Figure 2.8: CPI vs. DMPKI3 for the selected CPU2017 benchmarks, varying SLLC size and with prefetching (square marks) and without prefetching (x marks). Slope units are cycles/miss. Slopes are comparable in all graphs because the ratio between X and Y scales is constant (10:1).

low instruction-level and memory-level parallelism (low temporal overlapping among computation and SLLC misses, and low temporal overlapping of SLLC misses with themselves), as seen in `482.sphinx3.1` and `500.perlbench_r.3`.

Characterization highlight: Not all applications benefit equally, performance wise, from a decrease in DMPKI3.

2.4.3 Performance of the Hardware Prefetchers

In this subsection we analyze the impact of the different hardware prefetchers on the benchmarks' performance and bandwidth consumption. SKL-SP processors have four hardware prefetchers associated with the first and second cache levels [77]: *L1 Data cache unit prefetcher* (DCUI), *L1 Data cache instruction pointer stride prefetcher* (DCUP), *L2 Data cache spatial prefetcher* (L2A) and *L2 Data cache streamer prefetcher* (L2P).

All the benchmarks selected in Section 2.4.1 have been executed with different configurations: all prefetchers enabled, all prefetchers disabled and each prefetcher individually enabled. The experiments have been performed with the maximum SLLC size. Figures 2.9 and 2.10 show performance measured in cycles per instruction (CPI, left axis, bars) and bandwidth consumption measured in bytes read from main memory per kilo instruction (BPKI, right axis, line).

L2P is by far the best prefetcher. For the 14 CPU2006 benchmarks whose miss ratios are reduced by turning on hardware prefetching with the maximum SLLC size, L2P, by itself, achieves more than 70% of the CPI reduction obtained with all prefetchers enabled. Furthermore, for 10 of these 14 benchmarks, L2P is responsible for more than 90% of the CPI reduction obtained with all the prefetchers enabled. Similar results are observed for the CPU2017 suite. For the 10 benchmarks that even with the biggest SLLC size take advantage of hardware prefetching, L2P alone achieves more than 82% of the CPI reduction obtained when all prefetchers are active. This percentage is greater than 90% for 7 out of these 10 benchmarks.

The second-best prefetcher is DCUI, followed by DCUP. L2A obtains the worst results, since it only reduces CPI in more than 5% for 8 and 6 of the CPU2006 and CPU2017 benchmarks, respectively, with a maximum of 20% for `450.soplex`.

Characterization highlights. *Hardware prefetching is very accurate, since it only causes a significant increase of bandwidth consumption in 3 benchmarks of the CPU2006 suite (`403.gcc.7`, `433.milc` and `471.omnetpp`) and in 3 CPU2017 benchmarks (`520.omnetpp`, `549.fotonik3d` and `554.roms`). Moreover, in most of these benchmarks prefetching causes a considerable CPI reduction despite the increase in bandwidth consumption, with the exception of `omnetpp` (both `471.omnetpp` and `520.omnetpp`).*

2.4.4 Temporal Evolution of the Benchmarks

This subsection analyzes the temporal evolution of the benchmarks. Figures 2.11 and 2.12 show DMPKI3 (Y axis) for every million of executed instructions (X axis). This allows us to know the different phases of a benchmark and can help select simulation intervals. The execution was performed with the minimum SLLC size (1.75 MiB) and with all the hardware prefetchers enabled. We did the same analysis using other SLLC configurations and even using the metrics

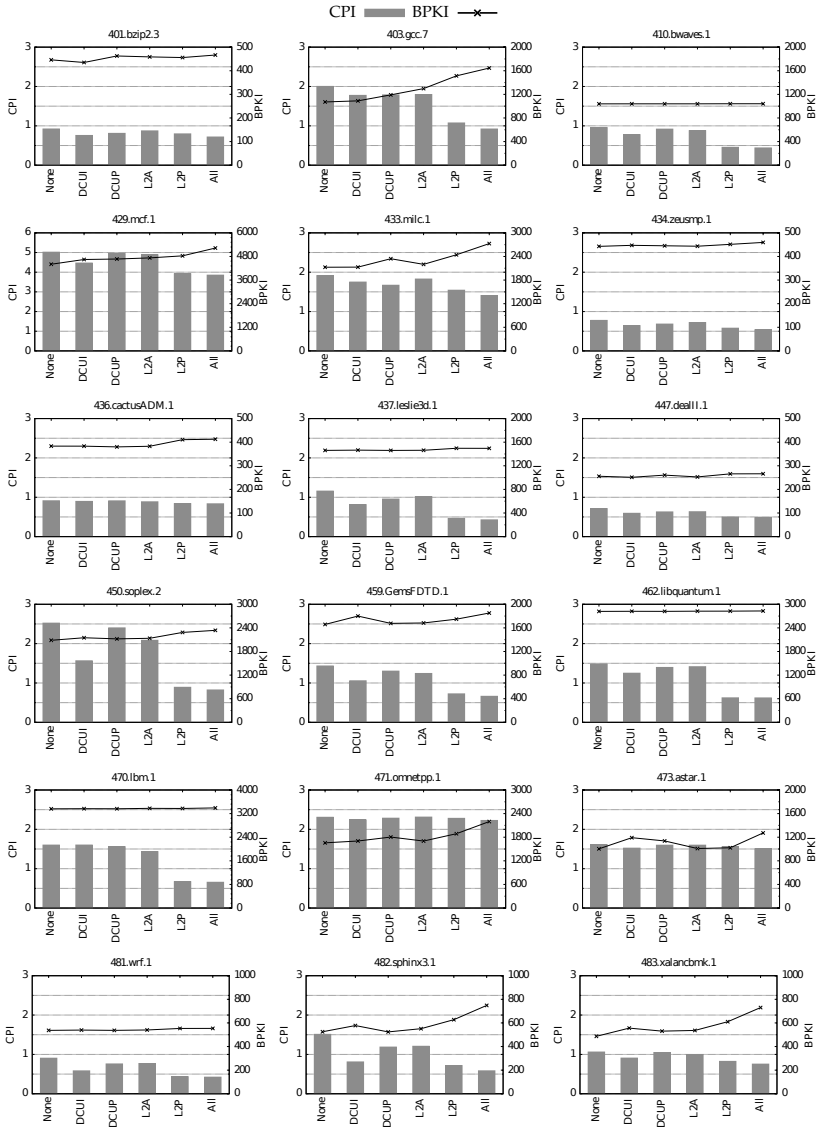


Figure 2.9: Impact of the different hardware prefetchers on performance (CPI, bars) and bandwidth consumption (BPKI, line) for the selected CPU2006 benchmarks.

of the other cache levels (DMPKI1 and MPKI2), and a total similarity was observed in all experiments. Thus, the temporal evolution of the applications seems to be very independent of the SDLLC configuration.

The graphs in Figures 2.11 and 2.12 also plot three vertical lines of different

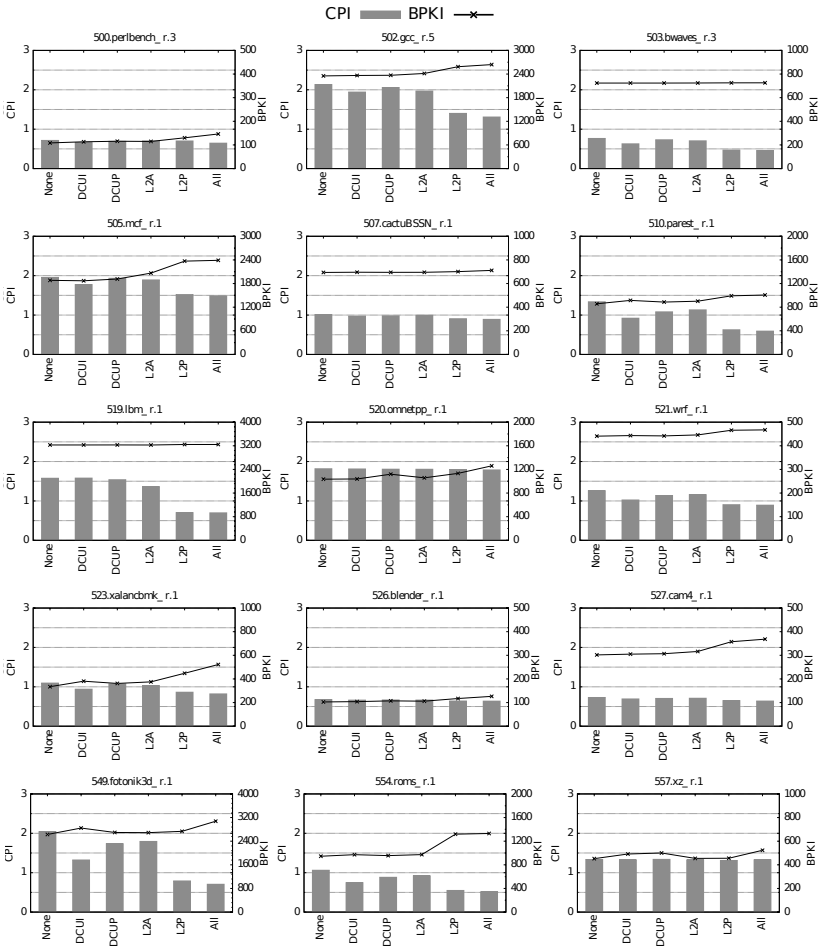


Figure 2.10: Impact of the different hardware prefetchers on performance (CPI, bars) and bandwidth consumption (BPKI, line) for the selected CPU2017 benchmarks.

patterns representing the first three simulation intervals obtained by *SimPoint* with the parameters MaxK = 3 and interval size = 300 million instructions. The solid, dotted, and dashed lines correspond to the most, second most, and third most representative intervals, respectively. The thickness of each vertical line is not proportional to the number of instructions that it represents. It has been increased in order to improve visibility.

The simulation intervals selected by *SimPoint* are not always representative of all the different phases of a benchmark. As an example, we can observe that `401.bzip2` in Figure 2.11 clearly has three phases with different DMPKI3 values and similar duration, approximately 100 million instructions each one.

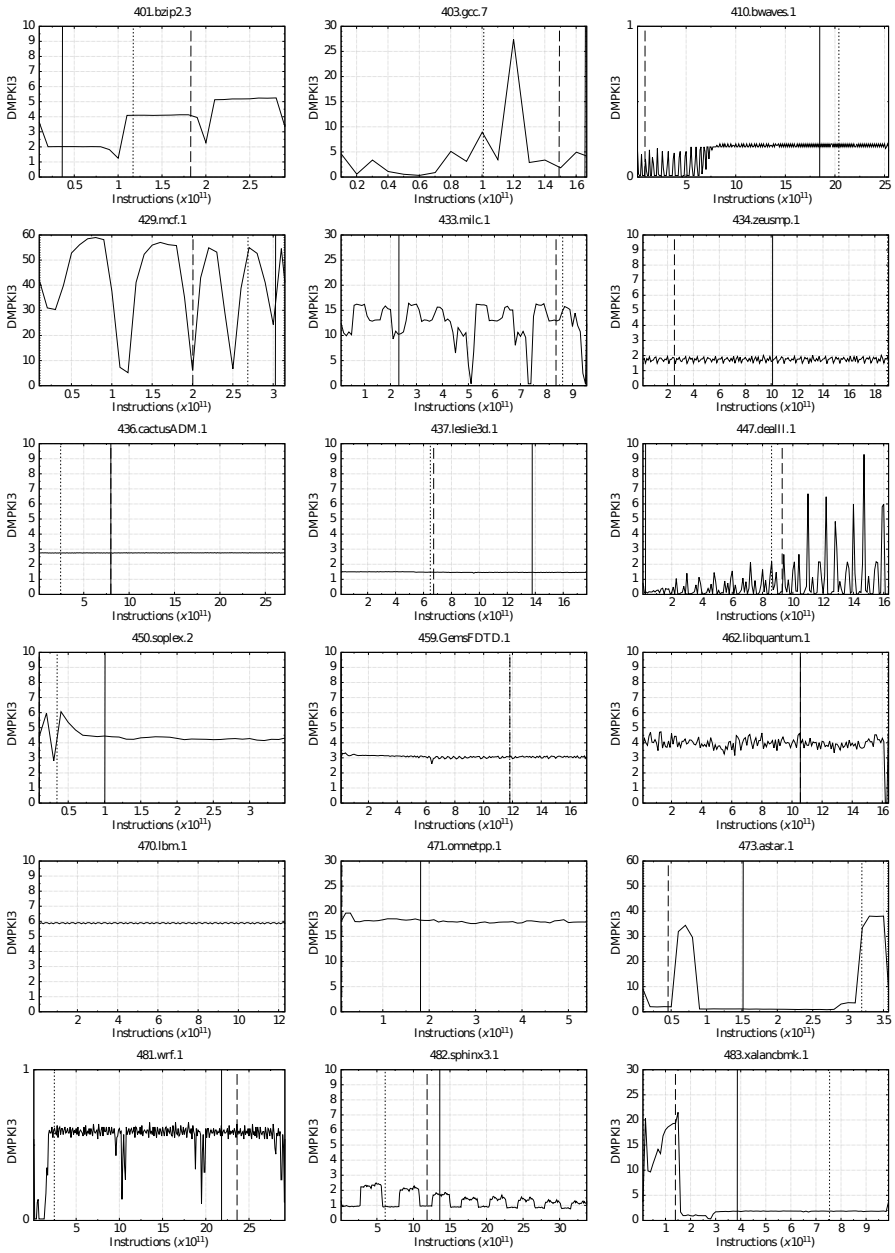


Figure 2.11: Temporal evolution of DMPKI3 and *SimPoint* selection for the selected CPU2006 benchmarks, with minimum SLLC size and hardware prefetching.

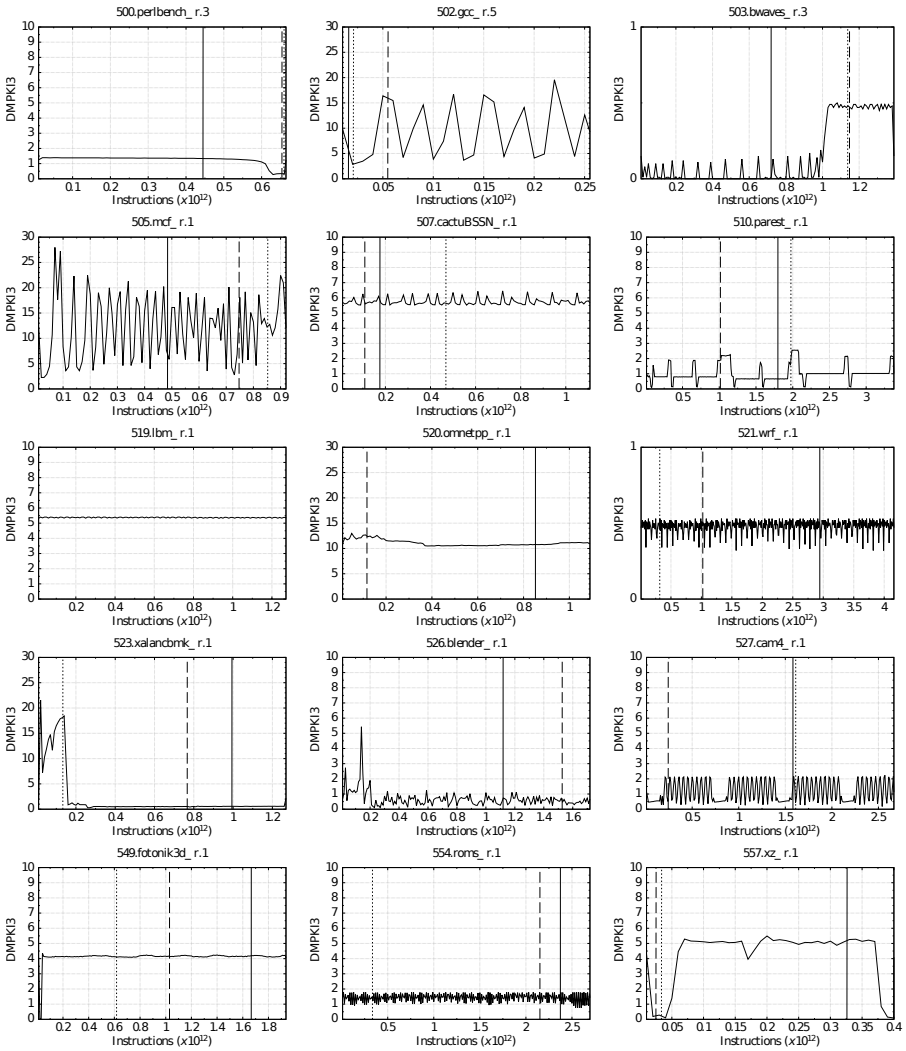


Figure 2.12: Temporal evolution of DMPKI3 and *SimPoint* selection for the selected CPU2017 benchmarks, with minimum SLLC size and hardware prefetching.

The DMPKI3 of each phase is very different with values around 2.0, 4.1 and 5.2, respectively. However, SimPoint selects its first interval from the first phase, the next two intervals from the second phase, and does not select any interval from the third phase. This is because Simpoint, as the methodology used to select applications, focuses on memory unrelated parameters, giving accurate outcomes, but only to evaluate design tradeoffs related to those parameters.

Characterization highlight: *SimPoint* has limitations to obtain representative intervals of a benchmark execution from the memory hierarchy point of view. The problem gets worse because most research papers based on this methodology only select the first interval.

Selection of benchmarks and simulation intervals

SPEC CPU2006 and SPEC CPU2017 are composed of several applications, some of them with different inputs, resulting in multiple application-input combinations. We define *benchmark* as an application-input pair. For example, there are 29 applications and 55 benchmarks (application-input pairs) in the CPU2006 suite. On one hand, the execution time of a complete benchmark on a simulator may last weeks or even months, which makes the simulation of a suite unfeasible. On the other hand, running all benchmarks on a real machine may be possible, but uninteresting. Since we may only be interested in testing benchmarks with certain characteristics, e.g. benchmarks that put pressure into the cache hierarchy or into the branch predictor, it is of no interest to run all benchmarks on a real machine. To address these issues, a one or two-level sampling can be carried out. First, a subset of benchmarks with the characteristics we are interested in is selected. Second, in case of time constrain due to simulation, one or more fragments of the complete execution representing the overall behavior are chosen as simulation intervals. In the literature, several successful sampling techniques have been proposed, such as *Hierarchical Clustering* [100] for benchmark selection, and *SimFlex* [165] or *SimPoint* [131] for intervals selection.

Hierarchical Clustering [100] The *Hierarchical Clustering* methodology is applied in three steps:

1. Execution of all benchmarks to obtain 20 metrics through hardware counters.
2. Analysis of the main components to reduce the number of metrics to 4.
3. Clustering of similar benchmarks.

In the first step, the authors select microarchitecture-independent metrics which are related to the type of instructions executed and their proportions. Therefore, this methodology does not consider the behavior of the memory hierarchy as a parameter to guide sampling.

SimFlex [165] The *SimFlex* methodology uses statistical sampling theory to select simulation intervals. It identifies numerous small-sized intervals that

are distributed throughout all the application execution, ensuring that they are representative of the benchmark.

However, *SimFlex* has an important drawback when it is applied to cache memory hierarchy research: the simulation intervals do not have enough extension to provide accurate data without a previous cache warm-up. Warming memory structures has an unacceptable overhead when simulating large caches or a large number of intervals.

SimPoint [131] *SimPoint* is one of the most used methodologies to select simulation intervals. First, it splits up the execution of a benchmark into intervals of equal number of instructions. For each interval, *Simpoint* calculates a signature that contains the number of executions of each basic block. Then, *SimPoint* executes the *k-Means* algorithm to group different intervals into clusters called phases. The intervals of a given phase execute similar code and therefore are expected to exhibit a similar behavior in the system (misses in the memory hierarchy, CPI ...). Finally, the centroid is selected as the most representative interval of each phase.

Although *SimPoint* offers several simulation intervals for each benchmark, most research related to memory hierarchy design uses only the most representative interval.

One of the contributions of this chapter is to assess the representativeness of the intervals selected by *SimPoint* regarding the interaction with the memory hierarchy. Towards that end we analyze the temporal evolution of different metrics, namely CPI, DMPKI2 and DMPKI3, across the whole application execution. By plotting such metrics as a function of time and superposing the first three intervals selected by *SimPoint*, we will see how well they match the memory hierarchy dynamics.

2.5 Concluding Remarks

In this chapter we have analyzed the performance of the memory hierarchy of an SKL-SP processor executing the SPEC CPU2006 benchmarks and CPU2017 single-threaded benchmarks. Below, we summarize the main conclusions that we can draw from this characterization.

A significant number of the benchmarks have very low miss ratios in the second and third level caches, even with a small SLLC size and without hardware prefetching. The CPU2017 demand for memory hierarchy resources is lower than the CPU2006 one.

We offer a classification of the benchmarks that demand resources in SLLC according to their sensitivity to SLLC size and hardware prefetching. Hardware prefetching is very effective in reducing SLLC misses for most benchmarks, even with the smallest SLLC size. Increasing the SLLC size is also effective in

reducing SLLC miss counts for many benchmarks. The effect of SLLC size is uneven across the different benchmarks.

The best prefetcher implemented in the SKL-SP processor is L2P. For most benchmarks, it is responsible for 90% of the CPI reduction when using prefetching. Hardware prefetching is very accurate. In general, the number of bytes read from main memory hardly increases.

Our analysis shows that the methodologies used in other works to select benchmarks [100] and simulation points [131] do not guarantee that representative workloads from the memory hierarchy point of view are obtained.

Chapter 3

Balancer: Bandwidth Allocation and Cache Partitioning

In recent years, processors brands like Intel or AMD have included technologies for SLLC content and/or bandwidth control between the different levels of memory hierarchy. The inclusion of these technologies has enabled the implementation of software control mechanisms that allow system administrators and/or users to achieve performance, fairness, power consumption or quality of service (QoS) objectives. In this chapter, we extend the characterization done in chapter 2 by analyzing the behavior of the SPEC CPU2006 and SPEC CPU2017 benchmark suites in an AMD EPYC Rome, and how applications interfere with each other in multiprogrammed workloads. We also characterize the performance of the benchmarks for different SLLC sizes and contention levels with memory bandwidth. Then, we present Balancer, a new set of mechanism that control the SLLC space and the DRAM memory traffic to improve the performance and fairness of a system when executing multiprogrammed workloads.

3.1 Introduction

Processors have an increasing number of cores and execution threads. For instance, AMD integrates up to 64 cores capable of executing 128 threads in the Rome and Milan microarchitectures [58, 63], Intel Xeon Skylake-SP processors can integrate up to 28 cores with 56 threads [28], and the IBM Power10 processors can integrate up to 120 threads, either with 15 SMT8 or 30 SMT4 core configurations [60]. This trend places greater pressure on shared resources whose capabilities, especially the SLLC and off-chip memory bandwidth, should be scaled and shared in the best possible way.

Shared resources, like SLLC or DRAM bandwidth, can be managed by imposing limits on their use by one or more threads. In this way, each thread can only use its allocated quota, reducing the interference with other threads. In recent years, commercial processors such as Intel Xeon [69], Arm ThunderX [163],

or AMD EPYC [58, 63], have included hardware support for users to control the allocation of both SLLC space and memory bandwidth to processor threads.

Hardware support for SLLC space management has given rise to many proposals pursuing one or more goals, such as improving system performance [44, 170, 95, 172, 129, 26, 158, 174, 163, 96] or fairness [144, 53], facilitating server consolidation and/or ensuring quality of service (QoS) [130, 160, 101, 117, 129, 177, 23], isolating tasks to decrease the worst case execution time (WCET) [173], decreasing the Turnaround Time [132], or seeking Social Welfare [51].

However, since processor support for memory traffic management is more recent, work on this topic is scarcer. Specifically, we are aware of only three proposals that control the memory bandwidth allocation to improve performance [129] or to support server consolidation [171, 130].

Table 3.1: Summary of papers on resource management with real machine experimentation. LLC: last-level cache, which can be inclusive (I) or non-inclusive (NI). BW_{mem}: memory bandwidth, IC: interconnect, MC: memory controller, Freq: core frequency, BW_{disk}: disk bandwidth, #Cores: number of cores, $Pref_{hw}$: hardware prefetcher, Net: Internet connection. The rows are arranged in chronological order of the processor on which the mechanisms are applied.

	Processor	**Resources**	**Goal**
Merlin [160]	Intel Westmere	LLC_I, IC, MC	Consolidation
Cook et al. [26]	Intel Sandy Bridge	LLC_I	Performance
Sun et al. [158]	Intel Sandy Bridge	LLC_I, $Pref_{hw}$	Performance
Pons et al. [132]	Intel Sandy Bridge	LLC_I	Turnaround Time
Heracles [101]	Intel Haswell	LLC_I, Freq, Net	Consolidation
Ginseng [51]	Intel Haswell	LLC_I	Social Welfare
Dirigent [177]	Intel Haswell	LLC_I, Freq	Consolidation
Selfa et al. [144]	Intel Haswell	LLC_I	Fairness
vCAT [173]	Intel Haswell	LLC_I	WCET
DCAPS [170], **DCAT** [174]	Intel Haswell	LLC_I	Performance
Chen et al. [23]	Intel Haswell	LLC_I, Freq, BW_{disk}, #Cores, Net	Consolidation
Kpart [44]	Intel Broadwell	LLC_I	Performance
Dicer [117]	Intel Broadwell	LLC_I	Consolidation
SWAP [163]	Cavium ThunderX	LLC	Performance
Kim et al. [96], **CPPF** [172]	Intel Skylake-SP	LLC_{NI}	Performance
Hypart [129]	Intel Skylake-SP	BW_{mem}	Performance
Copart [130]	Intel Skylake-SP	LLC_{NI}, BW_{mem}	Consolidation
EMBA [171]	Intel Skylake-SP	BW_{mem}	Consolidation
LFOC [53]	Intel Skylake-SP	LLC_{NI}	Fairness

As shown in Table 3.1, almost all previous works focus on Intel memory hierarchies, where the SLLC is shared among all cores, but there are no proposals specifically designed for a clustered SLLC organization, such as the one used by AMD in its contemporary processors [58, 63]. Focusing on the AMD EPYC, the SLLC is shared among only a few cores and the main memory access links form a hierarchical tree that terminates in several DRAM channels (details in Section 3.2). In this memory hierarchy design, the management of SLLC and memory bandwidth may become more complex, as the potential occurrence of local, intermediate, and/or global bottlenecks have to be taken into account.

Therefore, this chapter focuses on the efficient execution of multiprogrammed workloads on a real instance of a clustered core-LLC organization. However, the ideas used in the design could be applied to other processors, even with shared LLC. We propose very simple mechanisms, with negligible software overhead, that rely on existing monitoring and control support and require no hardware or operating system changes. The goal is to improve the system performance and thread fairness by selecting certain cores, for which the space they occupy in the LLC and/or the memory bandwidth available to them will be dynamically limited or increased.

We believe that this type of clustered organization is a promising trend, since increasing the number of processor cores makes a cache shared by all cores more inefficient in terms of access latency and interconnect network traffic. In fact, clustered organizations can be found in recent high-performance processors such as AMD EPYC [91, 90], IBM Power 10 [157], Fujitsu A64FX [50]. Moreover, a clustered design it allows the desired growth in the number of transistors to be achieved economically by integrating a set of separately manufactured dies (chiplets in AMD terminology [109]) into a passive module. Moreover, the application behavior in clustered hierarchies, especially when all the cores are running threads, has also not been studied. Not knowing if this behavior makes it difficult to choose existing control mechanisms or to propose new ones.

The remaining of this chapter is organized as follows. Section 3.2 describes the execution environment, the monitoring and control tools, and the workloads used in the characterization and the evaluation. Section 3.3 analyzes the impact of LLC occupancy and main memory traffic on the execution of multiprogrammed workloads and Section 3.4 details the Balancer mechanisms. Section 3.5 shows and analyzes the experimental results regarding performance and fairness. Section 3.6 describes the related works. The chapter ends with the concluding remarks in Section 3.7

3.2 Experimental Framework and Methodology

This section shows the main features of the selected server, the monitoring and control tools, and the employed workloads.

3.2.1 AMD Rome Core Organization

AMD launched in 2019 a family of processors aimed at the high-performance server segment. These models are based on the Zen2 microarchitecture, code-named Rome [58]. Specifically, we use a server with a 64-core EPYC 7702P processor. The 7702P processor is made up of up to eight compute core dies (CCDs) that are connected to each other and to the off-chip memory via an I/O die (Figure 3.1). Each CCD integrates two core complex (CCX) units that share an I/O connection. In turn, each CCX has four cores capable of executing eight threads sharing a 16 MiB victim SLLC with 16 ways, i.e. the LLC gets populated with the cache blocks evicted from the four L2 caches of a CCX.

This non-inclusive content management is also implemented in other recent processors such as Arm Neoverse [62] and Intel Skylake-SP [57]. Although there are 256 MiB of SLLC in total, note that the four cores of a CCX cannot store cache blocks outside of their 16 MiB SLLC. Each core contains a 512 KiB unified second level cache with 8 ways, and a split first level cache (32 KiB for instructions and 32 KiB for data, both with 8 ways). It also has two L1D hardware prefetchers (stride and region), and two L2C prefetchers (stream and next-line) [7]. Unlike in the Intel systems the hardware prefetchers can not be selectively enabled or disabled.

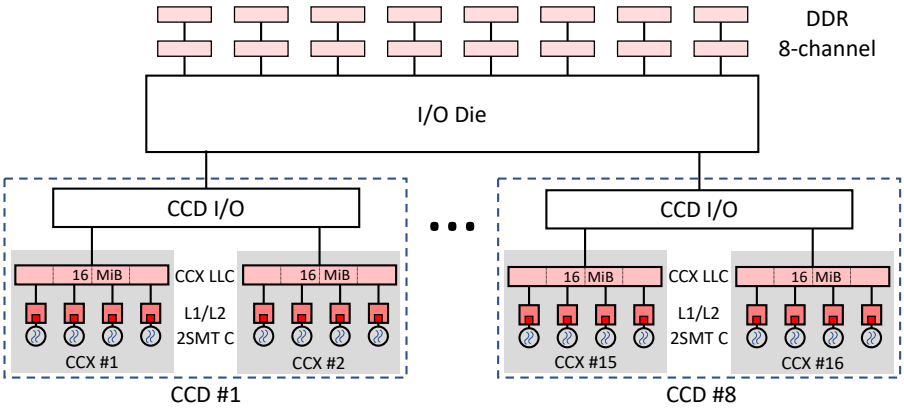


Figure 3.1: AMD Rome 7702P clustered memory hierarchy. The multichip module has nine dies: eight CCD dies and one I/O die. In total there are 64 2-SMT cores (2SMT C) organized in 8 CCDs, each one with 2 CCX. Each CCX has a 16 MiB LLC shared by four cores [58].

3.2.2 Runtime Environment

In order to ensure the reproducibility of the experiments, unless otherwise stated, each execution thread is pinned to a core, TurboBoost has been disabled, and the energy governor is set to performance.

The system runs a CentOS 8.2 GNU/Linux with the 4.18.0 kernel. The Table 3.2 details the main features of the system.

3.2.3 Monitoring and Control Tools

The monitoring of the events that allow to characterize the applications and then guide the Balancer control mechanisms has been done using hardware counters, see section 2.3.4. Unlike in the previous chapter, we manage them by reading and writing the corresponding model-specific register (MSR) [33]. GNU/Linux makes the MSR registers accessible through access to the files `/dev/cpu/[0-N-1]/msr`, where `N` is the number of cores. Table 3.3 shows the

Table 3.2: Main features of the selected server.

Processor	AMD EPYC 7702P
Cores × Threads	64 × 2
L1 I-Cache	32 KiB, 64 B line size, 8 ways
L1 D-Cache	32 KiB, 64 B line size, 8 ways
L2C	8-ways 512 KiB, 8-ways
LLC	16 MiB, 16-ways, non-inclusive (per CCX)
Main Memory	256 GiB DDR4, 8 channels
	Nominal peak BW: 204.8 GB/s
TurboBoost	Disabled
Hyperthreading	Disabled (1 thread/core)
OS	CentOS 8.2, kernel 4.18.0
Python	3.6.8

formulas used to calculate each metric from specific hardware counters of core or CCX scope.

Table 3.3: Metrics calculation from hardware counters [33, 32].

Metric	Formula	Scope
CPI	PMCx076 / PMCx0C0	Core
DMPKI	PMCx043 / (PMCx0C0 / 1000)	Core
MPKI	L3PMCx06 / (PMCx0C0 / 1000)	Core
HPKI	(PMCx043 + PMCx071 + PMCx05A) / (PMCx0C0 / 1000)	Core
L3Lat	(L3PMCx90 * 16) / L3PMCx9A	CCX
L3Occ	QOS L3 Occupancy	Core
rBW	((L3PMCx06 * 64) / 2^{30}) / Time	Core

We use *AMD64 Technology Platform Quality of Service Extensions*(AMD QoSE) [32] to monitor and enforce limits on the amount of SLLC and *read* bandwidth available to each thread. AMD-QoSE is similar to Intel-RDT technology explained in Section 2.3.3.

There are specific banks of MSRs registers, belonging to *AMD64 Technology Platform Quality of Service Extensions* (AMD QoSE) [69] devoted to monitoring and enforcing limits on LLC allocation and memory *read* bandwidth on a per-thread basis. This is achieved with a 16-bit per-thread binary mask. Each bit of the mask enables each thread to use a particular sixteenth fraction of the LLC (1/16 = 1 MiB). Several threads can mark the same fraction(s), implying competitive sharing of the same LLC subset[1]. Similarly, the memory *read*

[1]Changing the space allocated in the SLLC does not have to imply a change in associativity,

bandwidth can be limited per thread. This is achieved by writing an unsigned integer to a specific MSR register that sets a maximum read bandwidth in 1/8 GB/s increments[2].

3.2.4 Workloads

The workload involves a subset of the single-threaded, memory-intensive applications from SPEC CPU2006 and CPU2017. 33 applications have been selected, as suggested in [114]. As in Section 2.3.5, both suites have been compiled following the official documentation and multi-thread applications have not been considered [118, 119].

We will first characterize each application running alone on the system and then in a variety of multiprogrammed situations. For the latter, we generated one hundred *mixes*, which we use in the characterization and evaluation sections, made up of applications from the SPEC subset. In addition, we generated another ten different *mixes* that we use in Section 3.3 to adjust the parameters of our proposal.

Each mix consists of 64 instances (one for each core) randomly chosen from the 33 applications, so that any of the 33 applications have the same probability of occurrence, regardless of their execution time. For each application, we randomly selected a *reference* input data set from among those offered by SPEC (excluding *test* and *train* input data sets). Hence, zero, one, or more instances of the same application, with the same or different inputs, may appear in a particular mix.

During a mix execution, each application/input pair is pinned to a different core. A mix ends when its slowest application finishes its first execution (e.g. thread D in Figure 3.2). The rest of the threads are restarted with the same input and on the same core as they finish. Only *complete* executions are taken into account; this avoids over-representing the first phases of an application. In addition, in order to characterize each application, its behavior is first averaged within each mix, considering all the cores in which it appears and all the instances running on them; the same application in another mix with different partners maybe executed a different number of times in a different number of cores, and might behave in a different way.

Consequently, to obtain the metrics of an application we proceed as follows: first for each mix, compute the average of each application (all the completed instances in all cores); then for each application compute the average again over the values obtained across the one hundred mixes.

This method of mixing and collecting results has been used, except for the variation in inputs, in previous works [170, 96, 172].

---

unlike in Intel Skylake.

[2]The bandwidth restriction in Intel is from L2 to LLC/main memory and in AMD it is from LLC to main memory.

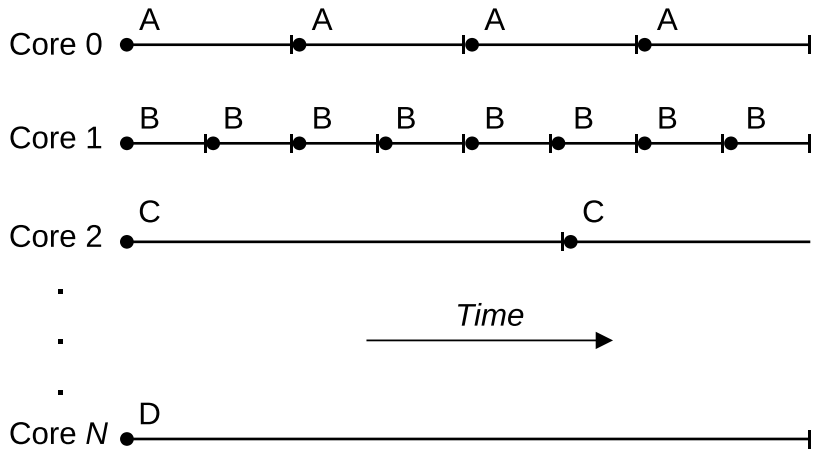


Figure 3.2: Example of aplications completions in a mix execution.

3.2.5 Metrics

Regardless of whether the applications are run alone or in a multiprogrammed way, we use seven metrics to characterized them: 1) CPI, cycles per instruction, 2) DMPKI, demand misses in the SLLC per thousand instructions, 3) MPKI, total misses (demand + prefetch) in the SLLC per thousand instructions, 4) HPKI, total hits (demand + prefetch) in the SLLC per thousand instructions, 5) L3Lat, memory latency of SLLC read misses, common to the four CCX cores, 6) L3Occ, average SLLC occupancy, and 7) rBW, read traffic with memory, in GB/s. Table 3.4 summarizes the main metrics used in this chapter.

Table 3.4: Used metrics.

Name	**Acronym**	**Definition**
Cycles per Instruction	CPI	Cycles / Retired Instructions
Instruction per cycles	IPC	Retired Instructions / Cycles
Demand Misses per Kilo Instruction	DMPKI	Demand Misses / (Retired Instructions ÷ 1000)
Misses per Kilo Instruction	MPKI	Misses / (Retired Instructions ÷ 1000)
Hit per Kilo Instruction	HPKI	Hit / (Retired Instructions ÷ 1000)
Bytes per Kilo Instruction	BPKI	Bytes / (Retired Instructions ÷ 1000)
Read Bandwidth	rBW	(LLC Misses · 64) ÷ 2^{30} / Execution Time
LLC Latency	L3Lat	Average LLC miss latency
LLC Occupancy	L3Occ	Average LLC occupancy

As a performance metric to compare two shared resources control mechanisms, we use the speedup relative to a baseline system.

To assess execution fairness, we use the metric M_1 defined by Kim et al [95]. Unlike other metrics such as the harmonic CPI, M_1 is a pure unfairness metric, independent of performance rewards. For a mechanism *mec* controlling the execution of a mix of applications, M_1 is calculated as:

$$M_1(mec) = \sum_i \sum_j \left(\left| \frac{IPC\_sta_i}{IPC\_mec_i} - \frac{IPC\_sta_j}{IPC\_mec_j} \right| \right)$$

where IPC_mec_i is the IPC of application i when running in a system controlled by the *mec* mechanism, and IPC_sta_i is the IPC when running in the baseline system.

3.3 Characterization

In the first two subsections, we analyze the relationship between the performance of each application running alone and the availability of LLC space and memory read bandwidth. In the third subsection, we study the use that applications make of both resources when running in a multiprogrammed context, modeled with the mixes discussed above.

3.3.1 Performance vs. SLLC Capacity

In this subsection, we characterize the behavior of the applications *running alone* when varying the SLLC capacity from 0 to 16 MiB with 1 MiB steps, with hardware prefetching enabled.

Figure 3.3 shows in the graphs, from left to right, CPI, DMPKI, and MPKI for each allocated SLLC capacity. Each graph shows three lines corresponding to `410.bwaves` (blue), `471.omnetpp` (green), and `554.roms` (red), representative of the three main trends observed in all applications.

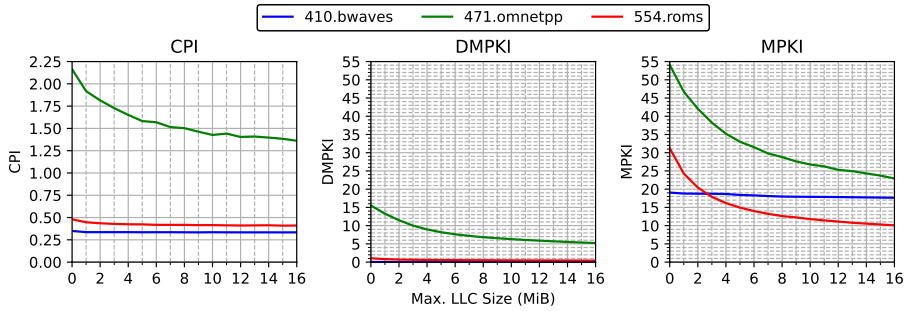


Figure 3.3: CPI, DMPKI, and MPKI for increasing LLC allocation limits (1/16 MiB steps).

Most cache partitioning mechanisms proposed so far use miss and speedup curves, similar to those of CPI and DMPKI shown in Figure 3.3, to decide how

much SLLC space should be allocated to each application [134, 44, 53]. In these two graphs we can distinguish two behaviors. On the one hand, `471.omnetpp` clearly takes advantage of its available space in the SLLC: CPI and DMPKI decrease significantly as the allocated space in the SLLC increases. We can say that the behavior of `471.omnetpp` is sensitive to the available SLLC size.

In contrast, `410.bwaves` and `554.roms` clearly waste the space they take in the SLLC. Both metrics remain virtually constant as the SLLC allocation bounds extend. We would say that the behavior of these two applications is insensitive to the available SLLC size. Consequently, partitioning mechanisms based on these metrics would take cache space away from applications such as `410.bwaves`, and `554.roms` and give more space to applications such as `471.omnetpp`.

But if we analyze the MPKI graph, the miss rate considering both demand and prefetch requests, while `471.omnetpp` and `410.bwaves` maintain the same behavior, `554.roms` now shows a large decrease in MPKI as the available SLLC space increases. Therefore, the MPKI metric would lead us to classify `554.roms` as sensitive to SLLC size, contrary to the DMPKI and CPI metrics. This behavior is due to the prefetcher being effective in preloading the private caches with the data to be used, which eliminates demand misses. In other words, the prefetcher reuses the data stored in the SLLC and, therefore, the more capacity the SLLC has, the higher the hit rate. As a result, giving more SLLC capacity to this type of applications does not imply a direct benefit for them but it does for the system, since it decreases the traffic with memory.

As far as we know, the behavior observed in `554.roms` with respect to the MPKI metric has not been previously highlighted and therefore it has not been considered when designing resource allocation mechanisms. Balancer will consider these applications as being cache sensitive and therefore will not limit their available SLLC space so as not to increase bandwidth consumption. As we will show in Section 3.4.2, higher bandwidth consumption can increase memory access latency, which in turn implies a performance degradation of all applications running on the system.

The CPI or DMPKI curves may show reasonable values and/or very little variation around small SLLC capacities. However, sometimes this insensitivity to size is hiding very noticeable prefetching performance. For these applications, keeping small capacities does not affect their performance, but it does degrade system performance. Considering the effect of prefetching (MPKI or HPKI) to guide SLLC quotas therefore seems more appropriate.

Characterization highlights: The MPKI metric, or its complementary HPKI, measures the benefit associated with SLLC occupancy more comprehensively than CPI or DMPKI, which are the metrics commonly used in previous work. The variation in CPI or DMPKI only reflects the benefit that affects the application itself, while MPKI or HPKI, in addition, reflects

the benefit that is achieved for the system.

3.3.2 Performance vs. Memory Bandwidth

Next, we characterize the behavior of applications *not running alone* with increasing throttling of the available memory bandwidth or, in other words, as memory contention grows due to increasing aggregate traffic, in the AMD Rome. For this purpose, we have used the Triad application.

Triad belongs to the STREAM benchmark [102]. STREAM is considered as the de-facto benchmark to measure sustainable main memory bandwidth. It is a simple synthetic program that we use to generate data traffic between the CPU and main memory.

Triad performs simple operations on vectors: $A[j] = B[j] + scalar * C[j]$. Vectors A, B, and C are larger than L2 + SLLC to ensure that there is no data reuse. We ran each of the selected applications alongside 0, 3, 7, 15, 31, and 63 Triad instances. The application to be characterized is pinned on the first core of the first CCX, and each Triad is pinned on another core, trying to occupy the maximum number of CCDs with an even split between CCXs. This thread scheduling runs the application on the AMD EPYC 7702P in configurations that first increase the number of active CCDs (from one to four and eight), then increase the number of active CCX per CCD (from one to two), and finally the number of cores per CCX (from one to two and four), see Table 3.5.

Table 3.5: Activation of AMD EPYC 7702P components according to the number of Triad instances that are executed together with the application to be characterized. Recall that this processor integrates 64 cores organized in eight CCDs, each with two CCXs, which in turn have four cores each.

# Triads	# active CCD	# active CCX per CCD	# active cores per CCX
0	1/8	1/2	1/4
3	4/8	1/2	1/4
7	8/8	1/2	1/4
15	8/8	2/2	1/4
31	8/8	2/2	2/4
63	8/8	2/2	4/4

Figure 3.4 shows read traffic with main memory (gray bars, left Y-axis) and memory latency (black bars, right Y-axis), both averaged for all applications vs. the number of co-executed Triads (X-axis). The read traffic saturation point is reached at around 105 GB/s with only seven Triad instances. However, latency continues to grow as the number of Triad instances increases beyond seven, from 378 to 676 cycles with equal data traffic.

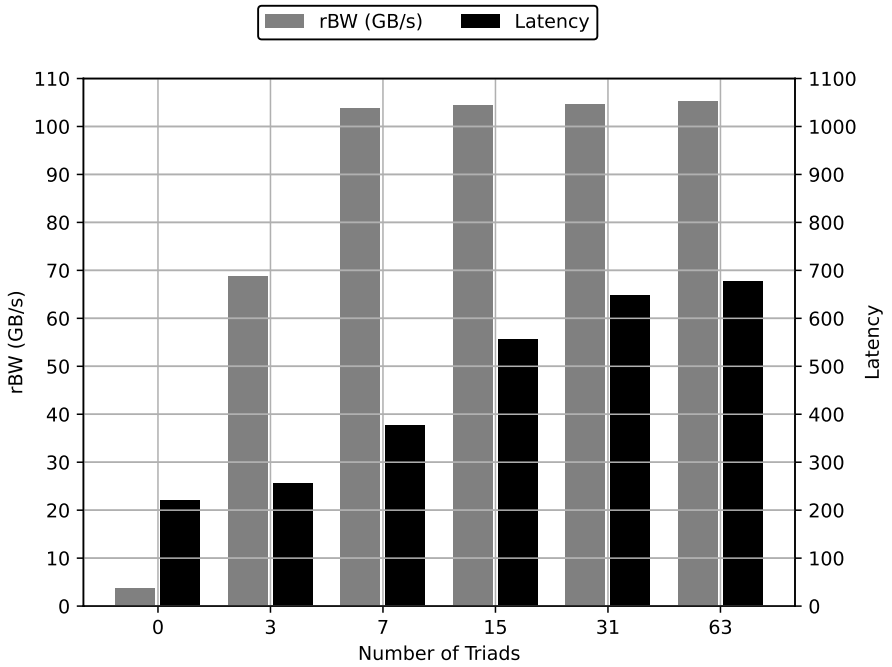


Figure 3.4: Read memory traffic (left Y-axis) and memory access latency (right Y-axis) vs. number of Triads (X-axis).

Figure 3.5 shows the performance impact of increased latency. For each application it can be seen how its CPI increases as memory contention grows. On average, the progressive growth of traffic induces an increase in execution time of 5, 20, 34, 49, and 61% when an application contends with 3, 7, 15, 31, and 63 Triads, respectively. In 5 applications, increases of more than 80% are observed when co-executing with 63 Triads.

Characterization highlights: Memory access latency is a better indicator than memory traffic for assessing memory contention. The increase in latency is a more direct measure of the impact on application execution time. In addition, memory latency increases if the request rate increases beyond the point of traffic saturation, allowing different degrees of contention to be identified.

3.3.3 Multiprogrammed Workload

Finally, we analyze the behavior of the applications on a fully loaded system, *running one application on each of the AMD Rome 64 cores*. Each application appears about two hundred times in the one hundred mixes, and is run

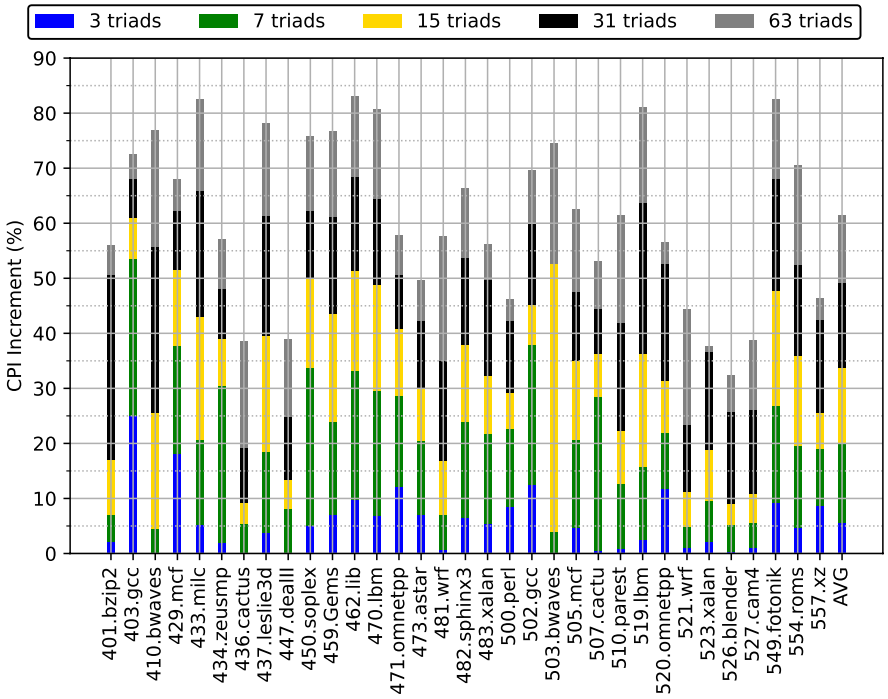


Figure 3.5: CPI increase when running with multiple Triads.

under different conditions, determined by the other sixty-three co-executing applications.

Figure 3.6 shows the averages of the metrics (CPI, DMPKI, etc.) for those two hundred instances, along with a vertical bar linking the minimum and maximum values. Average memory latency is between 500 and 600 cycles for all applications, reaching peaks of 650 for most of them. This indicates that memory traffic is always well above the saturation point.

Regarding SLLC occupancy, application behavior is very diverse. Eight applications occupy on average more than 6 MiB, three of them reaching maximums of 12 MiB, while five other applications occupy on average less than 2 MiB. The variation among runs of the same application is also large. The difference between the minimum and maximum is greater than 3 MiB in 24 applications.

The benefit applications get from the space they occupy in the SLLC is also very diverse as noted in Section 3.3.1 and Section 2.4.2. Applications such as `410.bwaves`, `433.milc`, `434.zeusmp`, `462.libquantum`, and `481.wrf` waste the space they occupy in the SLLC since HPKI is practically zero in all their executions, regardless of the SLLC capacity they fill. On the contrary, applications such as `471.omnetpp`, `403.gcc`, `429.mcf`, `450.soplex`, and `473.astar` show

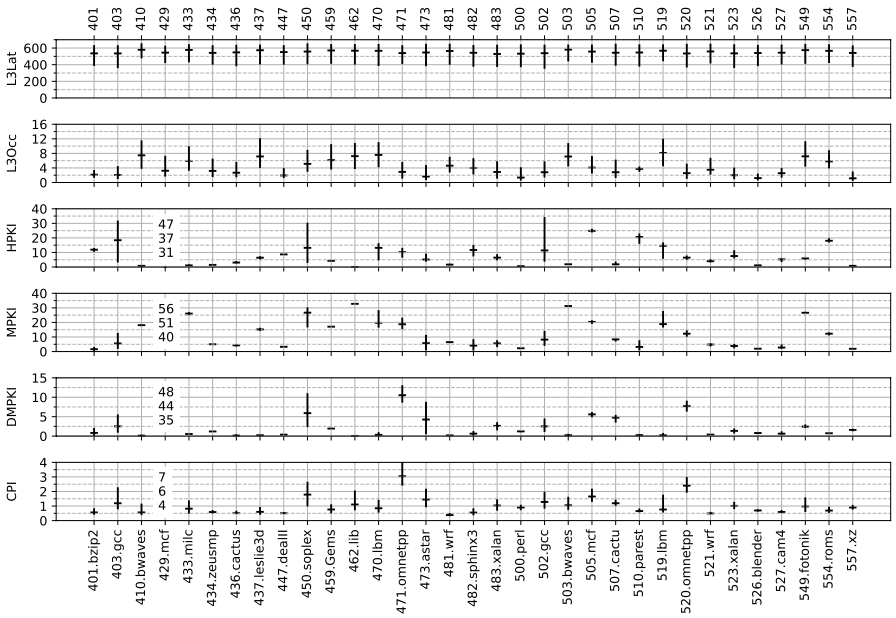


Figure 3.6: CPI, DMPKI, MPKI, HPKI, L3Occ, and L3Lat. For each metric and application, the mean value and a vertical bar linking the minimum and maximum values are shown.

significant differences between maximum and minimum values in occupancy, CPI, MPKI, and HPKI. These applications take advantage of the space in SLLC in a clear way, reducing their execution time if they get more space. Finally, in applications such as `436.cactusAMD`, `437.leslie3d`, `519.lbm`, `549.fotonik3d`, and `554.roms` we note significant variations in SLLC occupancy that do not translate into CPI and DMPKI differences, but do translate into MPKI and HPKI differences. Therefore, these applications do not get a direct benefit by occupying more space in the SLLC, but they can bring a benefit to the system by reducing the traffic with the main memory.

Characterization highlights: When running multiprogrammed workloads, it is common for the traffic generated by memory requests to congest the DRAM channels. This results in high memory latencies, which in turn affects application execution time. The SLLC space occupied by applications is very diverse and varies a lot among executions. The behavior reported in Section 3.3.1 and Section 2.4.2 is repeated when running applications on multiprogrammed workloads.

3.4 Balancer

This section introduces Balancer, a set of new mechanisms for allocating shared resources to the cores of a multicore processor. The first one, CCO (Control of SLLC Occupancy), manages the sharing of space in the SLLC. The second, CMT (Control of Memory Traffic), manages the amount of read memory bandwidth. Both can be tuned to act together in SLLC occupancy and read bandwidth (CCO+CMT).

Unlike other proposals, Balancer can be easily applied to clustered organizations because it can make different decisions in each cluster in a decentralized manner, in response to their particular cache utilization and bandwidth consumption.

These mechanisms have been implemented on a server based on an AMD 7702P processor. The scripts have been developed in Python3, and are executed in user space. A specific thread, called Balancer, executes such scripts every second. We define epoch as the time interval that elapses between two executions of the scripts triggered by Balancer. First, the monitored events recorded in the hardware counters are read, and the metrics of interest are calculated. The last ten values of these metrics are then averaged. This cal- culation prevents one-off peaks from triggering the imposition of constraints, and also prevents events in the distant past that are no longer relevant from affecting the values of the metrics. The Balancer thread is pinned to the first core of the first CCX, being this core the only one that has hyperthreading enabled. The other cores run their applications with hyperthreading disabled and zero overhead due to the execution of Balancer. This spatial decision to assign the Balancer thread to one or another CCX is irrelevant since its overhead is absolutely negligible. We have measured with perf the CPU time consumed by the Balancer thread (monitoring + control). This time represents 0.1% of the execution time of all system cores.

3.4.1 Control of SLLC Occupancy (CCO)

Motivation. SLLC space can become a scarce resource if there is competition among applications. Considering the overall benefit to the system, it would be desirable to allocate more space to the applications that can take the most advantage of it. As discussed in Section 3.3.1, monitoring of HPKI and MPKI metrics can identify such applications. The idea is to foster a good reuse of the data stored in the SLLC to improve the performance of individual cores and, in the process, generate less main memory traffic. We calculate the reuse ratio as the number of hits that each block fetched to LLC receives, i.e. $Hits_{LLC}/Misses_{LLC}$. However, different applications may need more or less LLC space to achieve the same reuse ratio. It would be desirable to give priority to those applications that achieve a higher reuse ratio while occupying less LLC space.

Proposal. We define a new metric, the *number of hits per miss and MiB occupied* (HpMO), to compare the LLC space usage efficiency of applications. This metric is directly proportional to the reuse ratio and inversely proportional to the space occupied in LLC. Hit, miss, and occupancy rates (HPKI, MPKI, and L3Occ) are sampled in each CCX SLLC. For each core of a CCX the *number of hits per miss and MiB occupied*, HpMO, is calculated as follows:

$$HpMO = \frac{Hits_{SLLC}}{Misses_{SLLC} \cdot Occupancy_{SLLC}}$$

HpMO quantifies for each core the profit obtained with the cache blocks it has in the SLLC, either evicted from L2 or brought by prefetching. The higher the HpMO, the better the space utilization in the SLLC and vice versa. For example, a core with a 20% reuse rate and 5 MiB occupancy has a very low HpMO value of 0.05, evidencing little benefit from SLLC occupied space.

Balancer considers that a core is using the space it occupies in LLC inefficiently if its HpMO is less than a certain threshold X. In this case, the core will be *constrained* in a shared 1 MiB partition. This SLLC partition is shared by all cores, constrained or not. Consequently, the LLC size occupied by all restricted cores is at most 1 MiB, leaving the other 15 MiB for the exclusive use of the unrestricted cores. In this way, CCO allows other applications to take advantage of the space left over in the SLLC, reducing MPKI and main memory traffic. Figure 3.7.a shows the CCO control algorithm for one core, which applies to all cores in the system every epoch. The proper values for the HpMO threshold depend slightly on the mechanism target, performance or fairness, and will be studied in subsection 3.5.2.

In the unlikely event that there is only one unconstrained core in a CCX, no new limits will be imposed. Finally, constraints on a core are removed when a phase change is detected. We assume that a phase change has occurred when the behavior of the program with respect to the LLC has experienced a significant change, which justifies Balancer to re-evaluate the metrics of interest. Therefore, the metric to detect a phase change must be related to LLC, since a change in other metrics such as branch predictor misses or L1D accesses might not correlate with a significant change with respect to LLC. We used total requests (demand and prefetch) instead of misses because they are not affected by external behaviors, e.g.: change in LLC partition or phase change in other applications. Balancer considers that a phase change exists if the number of SLLC prefetch and demand requests per kilo instructions ($HPKI + MPKI$) of a core differs by more than 20% from the previous measurement.

3.4.2 Control of Memory Traffic (CMT)

Motivation. As we have seen in Section 3.3.2, memory latency is a good proxy of communication contention between the executing cores and main memory. That is, when memory traffic reaches the limit supported by the system, the increased rate of requests to main memory translates into an increase in memory

latency which, in turn, affects the performance of running applications, see Figure 3.4.

Proposal. Balancer considers that memory access congestion exists if the memory access latency exceeds a certain threshold Y. Figure 3.7 shows the CMT control algorithm for one core, which applies to all cores in the system every epoch. The average latency of the off-chip memory requests made by each CCX is monitored. If it exceeds the threshold Y, the read bandwidth consumption of each core in that CCX is examined. The core responsible for the largest one is selected, and its bandwidth limited to 2.5 GB/s. If the selected core was already limited, its bandwidth limit is further decreased by 10%, until reaching a minimum that would correspond to an equal distribution between cores, i.e., 1/64 of 105 GB/s, the peak rBW measured. Again, the proper values for the CCX latency threshold depend slightly on the mechanism target, performance or fairness, and will be studied in Section 3.5.2.

As in CCO, constraints on a core are removed when a phase change is detected.

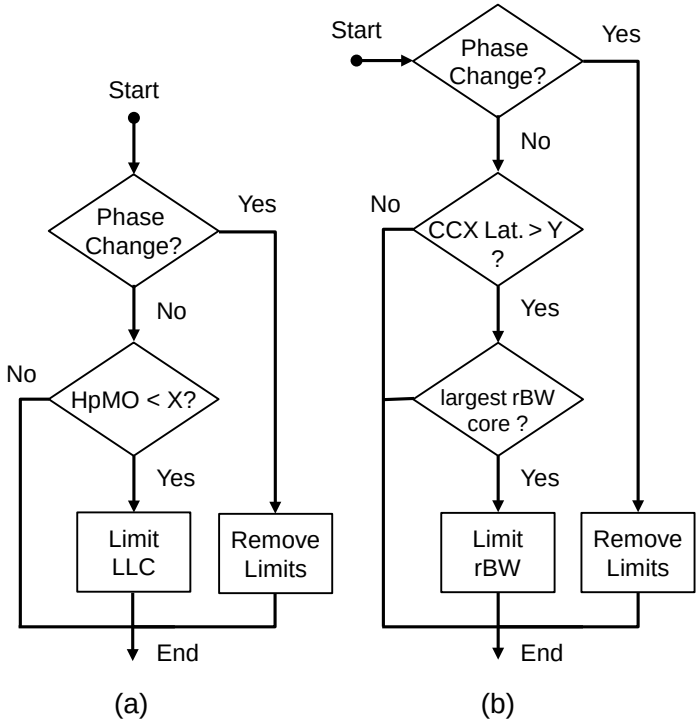


Figure 3.7: COC (a) and CMT (b) control algorithms.

3.4.3 Balancer: Simultaneous Control of SLLC Occupancy and Memory Traffic (CCO+CMT)

CCO and CMT can be combined to obtain better performance than that achieved by each mechanism separately. CCO is conservative in that it limits occupancy to cores that waste SLLC, in the hope that the freed capacity will be leveraged by the remaining cores to improve overall performance. In contrast, CMT is aggressive in that it limits traffic in applications that consume high memory bandwidth, even though it may be contributing to good performance. Therefore, we propose combining both controls, but first applying CCO, to gradually improve SLLC occupancy in successive epochs and then CMT. Thus, when a control epoch starts, firstly CCO acts: the HpMO of each core is compared with threshold X in each CCX, and the cores without SLLC occupancy limits that are below the threshold are confined in the 1 MiB partition. Secondly, CMT will act as explained only on those CCXs that have not experienced new confinements, i.e. on CCXs with latency above threshold Y the highest traffic core is selected and its bandwidth limited.

3.5 Evaluation

In this section, we evaluate our proposal (Balancer), and compare it with a system without control (Uncontrolled) and with three control mechanisms using cache partitioning: 1) equal sharing of resources through static allocation, i.e. 4 MiB of SLLC and 1.6 GB/s bandwidth per core (Static), 2) static UCP guided by DMPKI (UCPd), and 3) static UCP guided by MPKI (UCPm). UCP stands for Utility-based Cache Partitioning [134].

3.5.1 Metrics and Baseline System

As a baseline system we will use Static, which is positioned at one extreme of the performance/fairness tradeoff: it does not allow dynamic sharing (i.e. surplus hardware resources cannot be exploited by cores with scarcity), and it is intrinsically fair in terms of hardware resources (i.e. equal resource partitioning). Of course, reporting values relative to the baseline system does not exclude mutual comparison between the rest of the mechanisms, but it facilitates discussion when focusing on individual application behaviors, which in some metrics have very different absolute values.

3.5.2 Design Space Exploration

For decision making Balancer uses two thresholds on HpMO and CCX latency values, Sections 3.4.1 and 3.4.2, respectively. To analyze their impact on performance and fairness, we ran ten new mixes under Balancer control using a set of thresholds for HpMO (from 0.03 to 0.07 with steps of 0.01) and CCX latency (from 300 to 450 with steps of 50). Figure 3.8 shows the average value of Speedup (Y-axis) and M1(X-axis) obtained by each configuration across all the

mixes in our workload. The results of each evaluated configuration are shown with the combination of a shape and a color. The shapes specify the latency thresholds while the colors indicate the HpMO thresholds. For example, a yellow square represents the speedup and unfairness for a configuration with thresholds of 0.05 and 400 cycles for the HpMO and latency thresholds, respectively. The gray shapes show the results of CTM-only Balancer while the circles show the results of CCO-only Balancer. The horizontal and vertical dashed lines represent the speedup and M1 results, respectively, for Uncontrolled.

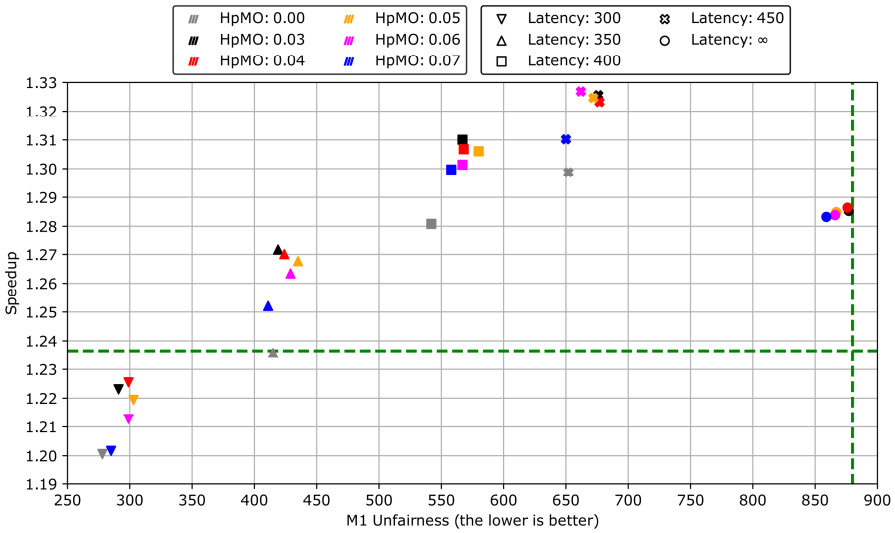


Figure 3.8: Speedup and unfairness for Balancer with different thresholds. Different colors and shapes represent results for different HpMO and latency thresholds, respectively. Green dashed lines correspond to an Uncontrolled system.

Variations in the CCX memory latency threshold significantly affect fairness and performance. Reductions in the latency threshold improve fairness, as the CMT mechanism imposes more traffic constraints on cores, approaching an equal sharing of memory bandwidth among all cores. CMT always achieves drastic reductions in unfairness, although in some configurations it produces performance losses. In the best case, CMT manages to divide the M1 metric by 3.17 with respect to Uncontrolled.

On the other hand, variations in the HpMO threshold affect performance but have negligible effects on fairness. CCO always has a positive impact on performance. It achieves a speedup varying between 0.1 and 4.0% with a small loss of fairness.

By combining CCO and CMT and properly selecting thresholds, the Balancer resource control can be directed to different targets, matching the performance/-fairness tradeoff at will. For their detailed evaluation with the one hundred

mixes cited in Section 3.2.4 we select two control solutions, Balancer-P and Balancer-F, see Table 3.6. *Balancer-P* is intended to optimize performance, uses values of 0.06 and 450 for the HpMO and latency thresholds respectively, and achieves a speedup of 7.01% with respect to Uncontrolled, with a 32.9% reduction in unfairness. *Balancer-F* is designed to optimize fairness, uses values of 0.04 and 300 for the HpMO and latency thresholds respectively, and achieves a 66.0% reduction in unfairness, although it produces a 1.2% performance loss with respect to Uncontrolled. As an example of minimum unfairness we have discarded the CMT-only system (gray inverted triangle in Figure 3.8), because it achieves a slight reduction in unfairness but with a significant loss of performance.

Table 3.6: Balancer-P and Balancer-F thresholds.

Metrics	Balancer-P	Balancer-F
HpMO, hits/(misses·MiB)	X = 0.06	X = 0.04
CCX mem. latency, cycles	Y = 450	Y = 300

3.5.3 Performance

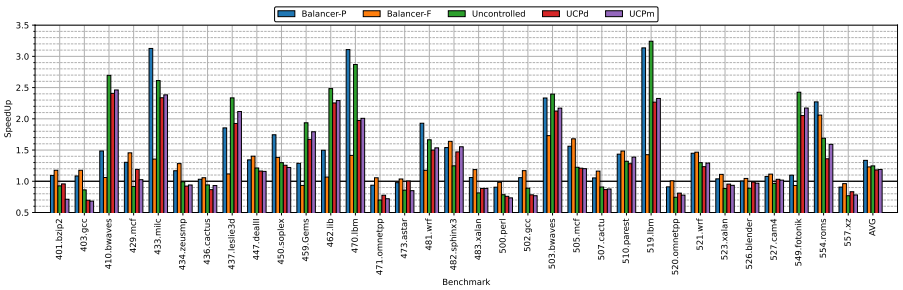


Figure 3.9: Speedup of the selected SPEC CPU2006 and CPU2017 applications for all control mechanisms relative to Static.

Figure 3.9 shows for each application the speedup obtained by all mechanisms with respect to the Static baseline system. The last group of bars on the right shows that, on average, all mechanisms improve performance against Static. However, only Balancer-P outperforms Uncontrolled. Balancer-P focuses on improving overall system performance. Since it limits bandwidth to those cores that cause high memory latencies, it causes 7 applications out of 33 to lose performance with respect to Uncontrolled, being `549.fotonik3d_r` the worst case with a 0.45 slowdown. However, Balancer-P outperforms Uncontrolled in 14 out of 18 CPU2006 applications and 12 out of 15 CPU2017 applications, and achieves an average improvement of 7.1%, with a maximum speedup of 42.4% on `429.mcf`.

Balancer-F, on the other hand, improves fairness by more aggressively limiting main memory traffic, resulting in performance losses relative to Uncontrolled in 10 applications out of 33 selected. Yet Balancer-P outperforms Uncontrolled in 11 out of 18 SPEC CPU2006 applications and in 12 out of 15 SPEC CPU2017 applications, with average performance only 1.3% worse.

Notice that Balancer-P slows down five applications with respect to Static (`471.omnetpp`, `473.astar`, `500.perl`, `520.omnetpp`, `557.xz`). These are *weak* applications in the sense that they take up few resources when in competition with other applications, as can be seen from the L3Occ data in Figure 3.6. Therefore, these applications achieve better performance if they are allocated 4 MiB of LLC without competition. The two Balancer configurations lose the least with respect to Static, with quite a difference in some cases with respect to all other mechanisms. In other words, Balancer manages to protect these *weak* applications better than the other mechanisms.

3.5.4 Fairness

Table 3.7 shows the average value of M_1 obtained by each mechanism for all the mixes in our workload. Note that M_1 is an unfairness metric and therefore the lower the better.

Uncontrolled presents the highest value of unfairness (M_1=851). This is an expected result as each application uses the resources it needs without regard to the impact on other applications. UCPd and UCPm manage to reduce the unfairness by 25% and 18% with respect to Uncontrolled, but at the cost of a significant loss of performance, as we have seen in the previous subsection. Balancer-F, on the other hand, achieves a much larger reduction in unfairness, 64.5% less than Uncontrolled, with a performance loss of only 1.3%. Even the performance optimized version, Balancer-P, manages to reduce the unfairness to values similar to those of UCPd and UCPm with a very significant performance improvement.

Table 3.7: Average M_1 execution fairness

Balancer-P	Balancer-F	Uncontrolled	UCPd	UCPm
693	302	851	634	698

An important benefit of improving system fairness is that execution time predictability increases. Less variability in the execution time of applications facilitates scheduling decisions and minimizes unexpected charges for services.

Figure 3.10 shows, for each application, the maximum, minimum, 75th, and 25th percentiles of the IPC values obtained for all application instances run in the different workload mixes. Values are shown for the two Balancer configurations and for Uncontrolled.

Balancer-F reduces the variability in IPC with respect to Uncontrolled in all applications except `502.gcc`. The reduction is very significant in many applications. As an example, with Balance-F, the difference between the 25th and 75th percentiles is less than 10.0% in 20 of the 33 applications, and greater than 20.0% in only 5 applications. With Uncontrolled, the difference is greater than 10.0% for all applications and greater than 20.0% for 23 out of 33. Balancer-P also reduces the variability compared to Uncontrolled in most applications, 26 out of 33, but to a lesser extent than Balancer-F.

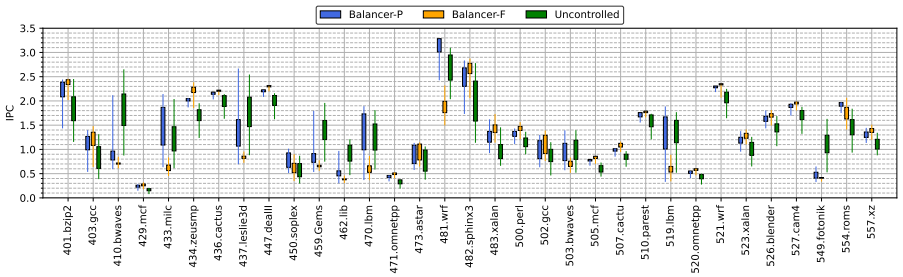


Figure 3.10: IPC variability: 75th and 25th percentiles and maximum and minimum values.

3.5.5 Number of Cores and Scalability

An alternative to limiting memory bandwidth is to decrease the number of cores used to run applications. By loading the system with a smaller number of applications, each application has a larger fraction of resources at its disposal.

To test the impact on system performance of leaving cores idle, we ran 10,000 SPEC CPU2006 and CPU2017 applications on our system using 64 cores (all active, one pinned application per core), 56 cores (1 idle core per CCD) and 48 cores (1 idle core per CCX). In all experiments, the Balancer thread is active, and when an application terminates and frees a core, the next application to run on it is the one with the lowest cumulative execution time. This ensures that all applications are represented uniformly, regardless of their individual running time.

Running the 10,000 applications on 64 cores took 15 hours 52 minutes, on 56 cores (87% of our processor's total capacity) it took 17 hours 27 minutes (10% more), and on 48 cores (75% of our processor's total cores) it took 18 hours 10 minutes (13% more). Therefore, we have not seen any performance improvement from leaving cores idle. In terms of performance it is best to keep the system at its maximum possible load.

Works such as García et al. [53] or Xiao et al. [172] use a maximum of 80% and 40% of the processor cores respectively to evaluate their mechanisms. García et al. uses 8, 12 and 16 cores to analyze the impact of the number of applications on fairness. Xiao et al. uses only 8 cores because they do not

have enough masks to run their algorithm with a larger number of cores. Our algorithm does not have any limitation on masks. Nor have we perceived a performance improvement that would justify the use of fewer cores than the maximum available in our processor.

3.6 Related Work

Most previous proposals for shared resource control act on the SLLC to improve overall system performance or fairness [44, 170, 95, 172, 129, 26, 158, 174, 163, 96, 144, 53, 134, 130], or turnaround time [132]. Among them we highlight KPART [44], DCAPS [170] and LFOC [53]. They use demand miss rate and IPC curves, which are dynamically computed, to guide application clustering and SLLC partitioning. LFOC identifies in a first step the applications with IPC insensitive to the SLLC capacity and isolates them in a single partition with two cache ways. Balancer uses this same technique with two differences: i) LFOC uses DMPKI to guide this decision while Balancer uses he number of hits per misses and MiB occupied. This implies that in some cases LFOC may increase bandwidth consumption. ii) Balancer uses a single cache way partition to isolate applications because its selection algorithm is more restrictive than that of LFOC.

Some of these proposals also use memory traffic to drive their decisions [44, 160, 101, 23, 117]. However, only CoPart acts directly on this resource in addition to acting on SLLC [130]. Our proposal acts on both resources. So, CoPart is the proposal that most closely resembles ours. CoPart acts on the SLLC and memory traffic of a 16-core Intel Xeon to improve fairness. It first performs profiling to determine the sensitivity of each application to the slowdown it experiences when the cache or available bandwidth gets reduced. Then, applications are dynamically ranked according to whether they need more resources or are able to give up the ones they have. CoPart formulates SLLC and memory bandwidth allocation as an economy problem, where resources are reallocated from one application to another using a heuristic looking for maximum fairness.

The main problem of all these mechanisms is the complexity of the control/monitoring actions that classify the applications. An example is the use of the miss rate curve. Usually, the mechanisms require the execution of each application with different resource limits to obtain such behavior models. This classification has to be repeated periodically to detect application phase changes. Moreover, this problem is more critical as the number of cores increases. In contrast, our proposals monitor simple metrics to detect inefficient use of SLLC or excessive latency due to bandwidth abuse.

Another important limitation of these mechanisms is that their heuristics, when changing the allocation of resources to an application, only consider the achieved self-profit, without taking into account the impact on the system. On the contrary, the mechanisms proposed in Balancer identify inappropriate uses

of resources in order to prevent them and improve system performance.

On the other hand, other proposals only act on memory traffic [129, 171, 71, 41, 82, 108, 116]. EMBA limits the memory traffic of the applications that use more bandwidth [171]. It does this progressively and observes the impact on system performance. The mechanism stops increasing the limitations when it detects a loss in performance. In addition, it restarts when it detects a phase change in one of the applications. The mechanism is tested with only eight cores, and unlike our proposal, it does not control SLLC allocation.

PABST focuses on controlling memory traffic by restricting request rates and changing the priority of memory requests [71]. However, it is evaluated on a simulator, and requires extra hardware. In contrast, our proposal runs on real hardware. Other works that also use simulation instead of real hardware are [41, 82, 108, 116]. These works propose new schedulers for the main memory controller to improve system performance and/or fairness.

Finally, we have considered comparison with several of these state-of-the-art SLLC control mechanisms, but it has not been possible for several reasons. Most authors rely on closed source codes, making it very difficult to ensure that a third-party implementation is truthfully conforming to their approach. We are aware of only two works that provide open source codes, ElSayed et al. [44] and Pons et al. [132]. However, both mechanisms are intended for Intel processors with an inclusive SLLC and no clustering of cores and resources. In contrast the AMD organization uses non-inclusive LLCs grouped in CCXs serving clusters of cores, which compete for main memory bandwidth through two levels of routing, first the CCD I/O and then the die I/O. Therefore, adjusting these mechanisms to a hierarchical organization requires in-depth changes that go beyond code adaptation. For example, it is necessary to decide whether to establish a single control mechanism throughout the system or a mechanism per shared SLLC in every CCX. Moreover, it is necessary to find the AMD hardware counters, if they exist, equivalent to the Intel ones. In our attempt to port these mechanisms, we were unable to compile the code of Pons et al. and obtained unsatisfactory results when running that of ElSayed et al. For these reasons, we decided to leave them out of comparison.

3.7 Concluding Remarks

This chapter presents a detailed characterization of the execution of the subset of single-threaded, memory-intensive test programs of the SPEC CPU2006 and CPU2017 SPEC suites on an AMD Rome processor. This analysis focuses on the impact of available SLLC and memory bandwidth on the performance of an application. We have expanded on the knowledge of the uneven use of resources by applications show in chapter 2 identifying a type of application whose performance is barely affected when its allocated SLLC space decreases, but whose memory bandwidth consumption increases significantly, negatively affecting system performance. We have also found that memory access latency

is a better indicator than memory traffic for assessing memory access contention. As far as we know, this is the first time that these findings have been highlighted.

From the characterization work, we have proposed strategies that impose limits on SLLC space utilization and memory traffic to specific applications. These constraints improve performance and/or fairness of multiprogrammed workloads, on average, with respect to a system with no control. Specifically, Balancer-P, tuned for performance, improves IPC 7.1% and reduces unfairness 18.6% compared to the system without control, while Balancer-F, tuned for fairness, reduces unfairness 64.5% in exchange for a 1.3% loss in performance. Balancer requires no hardware or operating system modifications. Our proposal is the only one, to our knowledge, that controls SLLC occupancy and memory traffic on an AMD processor with 64 cores organized in clusters.

Chapter 4

Berti: an Accurate Local-Delta Data Prefetcher

As we state in Chapter 2, data prefetching is a technique that plays a crucial role in modern high-performance processors by hiding long latency memory accesses and improving performance. Several state-of-the-art hardware prefetchers exploit the concept of deltas, defined as the difference between the cache line addresses of two demand accesses. Existing delta prefetchers, such as best offset prefetching (BOP) and multi-lookahead prefetching (MLOP), train and predict future accesses based on global *deltas. We observed that the use of global deltas results in missed opportunities to anticipate memory accesses. In this chapter, we propose Berti, a first-level data cache prefetcher that selects the best* local *deltas, i.e., those that consider only demand accesses issued by the same instruction. Thanks to a high-confidence mechanism that precisely detects the timely local deltas with high coverage, Berti generates accurate prefetch requests. Then, it orchestrates the prefetch requests to the memory hierarchy, using the selected deltas.*

4.1 Introduction

Data prefetching techniques play an important role in hiding long-latency memory accesses. Hardware prefetchers learn memory access patterns and fetch data into the cache hierarchy before time so that future memory accesses get cache hits. Data prefetching techniques can be employed either at the private first-level data cache (L1D), second-level cache (L2), or at the shared last-level cache (SLLC).

Most of the recently proposed storage-efficient spatial prefetchers target L2 [104, 147, 9, 93, 13]. Exceptions are the multi-lookahead offset prefetching (MLOP) [145] and the instruction pointer classifier-based prefetching (IPCP) [9], which are L1D prefetchers. It is well known that an L1D prefetcher provides better performance than an L2 prefetcher as the prefetched lines are brought

into L1D and not till L2. In addition, an L1D prefetcher sees unfiltered memory access patterns and can predict the future accesses better than an L2 or SLLC prefetcher. L1D also sees a sequence of virtual addresses as compared to physical addresses at the L2 and LLC, which can facilitate cross-page prefetching [54]. Also, compared to L1D, additional contextual information is not easy to propagate to L2 and LLC, such as instruction pointer (IP) [94], which is usually available at the L1D (e.g., Intel's IP-stride at the L1D [37]). However, designing a high-performance L1D prefetcher is hindered by (i) storage overhead, (ii) starved L1D bandwidth, (iii) L1D pollution because of inaccurate prefetching, and (iv) narrow scope for aggressive prefetching because of the limited size of the prefetch queue (PQ) and the miss status holding registers (MSHR).

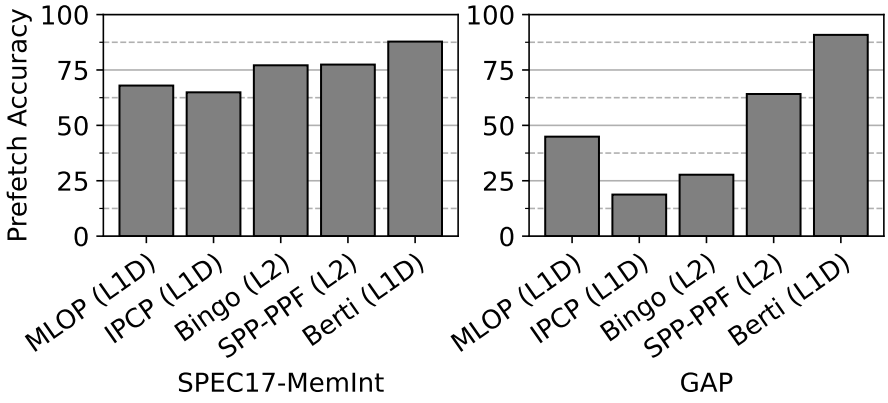
State-of-the-art data prefetchers push the limit of single-thread performance with average performance boosts of 3% to 5% [93, 13, 9, 92]. However, as shown in Figure 4.1(a), these prefetchers load a large number of useless blocks, ranging from 22.6% to 35.1% for SPEC CPU2017 and from 35.8% to 81.2% for GAP workloads, which results in sub-optimal performance and additional dynamic energy consumption [8]. Figure 4.1(b) shows that state-of-the-art prefetchers significantly increase the dynamic energy consumption at the memory hierarchy (caches and DRAM) up to 30.1% and 86.9% for SPEC CPU2017 and GAP workloads, respectively.

Our proposal, Berti, provides an accuracy of almost 90%, which translates into a dynamic energy overhead of only 9.0% and 14.3% for SPEC CPU2017 and GAP, respectively.

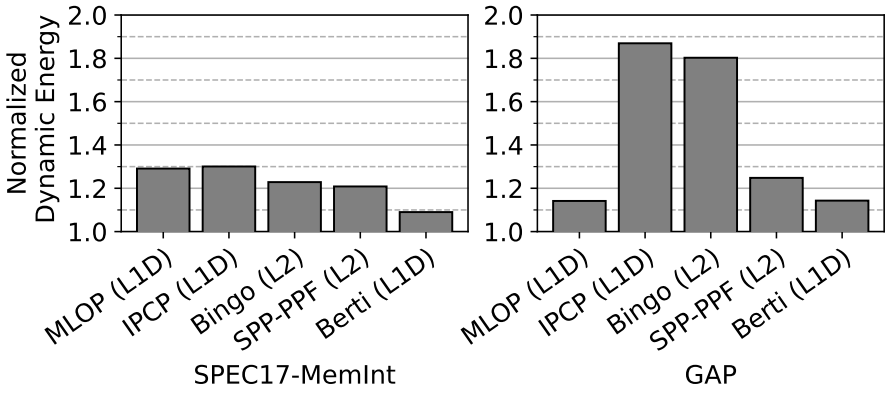
4.1.1 Our Approach

We ask the following simple question in designing our approach: "*for an L1D access to addresses X, what is the* timely and accurate *delta (d) that should be used for prefetching?*" The best offset prefetcher (BOP) inspires us to ask this question [104]. However, our approach is different from BOP and other offset prefetchers [145, 84]. Our key observation is that the best delta for access is dependent on the local contextual information, such as an instruction pointer (IP), and it varies based on the context (e.g., the best delta for IP_X is different from IP_Y). We argue that prefetching based on global (context-agnostic) deltas results in missing opportunities [121].

We propose Berti, a cost-effective, per-IP *best request time* delta L1D prefetcher that makes a strong case for timeliness and accuracy. For each IP, Berti learns the deltas that result in timely prefetch requests, and issue prefetch requests only for the deltas predicted to provide high coverage, which translates to overall high prefetch accuracy. Sited at the L1D and seeing all virtual addresses generated by the processor, Berti orchestrates the prefetch requests to the memory hierarchy. Our Berti prefetcher is inspired by Berti from DPC-3 [140].



(a) Accuracy of L1 and L2 prefetchers

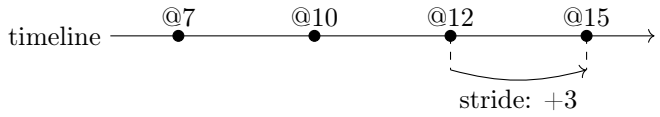


(b) Dynamic energy consumption normalized to no prefetching

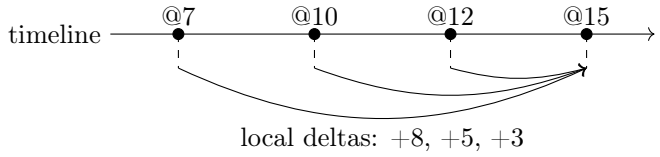
Figure 4.1: Prefetch accuracy and dynamic energy consumption of the memory hierarchy for state-of-the-art prefetchers (IPCP [122], MLOP [145], SPP-PPF [13], and Bingo [9]) averaged across single-threaded traces from memory-intensive SPEC CPU2017 [119] and GAP [10] workloads.

4.1.2 Accurate and Timely Local Deltas

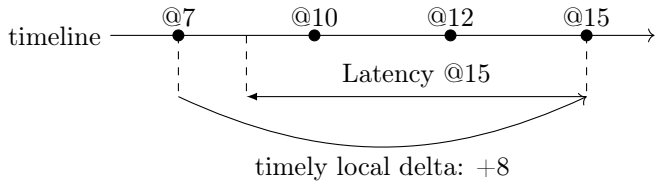
We define local delta as the difference in cache line adresses between two demand accesses that are issued by the *same IP*. The definition of delta differs from the definition of stride, being the later the difference between addresses of *consecutive* load accesses with the same IP. For example, an IP_X that accesses the following cache line addresses: X, X+2, X+4, X+6, sees a sequence of strides as follows: +2, +2, and +2. In this case, the stride is 2. However, access X+6 sees the following deltas: +6, +4, and +2. Figure 4.2 shows an example differentiating strides, local deltas, and timely local deltas. If the goal of a prefetcher is to *cover* addresses 15, then the prefetcher can initiate prefetching



(a) Stride



(b) Local deltas



(c) Timely local deltas

Figure 4.2: Strides, local deltas and timely local deltas. The values on the timelines (7, 10, 12 ...) represent the addresses referenced by the same instruction.

with deltas +3, +5, and +8 whenever it sees the demand accesses to adresses 12, 10, and 7, respectively. However, if we consider time to prefetch adress 15, then deltas of +3, and +5 will not completely mitigate the L1D miss latency, as they will be late prefetch requests. Instead, if a prefetcher issues a request for adress 15 with deltas of +8 on demand accesses to adress 7, respectively, it can prefetch adress 15 well ahead of time.

With Berti, we find the *timely* local deltas, and compute its respective coverage. We prefetch using deltas that used to show high coverage, which translates to overall high prefetch accuracy as we show in this work. We call these deltas the accurate and timely deltas.

The rest of the chapter is organized as follow. Section 4.2 motivates the need for a new prefetcher. Berti is detailed in Section 4.3. Section 4.4 describes the experimental methodology: simulation framework, energy model, workload and baseline. In Section 4.5 the evaluation of Berti is presented. Section 4.6 comments related research. Finally, the chapter ends with the concluding remarks, Section 4.7.

4.2 Recent Works and Motivation

4.2.1 Recent Advances in Data Prefetching

Data prefetching plays an important role in designing high performance processors. Recent developments in this field mainly come from the last two data prefetching championships, DPC-2 [3] and DPC-3 [4], co-located with ISCA 2015 and ISCA 2019, respectively.

Best offset prefetching (BOP)

The winner of DPC-2 is a degree-one L2 prefetcher that finds an offset that provides the maximum likelihood of future use at the L2 cache [104]. An offset of k means that a cache line is k cache lines away from the current demand addresses. BOP takes timeliness into account while selecting the best offset per application phase. Multi-lookahead offset prefetching (MLOP) [145] is an extension on BOP that is motivated by Jain's Ph.D. thesis [84]. MLOP considers multiple lookaheads for each offset and selects the offset and lookahead covering a specific cache miss. Both BOP and MLOP treat the demand addresses in isolation, and for each demand access, trigger prefetch requests based on the prefetch offset[1]. In general, MLOP provides better prefetch coverage than BOP.

Variable Length Delta Prefetching (VLDP)

This spatial data prefetcher uses multiple histories of deltas between successive cache lines observed within an operating system (OS) page to predict the future memory accesses in other OS pages [147]. One of the key features of VLDP is that it uses multiple prediction tables and makes predictions based on different lengths of history in terms of deltas.

Signature path prefetching (SPP)

This state-of-the-art delta prefetcher predicts irregular strides at the L2 cache [93]. SPP works by relying on the signatures (hashes of consecutive strides) observed within an OS page to index into a table that predicts future deltas. SPP uses a lookahead mechanism that recursively finds out deltas to prefetch until a delta falls below a *confidence*. Perceptron prefetch filtering (PPF) is a filter that further improves the effectiveness of SPP by deciding whether to prefetch into L2 or not [13]. In general, SPP combined with PPF (SPP-PPF) provides better prefetch coverage than VLDP.

Bingo

This L2 prefetcher makes a case for associating spatial access patterns to both short (such as IP) and long events (such as IP, IP+offset, and memory region)

[1]For BOP and MLOP, we use the term *global delta* instead of *offset* for the rest of the chapter.

and selecting the best pattern for prefetching [9]. A key point of Bingo is the use of only one hardware table for both short and long events. This table enables multiple predictions from a single entry, providing better coverage than single-event prefetching. In general, Bingo outperforms VLDP and SPP-PPF for SPEC CPU2017 traces. However, it requires significantly more storage than VLDP and SPP-PPF.

Instruction pointer classifier prefetching (IPCP)

The winner of DPC-3 is a state-of-the-art L1D data prefetcher that is composite in nature [122]. It classifies an IP into three classes: constant stride (CS), complex stride (CPLX), and global stream (GS). IPCP uses three lightweight prefetchers that issue prefetch requests according to the IP class. If it fails to classify an IP into one of the three classes, it uses a next-line prefetcher.

4.2.2 Motivation: Why a New Delta Prefetcher?

Why not a global delta prefetcher?

We observe that finding the best delta for an entire application results in missing opportunities because the best delta varies based on the program context, e.g., an IP or the OS page. Figure 4.3 shows the best deltas selected by BOP (red line) and Berti (gray lines) for different IPs of the `mcf-1554B` benchmark. We can see that the best delta is different for distinct IPs making a strong case for prefetching local, local deltas instead of a global best delta (oblivious to per-IP best deltas). We can also see that the global delta (+62) as selected by BOP does not cover all cache accesses, and it is not the best delta. For `mcf-1554B`, BOP provides coverage of only 2%, whereas Berti, that selects local deltas (per IP), provides better coverage as shown in Figure 4.3. As we will see in Section 4.5, the use of a global delta may be beneficial in some cases (e.g., in `CactuBSSN`), but this is not the common case.

Why not existing local L1D prefetchers?

A conventional IP-stride prefetcher covers consecutive constant strides and not necessarily timely deltas. For example, IP `0x401cb0` from `lbm-2676B`, Figure 4.4, generates the following stride sequence: +1, +2, +1, +2, ... +1, +2. For this pattern, an IP-stride prefetcher will provide zero coverage and will not gather enough confidence to prefetch either with stride +1 or +2. IPCP's CPLX prefetcher will be able to detect this pattern. However, IPCP ignores the timeliness of prefetching. In contrast to IP-stride and IPCP's CPLX prefetcher, a more accurate, and flexible approach would be to prefetch with deltas +3 or +6 that provide 100% coverage. Moreover, for the irregular stride sequence: -1, -5, -2, -1, -4, -1 associated to IP `0x402dc7` from `mcf-1554B`, IPCP's CPLX prefetcher fails to predict a pattern through its lookahead based on confidence. However, a local delta prefetcher with a delta of -1 can provide better coverage.

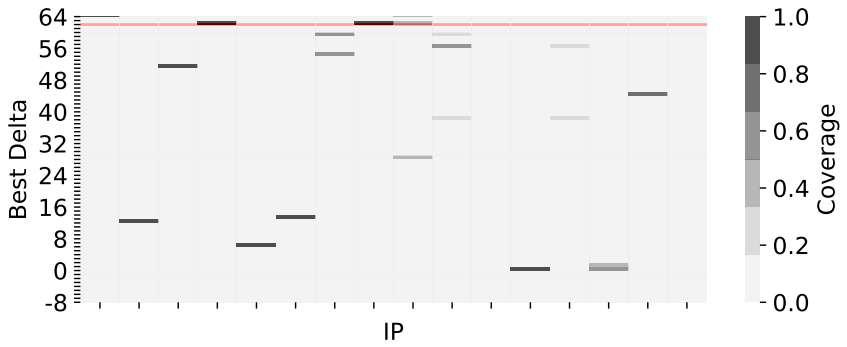


Figure 4.3: Best delta selected by BOP (based on global deltas) and Berti (based on per-IP local deltas) for `mcf-1554B`. Prefetch coverage is shown in grayscale. BOP selects +62 as the best delta (red line), which is not always accurate and provides a coverage of only 2%.

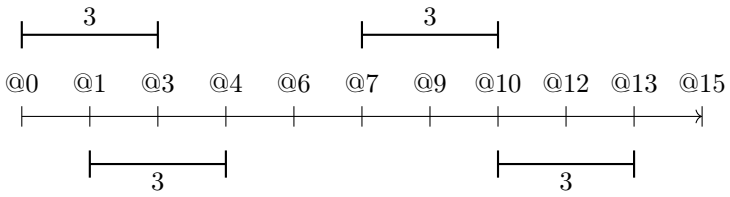


Figure 4.4: +1, +2, ... memory miss pattern. Prefetching with delta 3 get 100% coverage

Effect of out-of-order loads at the L1D

In an out-of-order processor, memory accesses get reordered due to out-of-order scheduling. Hence, the training of a delta prefetcher may be affected by the ordering of memory accesses. Let's consider a loop with a single IP accessing memory addresses 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 with constant strides of +1. An out-of-order processor can reorder, for example, the accesses to addresses 2, 3, 7, 8, and 9, resulting in the following sequence of addresses: 0, 1, 2, 4, 3, 5, 6, 9, 8, 7, 10 and strides +1, +1, +2, -1, +2, +1, +3, -1, -1, +3 at the L1D. This cannot be covered by an IP-stride or an IPCP's CPLX prefetcher unless a specific mechanism can provide the commit order [54, 159]. However, deltas have the important property of seeing past accesses already in order, thus there is no requirement of such in-order commit mechanisms. Indeed, the last three accesses in our example will see the following past deltas: addresses 8 will see -1, +2, +3, +5, +4, +6, +7, +8, addresses 7 will see +1, -2, +1, +2, +4, +3, +5, +6, +7, and addresses 10 will see +3, +2, +1, +4, +5, +7, +6, +8, +9, +10 that is, all possibilities regardless of their order. The prefetcher then can choose the deltas that can provide the best coverage from these set of values.

Figure 4.5 shows a +1 memory access pattern with memory reordering due to out-of-order and prefetching with delta 5.

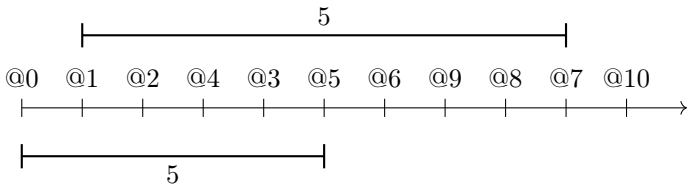


Figure 4.5: Memory reordering due to out-of-order processors. Prefetching with delta 5 get 100% coverage

Why a timely local delta prefetcher?

If the goal of a prefetcher is to *cover* address 15 (see Figure 4.6), then the prefetcher can initiate prefetching with deltas +3, +5, and +8 whenever it sees the demand accesses to addresses 12, 10, and 7, respectively. However, if we consider time to prefetch address 15, then deltas of +3, +5, and +8 will not completely mitigate the L1D miss latency, as they will be late prefetch requests. Instead, if a prefetcher issues a request for address 15 with deltas of +10 or +13 on demand accesses to address 5 or 2, respectively, it can prefetch address 15 well ahead of time.

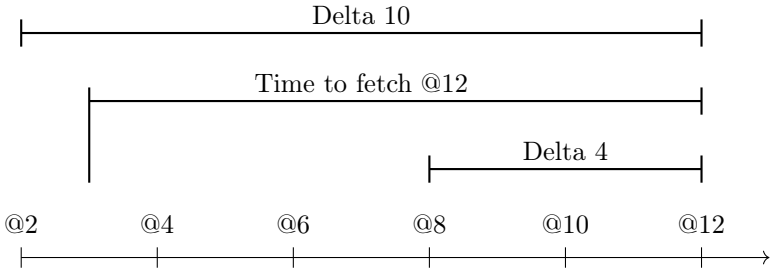


Figure 4.6: +2 Memory miss pattern, with multiple possible deltas but only delta +10 can hide the latency of @12 miss

4.3 Berti: A Local-Delta Prefetcher

Berti is a data prefetcher sited at the L1D, where it can see all the requests generated by the processor and orchestrate the prefetch requests to the memory hierarchy. Berti makes a strong case for prefetch accuracy. For each IP, it selects the deltas[2] that are timely and computes their respective local coverage. High accuracy is achieved by only using deltas with high coverage. Additionally,

[2]For the rest of the chapter, unless specified we use the terms delta and local delta interchangeably.

Berti is trained with virtual addresses, which helps in finding larger deltas and facilitates cross-page prefetching. Next, we describe how Berti performs training and prediction. Then, we propose a simple and cost-effective hardware implementation.

4.3.1 Training the Prefetcher

The goal of the training mechanism is to estimate the coverage of each seen delta, considering only those deltas that would result in a timely prefetch. The training consists of the following actions: measuring fetch latency, learning timely deltas, and computing the coverage of the deltas.

Measuring fetch latency

In order to learn the deltas that are *timely* it is necessary to measure the time required to fetch data to the L1D, i.e., the L1D miss latency. This measurement is performed for any cache line in L1D, both for demand misses and prefetch requests. Computing latency for prefetch requests is fundamental because, in an ideal scenario, there would not be L1D misses but just L1D hits due to timely prefetch requests. In addition, the latency of prefetch requests may be larger than the latency of demand requests due to prefetch queue (PQ) contention or L1D port contention. Fetch latency can be measured by keeping a timestamp for any L1D miss inserted into the MSHR and any prefetch request inserted into the PQ. On an L1D fill, the latency is simply computed by subtracting the stored timestamp from the current one.

Learning timely and accurate deltas

Once the fetch latency is obtained for each L1D fill, our prefetcher can precisely learn timely deltas, given that the history of accesses and timestamps by the same IP is recorded. By searching in the history of recent accesses and comparing the timestamp of each previous access with the timestamp when a prefetch should be issued to be timely, the accesses that would trigger timely prefetch requests are detected. Deltas are then computed by subtracting the addresses of each timely request in the history from the current addresses. Figures 4.7, 4.8, and 4.9 depict how timely deltas are detected. All addresses represented in the timeline are accessed by the same IP. When addresses 10 is demanded and its fetch latency computed (Figure 4.7), the history of accesses for that IP is searched, from the point in time, a timely prefetch should have been triggered. In this case, no previous accesses are found. After accessing addresses 12 and computing its fetch latency (Figure 4.8), a timely delta corresponding to addresses 2 is found. That is, addresses 2 should initiate the prefetch request for 12 in order to be timely. The timely delta +10 is therefore learned. Similarly, when computing the latency for access 15 (Figure 4.9), two deltas, +10 and +13, are detected as timely.

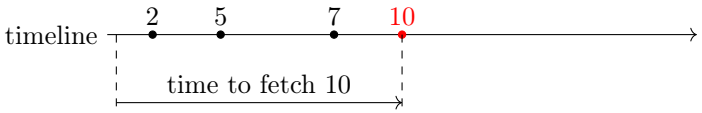


Figure 4.7: Access address 10: no timely delta found.

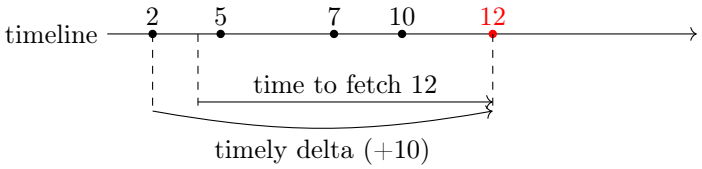


Figure 4.8: Access address 12: one timely delta found.

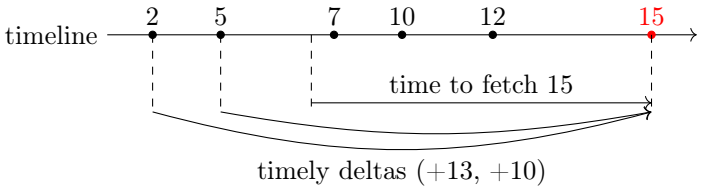


Figure 4.9: Access address 15: two timely deltas found.

Berti triggers the procedure to learn timely deltas for each miss that would have occurred in the baseline, which translates to two scenarios. First, when a demand miss fills the L1D with the requested data. Second, when a cache line brought into L1D by a prefetch request is demanded (i.e., misses that would have occurred without a prefetcher). Berti does not learn deltas on a cache fill caused by a prefetch request since its demand time is not known. Therefore, it is necessary to keep the latency of prefetch requests until the core demands the cache line.

Computing the coverage of deltas

On every search in the history, Berti obtains a set of timely deltas. Deltas that frequently appear in the searches would cover a significant fraction of misses, while deltas that rarely appear would result in low coverage. It is easy to compute the coverage by dividing the number of occurrences of a delta by the number of searches in the history. For example, in Figure 4.9, after three accesses, the delta +10 has the higher coverage, being in two out of three searches (66.7%). If the same access pattern continues, the delta +10 will reach close to 100% coverage. It is important to note that this local (per IP) coverage translates into accuracy. If a delta covers 100% of cache lines, since each access-delta pair results in only one prefetch request, that delta will bring 100% accuracy.

4.3.2 Prediction: Issuing Prefetch Requests

Once we know the deltas and their associated coverage, we can orchestrate the prefetch requests across the cache hierarchy. Based on both the coverage of each delta and the L1D MSHR occupancy, we decide which deltas to use and till which cache level to prefetch. We use four watermarks to decide where to issue the prefetch requests. If the coverage of a delta is above a *high-coverage* watermark and the L1D MSHR occupancy is below the *occupancy* watermark, then prefetch requests using that delta get filled at all the cache levels till L1D. Otherwise, if the coverage is above a *medium-coverage* watermark, irrespective of the L1D MSHR occupancy, prefetch requests get filled till L2. Finally, if the coverage is above a *low-coverage* watermark, requests get filled only in the LLC.

To generate a prefetch request, we add the selected delta to the addresses of the current access and the resulting addresses is inserted in the PQ. Requests in the PQ are processed in a first-in-first-out (FIFO) order. Since our prefetcher is trained with virtual addresses, the generated prefetch requests are also in the virtual addresses space. A prefetch request obtains the physical addresses from the L2 translation look-ahead buffer (STLB). If the translation misses in the STLB, the prefetch request is dropped. If the translation is obtained, the prefetch request checks if the target block is already present in the cache it wants to fill. In case of a miss, the block is prefetched, and the request is inserted into the MSHR.

4.3.3 Hardware Implementation

As outlined in Figure 4.10, Berti can be implemented with a small hardware budget and using simple structures and logic. Next, we describe the structures required to train the prefetcher and decide on the prefetch requests to issue to each cache level.

Measuring fetch latency

In order to be able to measure the fetch latency, the MSHR is extended with a 16-bit field (represented in Figure 4.10 in gray) that stores a timestamp on a demand L1D miss. Similarly, the PQ is also extended with an analogous field that stores the timestamp when a new prefetch request is added. The timestamp can be obtained from the clock of the local processor [141] or any other metric to approximate time (e.g., number of cache accesses). In our implementation, we use the former. When a prefetch request misses L1D, the timestamp is transferred from the PQ to the newly allocated MSHR entry. On an L1D fill, the latency of the request can be computed with a simple subtraction. The latency is stored using 12 bits. If an overflow is detected when computing the latency, it is set to zero, and therefore not considered for learning timely deltas. Based on our empirical results, on average across GAP and SPEC CPU2017 traces, we see 1.08 overflows per kilo L1D fills.

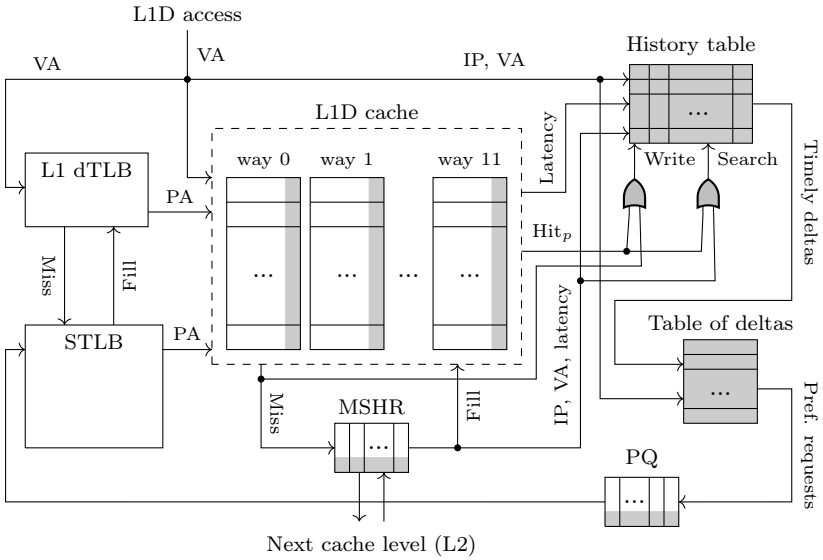


Figure 4.10: Berti design overview. Hardware extensions are shown in gray.

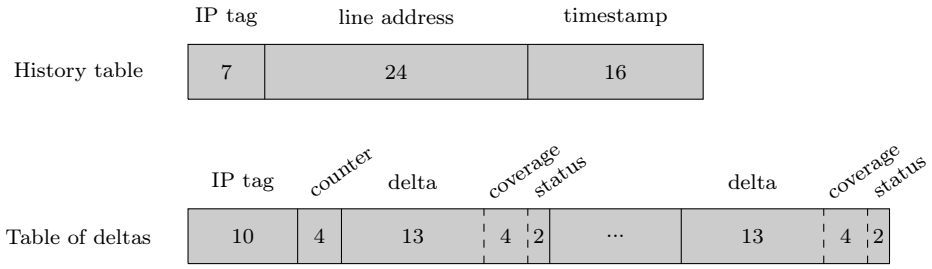


Figure 4.11: *History table* and *Table of deltas* entry format.

Learning timely deltas

To be able to learn timely deltas, the most recent accesses need to be tracked. The *History table* (see Figure 4.10) records that information and is organized as an 8-set, 16-way cache with a FIFO replacement policy and indexed and searched with the IP. The format of each entry in the history table is depicted in Figure 4.11. Each entry keeps a tag corresponding to the seven least significant bits of the IP (after removing the bits used for indexing the cache), the 24 least significant bits of the target cache line addresses, and a 16-bit timestamp. A new entry is inserted in the history table (*Write* port in Figure 4.10) either on-demand misses (*Miss* arrow from the L1D in Figure 4.10) or on hits for prefetched cache lines (Hit_p in Figure 4.10). The virtual addresses (VA) and the IP (*IP, VA* arrow in Figure 4.10) are stored in the new entry along with the current timestamp (not shown in the figure).

The search for timely deltas (*Search* port in Figure 4.10) is performed either on a fill due to a demand access (*Fill* arrow from the MSHR in Figure 4.10) or on a hit due to a prefetched cache line (Hit_p in Figure 4.10). In the first case, the search is done using the information from the MSHR (*IP, VA, latency* arrow in Figure 4.10). In order to enable the search on L1D hits, we keep the latency of the prefetch request (12 bits) along with each entry in the L1D (see Figure 4.10 L1D shadow part). Alternatively, an L1D shadow tag could be employed. A latency field set to zero indicates either an overflow when computing the latency or an already demanded cache line. In that case, a search in the history table is not performed. Otherwise, the search is done when the demand hit takes place, using the stored latency (*Latency* arrow in Figure 4.10), which is reset after the search. On every search, the 16-ways of the history table are looked up for a matching IP tag. A maximum of eight timely deltas, the ones corresponding to the youngest entries that would result in timely prefetch requests, are collected.

Computing the coverage of deltas

The results of each search in the history table (*Timely deltas* arrow in Figure 4.10) are accumulated in the *Table of deltas*, a 16-entry fully-associative cache with a FIFO replacement policy. The format of each entry in the table of deltas is depicted in Figure 4.11. Each entry consists of a 10-bit tag (based on hash function of the IP), a 4-bit counter, and an array of 16 deltas, each of them containing the delta itself (13 bits), the coverage (4 bits), and the status (2 bits) indicating till which cache level to prefetch. The counter is increased on each search in the history table. For each timely delta found during the search, its coverage counter is increased. When the counter overflows (its value increases to 16), we compute the coverage. Deltas that cross the *high-coverage* watermark (65% of coverage, i.e., a coverage value higher than 10) set their status to $L1D\_pref$. Deltas in between the *high-coverage* watermark and the *medium-coverage* watermark (between 65% and 35%, i.e., a coverage value lower or equal than 10 and higher than 5) set the status to $L2\_pref$. The maximum number of deltas selected for any of those status is bounded to 12. The remaining deltas' status is set to No_pref (i.e., do not issue prefetch requests for this delta). Once the status is set, the counter and the array of confidences are reset, and a new learning phase begins.

While warmingup the status fields, prefetch requests are also issued if at least eight deltas have been gathered, increasing the *high-coverage* watermark to 80%, as with just four deltas the prefetcher needs more confidence. Our empirical study shows that using watermarks higher that 65% leads to high accuracy.

Although Berti opens the possibility of prefetching to LLC only for low-coverage deltas, our evaluation showed no performance improvements when choosing this option. Hence, we set the *low-coverage* watermark to 35% (equal to the *medium-coverage* one), to disable prefetching to LLC only.

In order to constantly learn new deltas, evictions of deltas may be necessary.

Table 4.1: Storage overhead of Berti.

Structure		**Storage**
History table	8-set, 16-way (128-entry) cache, FIFO replacement policy. Each set: 4 bits (replacement policy). Each entry: 7-bit IP tag, 24-bit address, 16-bit timestamp	0.74 KB
Table of deltas	16-entry, fully-associative, 4-bit FIFO replacement policy. Each entry: 10-bit IP tag, 4-bit counter, and an array of 16 deltas (13-bit delta, 4-bit coverage, 2-bit status)	0.62 KB
PQ + MSHR	16+16 entries, 16-bit timestamp per entry	0.06 KB
L1D	768 cache lines, 12-bit latency per line	1.13 KB
Total		**2.55 KB**

On the arrival of a non stored delta, deltas with less than 50% coverage in the previous phase are candidates for evictions in the current phase. To this end, if the coverage when selecting the $L2\_pref$ status is lower than 50%, the status is set to $L2\_pref\_repl$. The eviction policy selects the delta with lower coverage whose status is $L2\_pref\_repl$ or No_pref. In case no such delta exists, the new delta is discarded.

Issuing prefetch requests

On every L1D access, the table of deltas is searched looking with a matching IP (*IP, VA* arrow pointing to the table of deltas in Figure 4.10). Since we use an L1D with two read ports and one write port, the table of deltas requires three search ports. The deltas with status $L1D\_pref$ or $L2\_pref$ are added to the current VA to form the prefetch requests that are inserted into the PQ (*Pref. requests* arrow in Figure 4.10). Those prefetch requests get filled into all cache levels till L1D when the status is $L1D\_pref$ and the MSHR occupancy is below 70% (the *occupancy* watermark). Otherwise, prefetch requests get filled till L2.

Storage overhead

Berti does not require any complex operation (e.g., multiplications) nor complex logic. Our history table has two read ports and one write port. The latency of this structure is two cycles, based on CACTI-P [99]. Since prefetching training is out of the critical path of memory accesses, the history table does not affect the cycle time. The storage requirements of Berti, whose breakdown per structure is provided in Table 4.1, is just 2.55 KB.

Table 4.2: Simulation parameters of the baseline system.

Core	Out-of-order, hashed perceptron branch predictor [88], 4 GHz with 6-issue width, 4-retire width, 352-entry ROB
TLBs	L1 iTLB/dTLB: 64 entries, 4-way, 1 cycle STLB: 2048 entries, 16-way, 8 cycles
MMU Caches	2-entry PSCL5, 4-entry PSCL4, 8-entry PSCL3, 32-entry PSCL2, searched in parallel, one cycle
L1I	32 KB, 8-way, 4 cycles
L1D	48 KB, 12-way, 5 cycles, with a 24-entry, fully associative IP-stride prefetcher [24]
L2	512 KB 8-way associative, 10 cycles, SRRIP [87], non-inclusive
LLC	2 MB/core, 16-way, 20 cycles, DRRIP [87], non-inclusive
MSHRs	8/16/32 at L1I/L1D/L2, 64/core at the LLC
DRAM controller	One channel/4-cores, 6400 MTPS [29], FR-FCFS, 64-entry RQ and WQ, reads prioritized over writes, write watermark: 7/8th
DRAM chip	4 KB row-buffer per bank, open page, burst length 16, t_{RP}: 12.5 ns, t_{RCD}: 12.5 ns, t_{CAS}: 12.5 ns

4.4 Experimental Environment and Methodology

4.4.1 Simulation Framework

For the evaluation of our proposal we use a modfied version of ChampSim [22]. ChampSim is a trace-driven simulator used for the 2nd and 3rd Data Prefetching Championships (DPC-2 [3] and DPC-3 [4]). Recent prefetching proposals [9, 13, 122, 145] are also coded and evaluated on Champsim. The recently modified Champsim extends the one provided with the DPC-3 with a decoupled front-end [138] and a detailed memory hierarchy support for addresses translation that further improve the baseline performance. We model a three level non-inclusive cache, cache lines are filled in all levels but the eviction of a block from a level is not notified to the rest of the cache levels. The L1D cache has an IP-Stride prefetcher that mimics the Intel prefetcher [37]. We faithfully model DRAM, including the queuing delays that contributes to the variable access time because of close vs. open page, page hit vs miss, DRAM bank conflicts, etc. Table 4.2 summarizes our system configuration, mimicking an Intel Sunny Cove microarchitecture [49, 166, 5].

We evaluate our proposal with single-core and multi-core simulations. We warmup the caches for 50M sim-point instructions [146] and collect statistics for the next 200M sim-point instructions. For multi-core simulations, we use heterogeneous mixes of single-threaded traces. For each mix, when a core finishes

its 200M instructions, it gets replayed until all the cores finish their respective 200M instructions. For both single- and multi-core, we report performance in terms of IPC improvement (speedup) with respect to an L1D with an IP-stride prefetcher. We use the geometric mean to average the speedups obtained by the different single-thread traces.

4.4.2 Energy Model

For reporting the dynamic energy consumption of the memory hierarchy. We obtain the energy consumption of reads and writes to tag and data arrays at each cache level and DRAM with CACTI-P [99] and Micron DRAM power calculator [105]. Then, we compute the total energy expenditure by accounting for the number of accesses of each type across the memory hierarchy. We use 22 nm process technology for our energy calculations.

4.4.3 Workloads

We use traces from SPEC CPU2017 [156] and single-threaded GAP benchmarks [10]. We limit our study to memory-intensive traces (MemInt), i.e., those that showed at least one miss per kilo-instruction (MPKI) at the LLC in our modeled baseline system. All GAP traces (20) and 44 SPEC CPU2017 traces are memory-intensive.

SPEC CPU2017 traces were generated with the reference inputs. Both real (Twitter, Web, Road) and synthetic (Kron, Urand) graphs were used as input for the GAP benchmarks.

We also report performance for the CloudSuite benchmarks [25] traces are publicly available [152, 52, 25]. For multi-core experiments, we simulate 200 random heterogeneous mixes from SPEC CPU2017 and GAP.

4.4.4 Berti and Variable Cache Fill Latency

Modern memory hierarchies can have variable cache fill latency that comes from sources like MSHR contentions at the private and shared caches, read queue (RQ) and write queue (WQ) contentions at various levels of caches and DRAM controller. At the DRAM level, memory access time gets affected because of row buffer conflicts, bank conflicts, etc. Our simulator reflects all this variability. Even non uniform cache access (NUCA) LLCs with multiple banks can cause variable fill latency. For example, suppose in a NUCA LLC with two banks an IP sees local deltas of +1 and +2 that get mapped to bank-1 and bank-0, respectively, and the latency to bank-0 is different from bank-1. In our experiments, the fill latency ranges from 22 to 2098 cycles with an average of 278 cycles averaged across SPEC CPU2017, GAP, and CloudSuite benchmarks and multicore mixes.

Table 4.3: Configurations of evaluated prefetchers.

SPP-PPF [13]	256-entry ST, 512-entry 4-way PT, 8-entry GHR, Perceptron weights with the following entries: 4096×4, 2048×2, 1024×2, and 128×1 entries, 1024-entry prefetch table, 1024-entry reject table
Bingo [145]	2 KB region, 64/128/4K-entry FT/AT/PHT
MLOP [9]	128-entry AMT, 500-update, 16-degree
IPCP [122]	128-entry IP table, 8-entry RST table, and 128-entry CSPT table

4.4.5 Evaluated Prefetching Techniques

We compare the effectiveness of Berti with high performing L1D and L1D+L2 prefetchers. As Berti is an L1D prefetcher, we first compare its performance with prefetchers designed for L1D (no prefetching at the L2), and then with multi-level prefetching combinations. The L1D prefetchers are i) MLOP [145] (DPC-3, 3rd place), an extension of the BOP (DPC-2 winner), and ii) IPCP (DPC-3 winner published at ISCA 2020 [122]). For multi-level prefetching, we evaluate two state-of-the-art L2 prefetchers along with MLOP and Berti at the L1D: SPP-PPF [93, 13] and Bingo [9]. We also compare with a multi-level IPCP that uses IPCP both at the L1D and L2. The evaluated prefetchers have been briefly described in Section 4.2.1. For all prefetchers, we use a highly tuned implementation as provided by the authors and tune it again for the parameters mentioned in Table 4.2. Fine tuning was an easy exercise as all the competing prefetchers use Champsim for their evaluation. Table 4.3 shows the configurations used for all the evaluated prefetchers.

4.5 Berti Evaluation

4.5.1 Speedup vs. Storage Requirements

Figure 4.12 summarizes the speedup of the evaluated prefetchers with respect to IP-stride for SPEC CPU2017 and GAP, along with their storage requirements. L1D prefetchers are shown with a circle, L2 prefetchers with a square, and multi-level (L1D+L2) prefetchers with a diamond.

Among the L1D prefetchers, Berti achieves the highest speedup with a size similar to IPCP, the prefetcher with the lowest storage budget. With only 2.55 KB of storage overhead, Berti improves performance by 8.5% over IP-stride and 3.5% over IPCP, the state-of-the-art L1D prefetcher.

The Berti+SPP-PPF multi-level prefetcher obtains the highest speedup (10.2%, additional 1.5% on top of Berti at L1D) among all multi-level combinations with 41.8 KB combined storage for L1D and L2 prefetchers. However, the highlight of Figure 4.12 is that Berti at L1D without any prefetching at L2 outperforms all the multi-level prefetching combinations that do not include

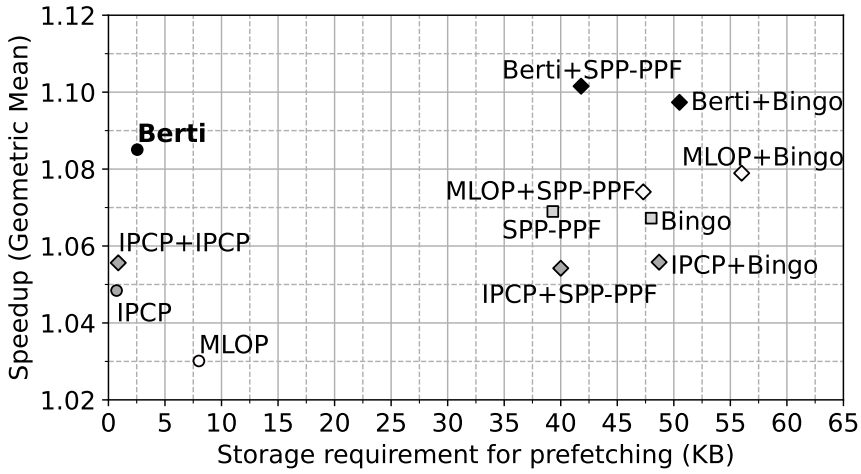


Figure 4.12: Speedup vs. storage requirements. Speedup is normalized to L1D IP-stride and averaged across memory-intensive SPEC CPU2017 and GAP traces. X+Y denotes prefetcher X at L1D and prefetcher Y at L2.

Berti.

4.5.2 Performance of Berti as an L1D Prefetcher

Figure 4.13 shows the speedup achieved by the L1D prefetchers for SPEC CPU2017 and GAP. Berti is the best prefetcher across both suites. On average, Berti at the L1D improves performance by 11.6% and 1.9% for SPEC CPU2017 and GAP, respectively. All three prefetchers achieve good speedups for SPEC CPU2017, and Berti outperforms IPCP and MLOP by 2.8% and 3.0%, respectively. The speedup differences are more significant with the GAP traces, where Berti is the only L1D prefetcher that improves IP-Stride, by up to 1.9%, while IPCP and MLOP are 2.9% and 7.8% below, respectively. Overall, across SPEC CPU2017 and GAP traces, Berti outperforms IP-stride and IPCP by 8.5% and 3.5%, respectively. This is significant performance improvement on top of the high-performing state-of-the-art IPCP prefetcher.

Figure 4.14 shows the individual speedup for the memory-intensive SPEC CPU2017 and GAP traces. For CPU2017, Berti achieves similar or significantly better results than the other prefetchers on all traces except for `CactuBSSN`.

In this benchmark, we observe that the memory access instructions follow stride patterns. However, there are hundreds of these instructions executing interleaved. Therefore, to track the local behavior of instructions in this benchmark, Berti would need very large history and delta tables. In contrast, prefetchers that detect patterns in the global addresses stream do not have

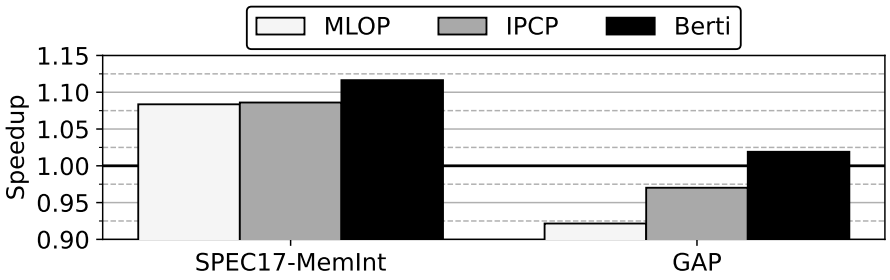
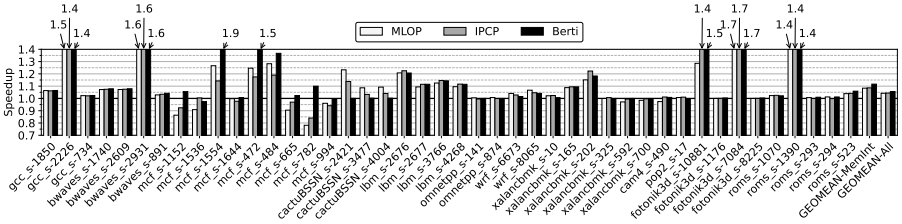
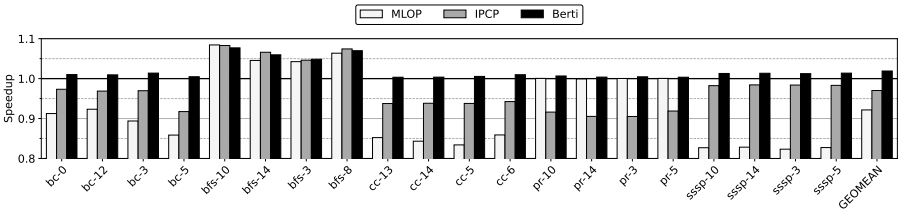


Figure 4.13: Speedup of L1D prefetchers compared to a system with L1D IP-stride for memory-intensive SPEC CPU2017 and GAP traces.



(a) Memory-intensive SPEC CPU2017



(b) GAP

Figure 4.14: Speedup with Berti as an L1D prefetcher for (a) 44 SPEC CPU2017 and (b) 20 GAP memory-intensive traces normalized to L1D IP-stride. Geomean-all corresponds to the geometric mean of all the 95 SPEC CPU2017 traces.

this problem, as is the case with MLOP or the IPCP GS class. Barring the exception of `CactuBSSN` where global deltas perform better than local deltas, Berti shows that local deltas are prevalent across a large number of benchmarks and it accurately selects them.

A key observation is that the state-of-the-art prefetchers do not consistently improve performance over IP-stride across all workloads, but Berti only shows a small degradation of 2.6% with respect to IP-stride for `mcf_s-1536`.

For SPEC CPU2017, Berti achieves its best result for `mcf_s-1554` where it provides speedups of 1.89×, 1.65×, and 1.49× with respect to IP-stride, IPCP, and MLOP, respectively. MLOP and IPCP achieve performance at least

1% below IP-stride on five and eight traces out of 44, the worst case being in `mcf_s-782` with drops of 16.0% and 21.9%, respectively. In `mcf_s-782`, only three IPs (`0x4049de, 0x4049e5`, and `0x4049cc`) represent the 75% of all L1D accesses. MLOP uses a global delta to prefetch that is affected by the interleaving of accesses from these three IPs. IPCP uses the CS and CPLX class prefetchers but with an accuracy below 25%.

As for GAP, Berti is the best prefetcher for all the benchmarks but three (`bfs-8`, `bfs-10` and `bfs-14`). It consistently achieves similar or better results than IP-stride for all traces, while MLOP and IPCP perform worse than IP-stride in 12 and 16 benchmarks, respectively. In some cases, the MLOP slowdown is very significant, for example, 17.7% in `sssp-3`. We have also analyzed the behavior of the prefetchers in one of the GAP applications, namely `bc-5`. All `bc-5` IPs show a rather chaotic memory access pattern except for one that is very regular. IP-stride and Berti, by separately tracing the IPs, detect the regular IP pattern and prefetch correctly for it. They do not prefetch for the other IPs. MLOP fails due to the use of a global delta. The accesses issued by IPs with irregular pattern prevent discovering a global delta and therefore the prefetcher issues very few requests, and is not able to prefetch correctly for the regular IP. IPCP detects the delta pattern for the regular IP through its CPLX component, and prefetches correctly for it. However, the GS component generates many useless prefetches that drastically decreases the accuracy of IPCP and results in the loss of performance shown in Figure 4.14.

Accuracy

Figure 4.15 shows the accuracy of the L1D prefetchers. Berti is a very accurate prefetcher. On average, about 87.2% of its prefetched lines are useful compared to 62.4% for MLOP and 50.6% for IPCP. The effectiveness of IPCP is driven by the performance of several tiny prefetchers: a global stream prefetcher (GS class), a constant stride prefetcher (CS class), and a complex stride prefetcher (CPLX class) that work in tandem. For regular access patterns, the CS prefetcher provides high accuracy. However, for complex access patterns, the effectiveness of the CPLX prefetcher is low, with an accuracy of 52.7% and 9.8% for SPEC CPU2017 [156] and GAP [10] workloads, respectively.

MLOP, like Berti, is based on the detection of the best timely deltas. However, it achieves much lower accuracy. The improvement of Berti over MLOP is mainly due to two factors: i) MLOP uses global deltas for the whole application while Berti detects different deltas for each IP. As we have shown in Section 4.2.2, benchmarks such as `mcf` generate different delta patterns for each IP. ii) Berti uses a stringent policy to decide which deltas to use for issuing prefetch requests into L1D, as we have described in Sections 4.3.2 and 4.3.3, while MLOP generates prefetch requests for the best delta with each lookahead regardless of its confidence.

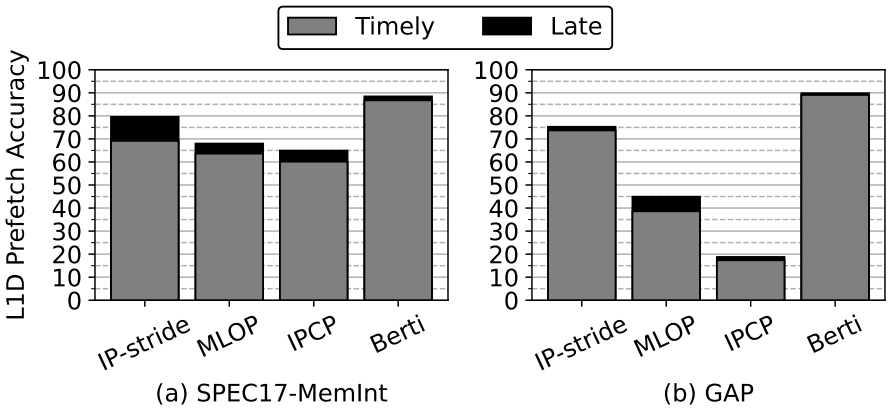


Figure 4.15: Prefetch accuracy at the L1D. Percentages of useful requests are broken down into timely (gray) and late (black) prefetch requests.

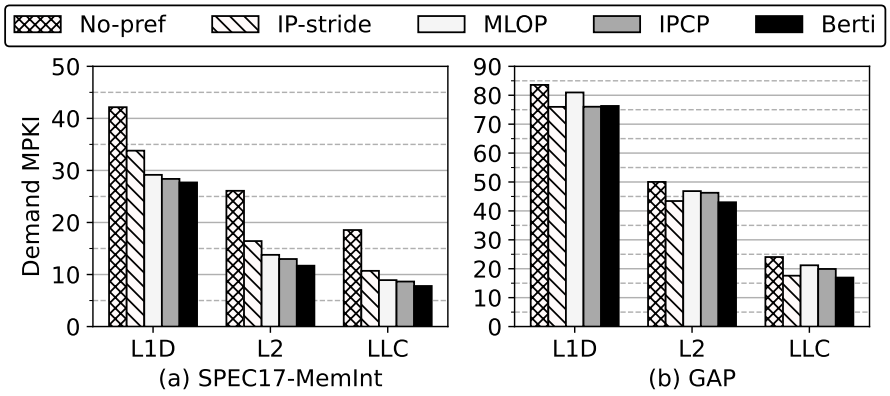


Figure 4.16: Prefetch coverage in terms of average L1D, L2, and LLC demand MPKIs for all L1D prefetchers.

Timeliness

The darker part of each bar in Figure 4.15 represents the prefetch requests whose retrieved data arrive late to L1D. Almost all prefetch requests generated by Berti are timely, while MLOP and IPCP produce a significant number of late requests. IPCP does not use any mechanism to adapt the prefetch requests timing to the miss latency, while MLOP and Berti do. However, Berti achieves better timeliness than MLOP due to specific and timely deltas for each IP.

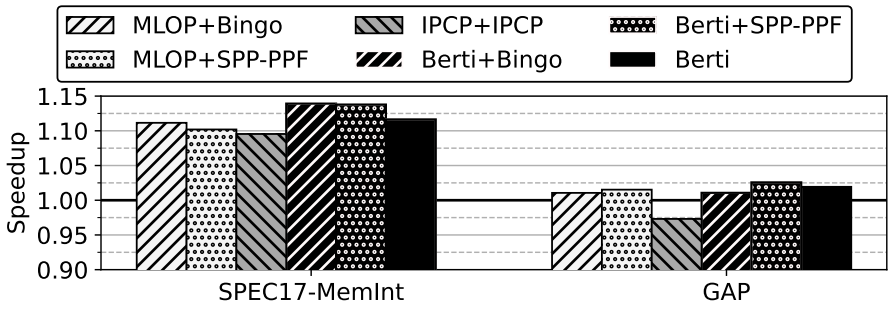


Figure 4.17: Spedup with multi-level prefetching normalized to L1D IP-stride.

Coverage

Figure 4.16 shows demand misses per kilo instructions (MPKI) at the L1D, L2, and LLC with and without L1D prefetchers. Berti and IPCP achieve a similar reduction of misses in L1D (8.7% in GAP, 33.4% in SPEC CPU2017) and slightly higher than MLOP. However, Berti manages to eliminate more misses than IPCP and MLOP at L2 and LLC due to its line preloading policy directed by the L1D prefetcher. The biggest differences are observed in GAP, where Berti reduces LLC demand misses by 17.7% and 12.4% compared to MLOP and IPCP, respectively. Similarly, Berti reduces L2 demand misses by 6.7% and 5.6% compared to MLOP and IPCP, respectively.

4.5.3 Multi-level Prefetching Performance

Figure 4.17 shows the speedup achieved with the multi-level prefetching combinations compared to a system with IP-stride. We select the best five multi-level prefetching combinations out of all possible combinations of L1D and L2 prefetchers. Multi-level prefetching combinations do not offer a significant performance boost. The best multi-level prefetching combinations without Berti are MLOP+Bingo for SPEC CPU2017 and MLOP+SPP-PPF for GAP. In both cases, these combinations achieve a similar speedup to Berti alone, with a storage requirement 22 and 18 times higher, respectively. IPCP at L1D and L2 (IPCP+IPCP), with a meager hardware budget as Berti, achieves a significantly lower speedup than Berti alone at the L1D, especially in GAP, with a difference of 4.6%.

Adding a prefetcher at the L2 cache along with Berti at the L1D achieves a moderate performance gain. The most significant gain is 2.0% and is obtained with the Berti+Bingo configuration in the memory-intensive subset of SPEC CPU2017 traces. Given the high hardware cost of the L2 prefetchers, the configuration with Berti alone at the L1D seems to be a better design in terms of performance and storage trade-off.

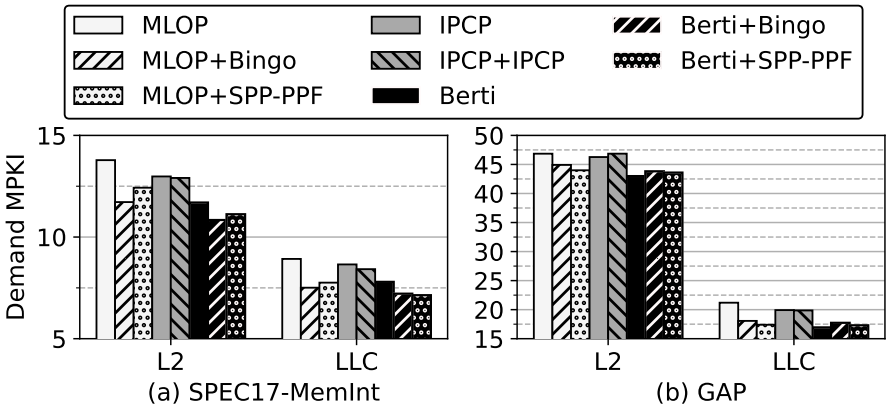


Figure 4.18: Prefetch coverage in terms of average L2 and LLC demand MPKIs with multi-level prefetching.

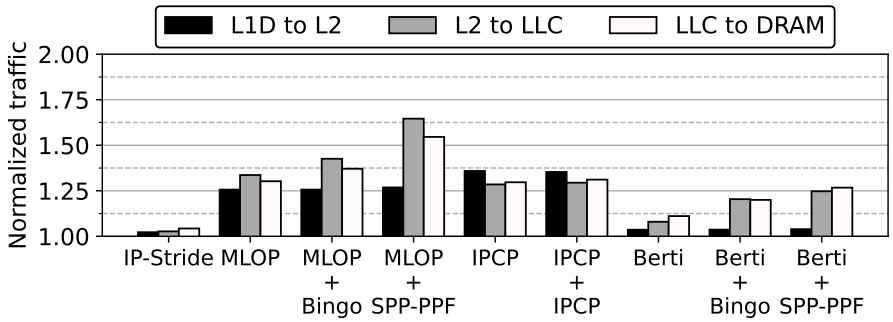
Coverage

Figure 4.18 shows demand MPKIs at the L1D, L2, and LLC for the multi-level prefetching combinations. We also show MPKIs without prefetching at L2 for ease of analysis. MLOP+Bingo and MLOP+SPP-PPF decrease MPKI relative to MLOP alone in both L2 and LLC consistently across all suites (maximum reduction in MPKI from 13.8 to 11.7 at L2 for SPEC CPU2017). Adding a prefetcher at L2 is less effective for IPCP and Berti. In both cases, MPKIs at L2 and LLC decrease for SPEC CPU2017 but remain the same or even increase slightly when working with the irregular access patterns of GAP. As a result, the MPKIs at the L2 and LLC achieved by Berti at the L1D are always better than those obtained by multi-level prefetchers with no Berti, except for MLOP+Bingo in SPEC CPU2017.

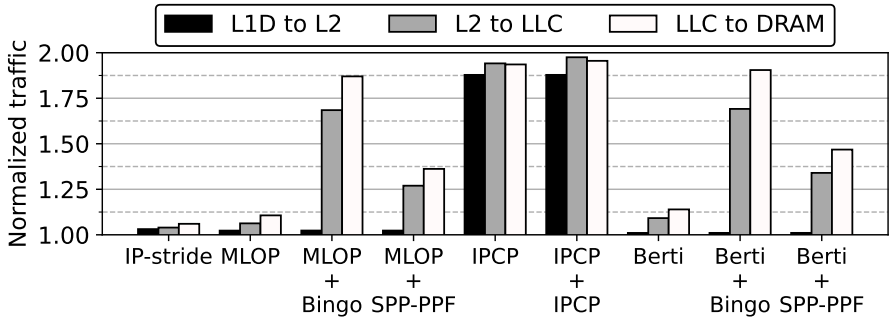
4.5.4 Memory Hierarchy Traffic and Energy

Figure 4.19 shows the traffic between the different levels of the memory hierarchy (demand and prefetch requests) for different prefetching combinations normalized to no prefetching. All prefetchers increase traffic as a result of the useless blocks they request. For the systems with prefetcher only at the L1D, we can observe how the traffic increase is inversely proportional to the prefetcher accuracy for SPEC CPU2017 (refer Figure 4.15). Consequently, Berti is the prefetcher with the lowest traffic increase at all levels. For GAP, MLOP increases traffic marginally, despite its low accuracy, because it detects few patterns and generates scarce prefetch requests. Berti increases traffic with L2, LLC and DRAM by 1.0%, 9.2% and 13.9% respectively, whereas IPCP increases traffic at these three levels around 90%.

The L2 prefetchers Bingo and SPP-PPF added to MLOP and Berti on L1



(a) Memory-intensive SPEC CPU2017



(b) GAP

Figure 4.19: L2, LLC and DRAM demand and prefetch traffic normalized to no-prefetching

significantly increase traffic with LLC and DRAM, especially at GAP due to the irregular access patterns. MLOP+Bingo induces 69.0% additional off-chip traffic compared to MLOP alone, while Berti+Bingo adds 67.2% additional off-chip traffic compared to Berti alone.

Energy efficiency

Figure 4.20 shows the average dynamic energy consumption in the memory hierarchy (L1D, L2, LLC, and DRAM) normalized to no prefetching. As expected, there is a direct correlation between traffic and energy consumption overheads in the memory hierarchy. If we focus on the state-of-the-art L1D prefetchers, Berti consumes the least extra energy for SPEC CPU2017 (9.0% vs. 29.1 and 30.1% for MLOP and IPCP), despite achieving the highest speedup (see Figure 4.13). As for GAP, the energy overheads of Berti and MLOP are similar (14.3% vs. 14.2%), and significantly lower than for IPCP (86.9%). Berti is the only prefetcher that manages to translate its dynamic energy increase into speedup. The L2 Bingo and SPP-PPF prefetchers on top of MLOP and Berti significantly increase energy consumption, especially in the case of Bingo

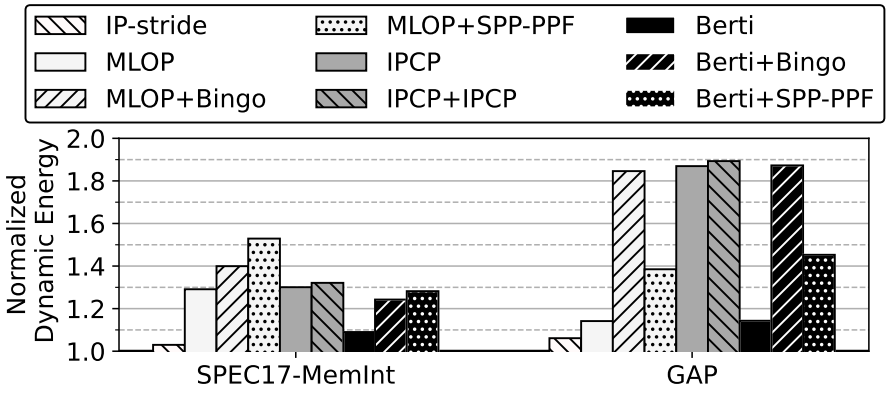


Figure 4.20: Dynamic energy consumption in the memory hierarchy normalized to no-prefetching.

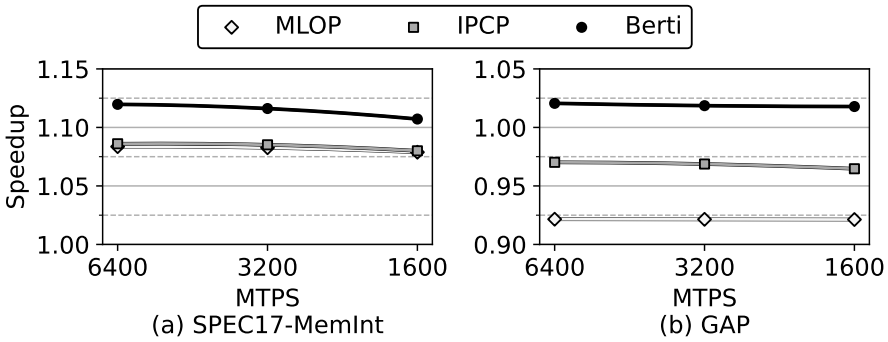


Figure 4.21: Performance of L1D prefetchers in constrained DRAM bandwidth, in MTPS.

for GAP, with increases of over 60% with respect to MLOP and Berti alone.

4.5.5 Effect of Constrained DRAM Bandwidth

So far, we have considered the latest DDR5-6400 channel per four cores that provides 6400 million transfers per second (MTPS) with a per-core DRAM bandwidth of approx. 12.8 GBps. This Section evaluates prefetchers with DRAM bandwidth configurations such as DDR4-3200 (MTPS of 3200) and DDR3-1600 (MTPS of 1600) [29]. Figures 4.21 and 4.22 show the effect of DRAM bandwidth on speedup for L1D and multi-level prefetching, respectively. When moving from 6400 to 1600 MTPS, the performance loss is negligible for all the prefetchers with GAP traces and moderate for SPEC CPU2017 traces (maximum reduction of 4.1% with Berti and Berti+SPP-PPF).

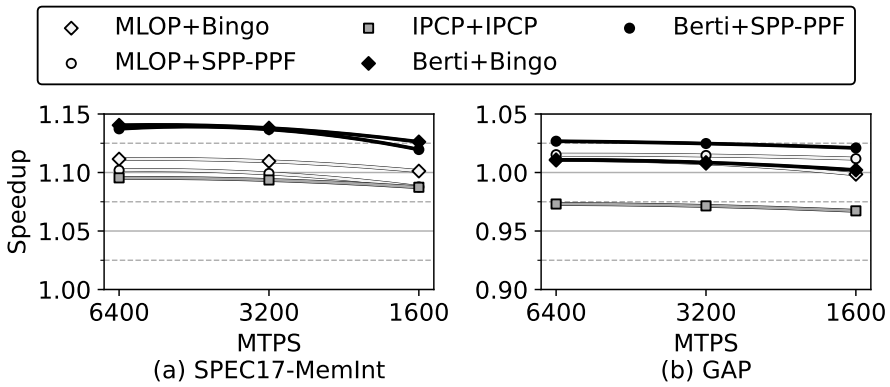


Figure 4.22: Performance of multi-level prefetching in constrained DRAM bandwidth, in MTPS.

4.5.6 CloudSuite Performance

Figure 4.23 shows speedup with the CloudSuite benchmarks for L1D and multi-level prefetching combinations. `Classification` is one benchmark where all the prefetchers fail except Berti, thanks to its high prefetch accuracy. Note that for some of the benchmarks like `cloud9` and `nutch`, even an ideal L1D prefetcher (L1D with a hit rate of 100%) fails to provide significant performance, which shows that there is limited scope for data prefetching. The primary reason for this trend is that the L1D MPKI of CloudSuite without prefetch is low: 6.9 on average, with a maximum of 14.5, while the average L1D MPKI of SPEC and GAP is 42.2 and 83.6, respectively. On the other hand, the L1I MPKI of CloudSuite traces are higher than SPEC and GAP traces.

4.5.7 Interaction With a Temporal Prefetcher

We simulate managed irregular stream buffer (MISB) prefetcher [169], a storage efficient version of ISB [85] at L2 with MLOP, IPCP, and Berti at L1D, as shown in Figure 4.24. ISB is an addresses correlation-based data prefetcher that correlates cache accesses at a new indirection level named structural addresses space. Berti with MISB improves the effectiveness of multi-level prefetching for CloudSuite traces, in particular for `Cassandra` and `Classification`. For SPEC CPU2017 and GAP, MISB performs worse than SPP-PPF with MLOP and Berti at the L1D. Note that the performance improvement with CloudSuite comes with a storage overhead of 98 KB with MISB, out of which 32 KB is used for the metadata cache and 17 KB for the Bloom filter.

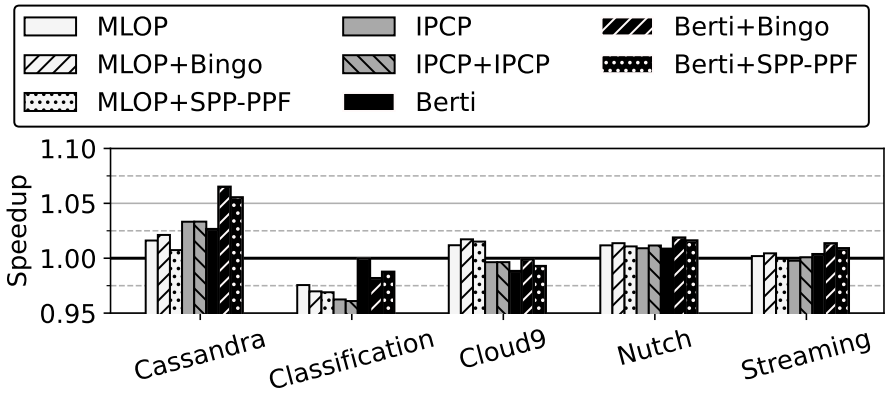


Figure 4.23: Speedup for CloudSuite.

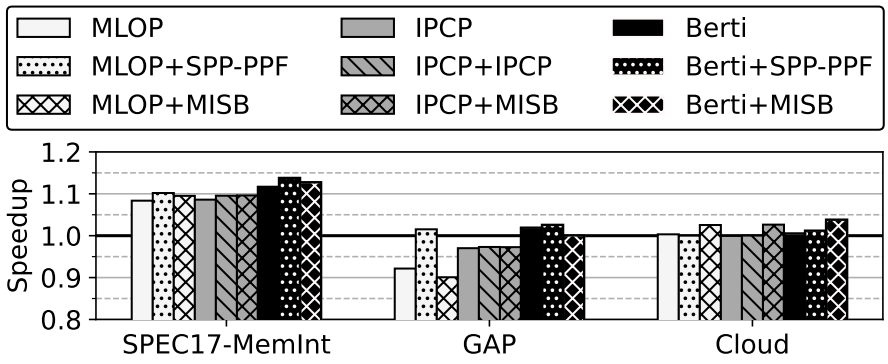


Figure 4.24: Speedup with and without MISB.

4.5.8 Multi-core Performance

Figure 4.25 shows speedup on a 4-core simulated system averaged across 200 randomly generated heterogeneous mixes based on memory-intensive SPEC CPU2017 and GAP traces. Among the L1D prefetchers, Berti performs the best with a performance improvement of 16.2%, outperforming both MLOP and IPCP on average. There are only nine mixes in which MLOP and/or IPCP gain more than 10% over Berti, and `CactuBSSN` is part of seven of these mixes. Berti outperforms competing prefetchers for majority of the mixes that do not have `CactuBSSN`. Overall, Berti performs better because in the case of multicore systems, per core available DRAM bandwidth goes down because of cross-core contention. Thanks to Berti's timely and accurate deltas, it is still able to deliver high coverage even in the presence of shared DRAM bandwidth contention.

Berti at L1D also outperforms other multi-level prefetching combinations

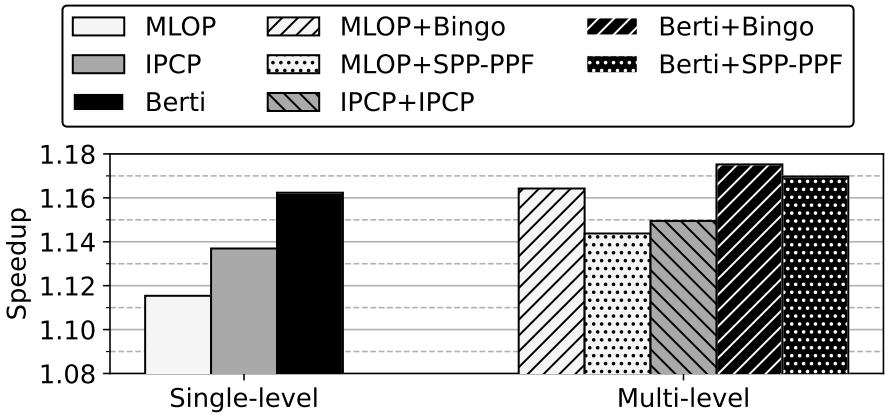


Figure 4.25: Summary of multi-core speedups relative to a system with L1D IP-stride prefetcher.

making a strong case for Berti as an L1D-only prefetcher. Note that Berti outperforms MLOP+Bingo, the combination of the second place and first place prefetchers in the 4-core evaluations at the DPC-3.

4.5.9 Sensitivity to Design Choices

Effect of L1 and L2 watermarks

Figure 4.26 shows the effect of L1 and L2 confidence watermarks on overall speedup averaged across single-core SPEC CPU2017 and GAP benchmarks, normalized to the baseline system. Our chosen watermarks, more than 65% for L1 and in between 35% to 65% for L2 provide the sweet-spot in terms of prefetch accuracy and prefetch coverage. Usage of extremely small or extremely large watermarks affect both coverage and accuracy, and negatively affects speedup. Interestingly, a large number of watermark configurations provide similar benefit in terms of speedup. Our chosen high watermarks provide maximum speedup with the maximum prefetch accuracy.

Effect of the size of Berti tables

Figure 4.27 shows the effect of the size of the Berti tables (history table, table of deltas, and the number of deltas) on speedup. Decreasing the size of the table of deltas by a quarter degrades performance by 12.1%, whereas decreasing the number of deltas by a quarter reduces performance by 1.2%. Also, doubling/quadrupling the size of the tables provides a marginal performance gain. `CactuBSSN` is one outlier where increasing the table sizes to 1024 entries with 1024 sets improve performance by 22%.

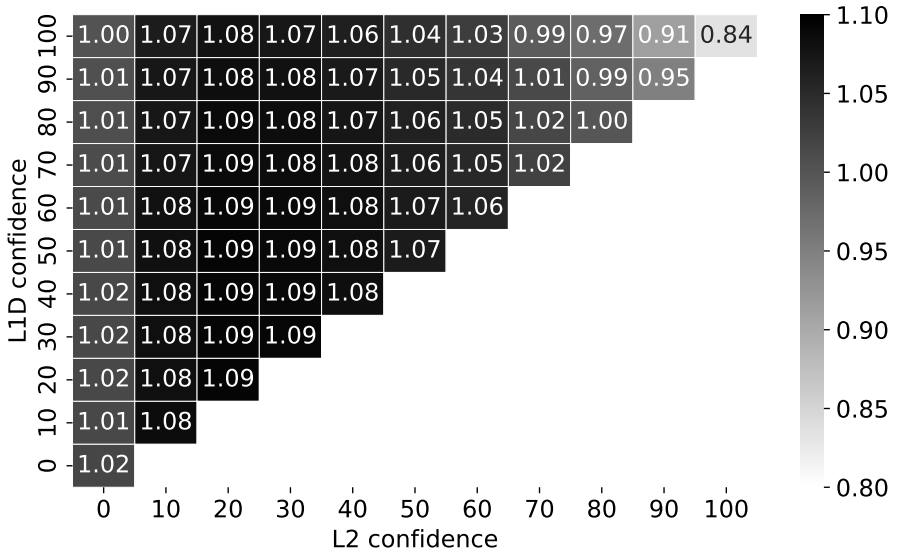


Figure 4.26: Normalized speedup with different L1D and L2 confidence watermarks averaged across memory intensive SPEC CPU2017 and GAP benchmarks. Speedup is rounded to two decimal places (1.085 is rounded to 1.09).

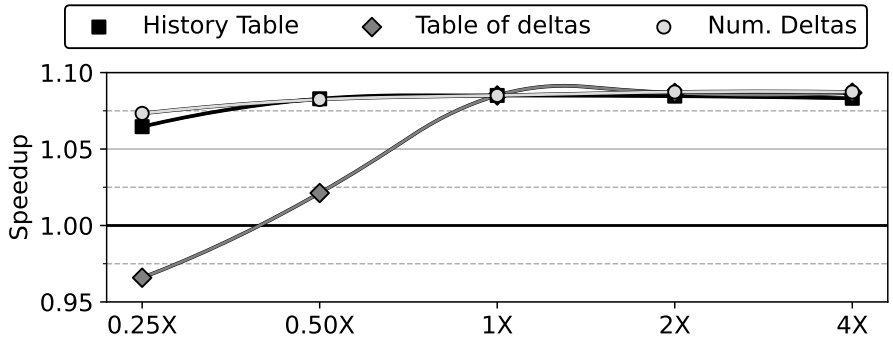


Figure 4.27: Speedup vs. size of Berti tables and number of deltas. 0.25× to 4× correspond to one-fourth and four times, respectively.

Effect of the latency counter

In our evaluations, we use a 12-bit latency counter per line at the L1D. When we increase its size to 32 bits, performance is not improved. However, using a small 4-bit timestamp, we see a performance drop from 1.16 to 1.07, and from 1.02 to 0.98, for SPEC CPU2017 and GAP, respectively.

Effect of cross-page prefetching

As Berti is an L1D prefetcher and operates on virtual addresses, it does cross-page prefetching as long as prefetch requests get a hit in the STLB. To understand the utility of cross-page prefetching, we evaluate Berti, where we do not issue prefetch requests (but keep training) that cross an OS page. We see an average performance drop from 1.02 to 1.01 and 1.16 to 1.10 for GAP and SPEC CPU2017 traces, respectively. The performance drop shows that most of the deltas selected by Berti are within the OS page boundary of 4 KB.

4.6 Related Work

In section 4.5 we presented a quantitative comparison of Berti with recent hardware prefetching techniques [13, 93, 9, 145, 122]. In this Section we compare other relevant prefetching techniques qualitatively.

4.6.1 Temporal Prefetchers

Temporal prefetchers track the temporal order of cache-line accesses (and not the deltas) [89, 74, 85, 149, 161]. Temporal prefetchers usually demand hundreds of KBs of storage, which demands the storage of prefetch metadata in the off-chip memory. Some of the recent works on temporal prefetching are in the pursuit of improving the storage overhead without affecting the prefetch coverage [169, 168]. Berti, on the other hand, incurs a storage overhead of just 2.55 KB per core.

4.6.2 Spatial Prefetchers

Compared to temporal prefetchers, spatial prefetchers are lightweight in terms of storage overhead and usually learn memory access patterns within a small spatial region of a few KBs. Conventional prefetchers like stride [38] and stream [75, 150] are already deployed on commercial processors. Timely Stride prefetching improves the timeliness of conventional stride prefetchers [178]. However, it does not provide better prefetch coverage when compared with state-of-the-art L1D and L2 prefetching techniques. Spatial prefetchers like Spatial Memory Streaming (SMS) [150] (similar to Bingo) usually learn single repeating deltas or bit patterns within a spatial region, where a set bit denotes a cache line that should be prefetched. All these techniques do not consider prefetch timeliness.

Kill the program counter (KPC) proposes a holistic cache replacement and prefetching framework [94]. However, the prefetching technique is similar to SPP, with similar performance improvements as SPP. Multi-level adaptive prefetching based on performance gradient tracking [137] (3rd place in DPC-1 [1]) is one of the first proposals that propose a correlation between an IP and delta sequences. DSPatch [12] tunes a hardware prefetcher based on available DRAM bandwidth and selects memory access patterns based on prefetch accuracy (if

the available DRAM bandwidth is low) and prefetch coverage (if the available DRAM bandwidth is high). Overall, SPP-PPF performs marginally better than SPP+DSPatch.

4.6.3 Machine Learning for Hardware Prefetching

Machine learning (ML) has been used for microarchitecture research, and ML techniques for data prefetching have been proposed in recent years [11, 148]. In ISCA 2021, a prefetching competition with ML techniques shows that non-ML techniques still outperform with limited storage. However, ML techniques have the potential to learn highly complex memory access patterns, and Pythia [11] shows that with a high performing L2 prefetcher. Berti is an L1D prefetcher in contrast to Pythia, and with Berti at the L1D, we find negligible performance improvement with Pythia (less than 1%).

4.6.4 Prefetch Filters and Throttling Mechanisms

Similar to PPF [13] and DSPatch [12], there are proposals that control the aggressiveness of prefetchers by controlling its prefetch degree and distance, or decides whether to prefetch into the L2 or to the LLC [42, 6, 123, 125, 124, 66]. These techniques incur additional storage and perform well for conventional prefetchers with low prefetch accuracy. However, with Berti, the accuracy is significantly higher than prior prefetching techniques, and the implicit confidence mechanism acts like a prefetch throttler.

4.7 Concluding Remarks

In this chapter we present Berti and made a case for an L1D prefetcher based on local, timely deltas. Berti learns the best delta to prefetch, keeping timeliness (in the form of time to prefetch an addresses) and prefetch accuracy in mind. We showed that Berti could learn varieties of memory access patterns. We quantified the effectiveness of Berti across SPEC CPU2017 and GAP workloads, and showed high prefetch accuracy and timely prefetching into the cache hierarchy. On average, Berti outperforms state-of-the-art L1D and L2 prefetchers. Berti is equally effective even in the constrained DRAM bandwidth scenarios and also for multi-core mixes. Berti consumes the least dynamic energy at the memory hierarchy among all state-of-the-art prefetchers. In summary, Berti provides high prefetch accuracy, timely prefetching, and good coverage with a limited storage overhead of 2.55 KB per core.

Chapter 5

Synchronization Strategies on Many-Core Systems

Multicore processor design, ubiquitous in all market segments from cell phones to supercomputers, has driven a paradigm shift in how applications are developed. If developers want to achieve the highest possible performance in an application, it must be multithreaded to take advantage of the full compute capacity of the system. Conventional inter-thread synchronization strategies using locks have proven to be efficient in systems with a small number of cores/threads, but they require considerable programming effort. In addition, at times of high concurrency in shared resource access, they cause an invalidation storm that consumes shared resources intensively, including access to several levels of the hierarchy and the on-chip interconnection network. To facilitate this synchronization, a new mechanism, transactional memory, appeared 30 years ago. This mechanism promised to be an efficient and easy-to-use alternative to established technologies. However, the increase in the number of cores per processor raises questions about the scalability of all these strategies, mainly due to the saturation of the shared resources. This chapter presents an extensive scalability study of the most widely-used synchronization mechanisms in a many-core CMP system, evaluating performance and latency. The experiments show that hardware transactional memory (HTM) matches fine-grained locking performance and scales better as the number of threads increases. We also analyze the impact of simultaneous multithreading (SMT) technologies on HTM performance. We propose a new cache replacement policy for L1D cache that takes into account the states of each cache line and aims to mitigate SMT-induced transactional overflow aborts.

5.1 Introduction

Improvements in manufacturing technology and the integration of a greater number of cores have made multicore systems the norm for virtually all kinds

of commodity computing devices, ranging from mobile to large-scale server machines. Currently the CPUs can reach hundreds of native threads, e.g. AMD Rome with 128 threads [59] or IBM Power 10 with 120 threads [61]. In addition, it is common to have multiple sockets on a shared memory server, IBM Power 10 allows up to 16 sockets sharing memory [61], which further increases the number of threads available in the system. This paves the way for a higher degree of parallelism and, thus, for better performance across concurrent applications. However, efficient concurrency comes at a price, oftentimes paid in increased complexity for the synchronization mechanism. Classical synchronization systems, such as fine-grained locks or lock-free mechanism, require an in-depth understanding of the concurrent program's structure and its internal interactions to achieve good performance. This complexity leads to unsound implementations, faulty executions, deadlocks...

In recent decades, a new synchronization paradigm has captured the interest of the community: the transactional memory. TM proposes a simpler way of reasoning about application concurrency. The concept of transactional memory was introduced almost 30 years ago by Herlihy and Moss [70]. However, since no hardware infrastructure was readily available to provide TM, software transactional memory (STM) was the first heavily studied category. Its usefulness in real-life scenarios was fiercely disputed [21, 21]. In 2009 Sun Microsystem [34] announced a multicore processor with support for HTM, since then it has been progressively adopted by all major hardware vendors, quickly becoming a hot topic.

Numerous studies have been performed on the performance of different synchronization mechanisms: e.g., comparisons between TM systems [36], between locking techniques [151] or even HTM versus classical locks and atomic primitives [128]. However, there are no studies that analyze the scalability of all synchronization mechanisms in many-core systems and quantify the impact of SMT. This chapter aims to bridge this gap, with an extensive study on the scalability of various synchronization mechanisms paying special attention to the impact of SMT. Furthermore, while SMT is recognized as a limiting factor for HTM performance [164], its impact is not clearly quantified and no solution has been proposed to mitigate it.

The rest of the chapter is organized as follows: the next Section presents the context of this work. Section 5.3 evaluates the scalability of synchronization mechanisms on data structures. Section 5.4 presents a real-world use case of hardware transitional memory, comparing its performance with fine-grain locks in real workloads. Section 5.5 studies the negative impact of SMT on hardware transactional memory and presents a L1D cache replacement algorithm to mitigate it. The chapter ends with related work, Section 5.6, and conclusions, Section 5.7.

Table 5.1: Synchronization mechanisms with their benefits and drawbacks. *Non-portable* stands for implementations that are dependent on the architecture.

Mechanims	Benefits	Drawbacks
Locks	Well-known, good performance	Error prone, complex
Atomic Instructions	Good performance	Complex, non-portable
STM	Ease-of-use	Poor performance
HTM	Good Performance, ease-of-use	Limited, non-portable

5.2 Background on Synchronization Strategies

We evaluate the behavior and performance of four different synchronization mechanisms: locks, atomic instructions, STM, and HTM. Table 5.1 summarizes the benefits and the drawbacks of these synchronization strategies.

5.2.1 Classical Strategies

Most parallel applications are implemented using locks or atomic instructions. A lock controls the access to data regions shared by multiple threads. Programmers usually use multiple locks (*fine-grain*) to increase the parallelism and performance of the applications. However, fine-grain locks require in-depth knowledge of the application and errors such as deadlocks or race conditions are not uncommon. Lock-free algorithms rely on atomic instructions, such as *compare-and-swap (CAS)*, to manage concurrency among threads. They do not suffer from lack of scalability, but require extensive knowledge on the underlying processor architecture and the program's structure.

5.2.2 Transactional Memory

According to the literature [127], transactional memory seems to be a promising mechanism for synchronizing processes. TM enables the programmer to mark a section of code, specifying that it has to be executed as a transaction. The TM system guarantees that transactions are executed atomically. TM presents two main benefits over classical synchronization strategies: **(1)** ease-of-use, while also avoiding well-known concurrency issues, such as race conditions; **(2)** optimistic execution, allowing for a higher degree of parallelism.

Transactional memory comes in three flavors: software (STM), hardware (HTM), and hybrid (a combination of both software and hardware). In general, STM libraries [40, 39, 46, 46, 111, 35] instrument applications' code to detect and solve memory conflicts. The high flexibility given by its software implementation is tarnished by its instrumentation performance overhead. HTM [135, 136, 107, 175], on the other hand, aims to address this performance concern, but its hardware constraints limit the flexibility of the solution. In 2009, Sun Microsystems was the first company to announce a multicore processor with HTM support, codenamed Rock [34]. Later, HTM support was implemented

in commercial processors by other hardware vendors, such as Intel (starting with the Haswell family) and IBM (POWER [98], Blue Gene/Q [162], the zSeries [83]).

Intel's HTM implementation goes by the name of Transactional Synchronization Extensions (TSX). It has two different interfaces: Intel Hardware Lock Elision (HLE) and Intel Restricted Transactional Memory (RTM). Intel TSX uses the cache hierarchy to track write and read-sets [65], and the coherence protocol to detect conflicts at a cache-line granularity. It is a best-effort TM system. Intel provides assembler-level instructions [80], e.g. `XBEGIN`, and intrinsics [81] for using RTM. These instructions and intrinsics provide information to the user about the execution of the transaction: if it successes, if it fails, reason for abort.... IBM provides a different HTM implementation in its IBM POWER line of products. Instead of using the cache hierarchy directly for tracking transactional read and write accesses, IBM implements a per-core *transactional buffer* [48]. This buffer contains content-addressable memory and it is linked to the L2 cache. With this transactional buffer in place, also called *TM-CAM*, IBM machines allow for a 64-cache line transactional capacity. In addition to this, another read tracking structure is made available for transactions with large read sets. Finally, IBM introduces *rollback-only transactions* (ROTs), which allow only for the write set to be tracked, reads being performed non-transactionally. While this strategy significantly increases the capacity per transaction, it also fails to provide strong consistency guarantees. More precisely, read-write conflicts are not guaranteed to be detected, potentially leading to incorrect outcomes unless used as recommended [76]. Intel TSX and IBM's implementation is a best-effort TM system.

Best-effort TM system does not guarantee that a hardware transaction will ever commit. Since hardware transaction can fail for multiple reasons: a line written in a transaction is written or read by another thread (*conflict*), hardware frameworks do not have enough capacity to track read/write sets (*capacity*), purposely cancelled by the developer (*explicit*), ... The developers need for a software fallback to provide progress using another synchronization mechanism. Most often, a global lock is used inside the fallback path, but more complex fine-grained locking schemes can be exploited for added efficiency [20, 133]. There are several considerations when providing a transactional fallback-path. On the one hand, the abort reason needs to be taken into account when deciding whether the transaction should be restarted, depending on the chances of commit on a retry. On the other hand, the number of retries needs to be well-balanced: if there are too few retries, the transaction may be serialized prematurely; if there are too many, the eventual commit may not amortize the cost of the rollbacks. Finally, a back-off time may be inserted between retries, in order to reduce conflicts.

Table 5.2: Main characteristics of the workstation used in the evaluation.

Processor	2×Intel Xeon Gold 6238T
Threads	2 × 44 (88)
Family	Cascade Lake
Speed	1.9 GHz
Cores	2 × 22 (44)
TurboBoost	No
L1D	32 KiB 8-way
L2	1 MiB 16-way
LLC	22×1.375 MiB 11-way
Mem	192 GiB 6 channels
OS	Ubuntu 20.04
Kernel	5.4.0
NUMA policy	Default
Governor	Performance
Compiler	GCC 9.3.0

5.3 Scalability Analysis of Synchronization Mechanisms

This sections presents the scalability study of the syncronization strategies on a many-core system in terms of application throughput and operation latency. We focus on two widely-used concurrent data structures, a hash-table and a binary search tree, representing typical building blocks in the development of large-scale systems. We implement them with four different synchronization mechanisms: atomic primitives [64, 72], fine-grain locks [67], STM (TinySTM engine), and HTM (Intel RTM). All implementations are optimized, avoiding typical parallel applications issues such as false-sharing, and are lock-free for lookup operations[1].

5.3.1 Experimental Setup and Methodology

System description. Experimentation has been performed on a server with two Intel Xeon Gold 6238T (Cascade Lake). Each CPU has 22 cores that can execute up to 44 threads. In total our system can run up to 88 threads, with a 192 GiB main memory distributed in 6 channels and shared among all the threads in a non-uniform memory access configuration (NUMA). The server runs Ubuntu 22.04 with kernel 5.4.0. We use the default NUMA policy and the power governor is set to performance. Table 5.2 summarizes our experimental setup.

Software implementation details. We implement the HTM version we use C/C++ intrinsics, a global spin-lock without back-off as fallback, and

[1]The code is available at: https://github.com/agusnt/Synchronization-Strategies-on-Many-Core-SMT-Systems

a maximum of 10 retries before jumping to the fallback path. This is the simplest and most common fallback path implementation. This strategy gives us a lower-bound on HTM performance. In addition to this, we follow the recommendations laid out by Intel [79] and Bonnichsen et al. [17] to increase the chances of commit: small transactions, no system calls inside a transaction, and small memory footprint.

The lock-based version uses standard `pthread` mutex locks to synchronize its critical sections. According to the extensive study on lock algorithms done by Guiroux et al. [56], `Pthread` locks are amongst the best performing locking structures for a variety of applications. We thus decide to rely on this implementation, rather than incurring extra overhead from a lock interposition library such as LiTL [55] or introducing unneeded complexity from a home-brewed locking algorithm. The lock-free version relies on *compare and swap* instruction.

Workloads. We evaluate the concurrent data structures with three different workloads: **(1)** 100% lookup operations; **(2)** 80% lookups and 20% updates (10% insertions and 10% deletions); and **(3)** 50% lookups and 50% updates (25% insertions and 25% deletions). For brevity, we call these workloads *AllLookup*, *Update20*, and *Update50*, respectively. The workloads consist of 2^{26} predefined operations. To minimize execution variability, the sequence of operations is the same in all experiments, regardless of synchronization mechanism. The data structures are populated before each experiment with previously-generated random elements: 2^{18} for the hash-table and 2^{17} for the binary search tree.

Each workload is executed several times by varying the number of threads. The threads are distributed so that they occupy as many cores as possible. In experiments on up to 44 threads, we pin each thread to the first logical core on each NUMA node in turn. Above this thread count, we move to the second logical core and continue pinning the threads in the same order as before on each NUMA node.

Metrics. For the analysis we rely on three metrics: (i) **throughput** measured in operations per second (performance), (ii) **latency** defined as the time it takes for an operation to be executed, and (iii) **HTM events** as the percentage of transactions committed and aborted, the abort rate being further split according to the abort reason.

For the evaluation of throughput and HTM events, we execute each experiment <synchronization method, workload, thread count> 11 times and take the median of all executions. The latency analysis is based on all latency data points per workload execution, on 88 threads. For simplicity and readability, we present the data from a single run per implementation, but note that the results are consistent over multiple runs.

5.3.2 Concurrent Hash-Table

We implement a fixed-size hash-table with 2^{17} buckets. In order to solve potential key conflicts, each bucket contains a sorted linked-list of keys.

Throughput. We first compare the four different hash-table implementations in terms of throughput. Figure 5.1 shows the throughput in *Million Operations Per Second (MOPS)* on the Y axis and the number of threads on the X axis. The vertical lines mark the following transitions of interest: **(1)** from 22 to 23 threads, i.e., from using one processor to two processors; **(2)** from 44 to 45 threads, i.e., from no SMT to using SMT on the first processor; and **(3)** from 66 to 67 threads, i.e., from no SMT on the second processor to using SMT on both processors. The relative standard deviation is 5.4% on average across all workloads, contention levels and implementations, with a maximum of 18%. Only 6 datasets out of 108 present outliers (one dataset being composed of all 11 results of an experiment). More precisely, in all cases there is at most one outlier, which we believe is due to an anomaly on the machine where experiments were running. The outliers were computed with the `zscore` function.

From a performance perspective, there is no clear winner across all implementations, workloads and number of threads. For the *AllLookup* workload, all versions behave similarly, since they are all equally lock-free. The slight quantitative dissimilarity between the lines in the graph is explained by the inherent differences in the implementation of the concurrent algorithms.

The *Update20* workload shows comparable throughput for the lock-free, fine-grained locking, and HTM implementations. All three synchronization mechanisms scale with the number of threads. Starting with 45 threads, HTM performs better than the classical methods. On 88 threads, HTM achieves 18% and 27% higher throughput than the fine-grained locking version and the lock-free implementation, respectively. STM is the only synchronization mechanism not able to scale further than 22 threads. On 88 threads, the use of STM leads to a performance degradation of more than 5x on average compared to the other versions, mainly because of the overhead of instrumenting all memory accesses. We observe a performance drop (9% on average) across all implementations when we start using the second processor (in both transition points, 22 – 23 threads and 66 – 67 threads, respectively). The communication between the two processors and its impact on the cache can explain this behavior. Using the second set of logical cores (i.e., 45 threads and more), on the other hand, has a visible impact only on the lock-free and locking implementations.

In contrast, the more contended workload, *Update50*, has a bigger impact on performance. Up to the point where the second logical core starts being used, all synchronization mechanisms except STM follow the same trend with similar throughput. While HTM continues to scale on more than 45 threads, the lock-free and fine-grained locking versions stop scaling at 44 threads and their throughput starts decreasing when running on more than 66 threads. When the second logical core on the second NUMA node is used (transition point **(3)**),

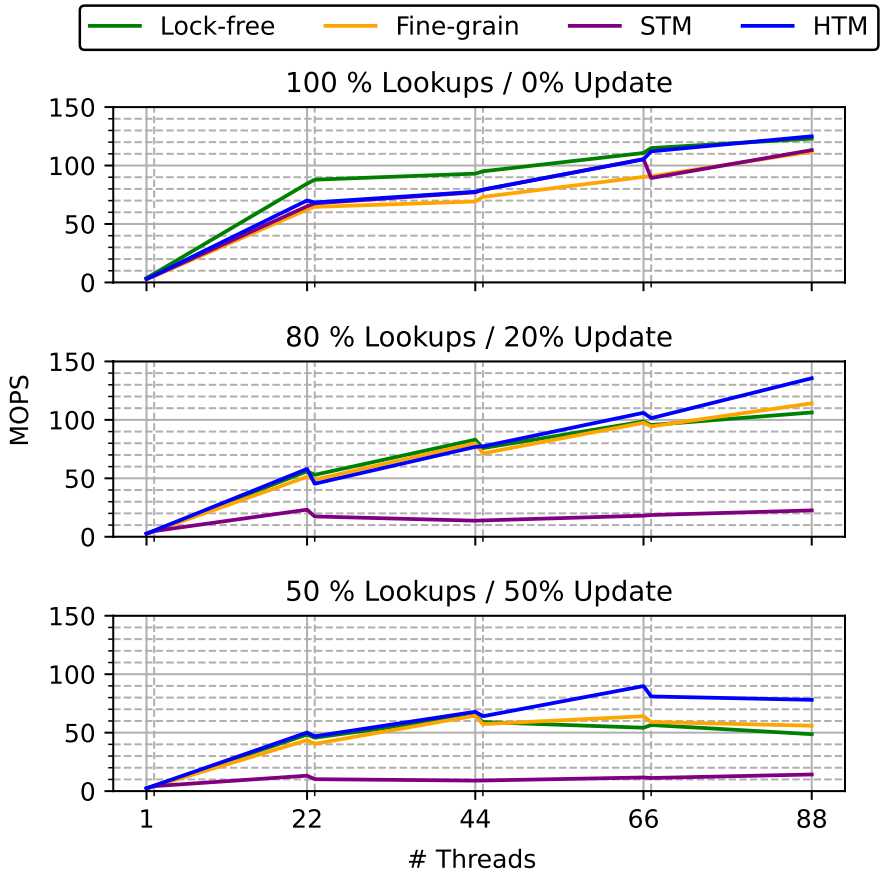


Figure 5.1: Throughput of the different synchronization mechanisms on a concurrent hash-table.

the performance of the HTM-based implementation has a drop of $\sim 11\%$ and the throughput starts decreasing. As with the *Update20* workload, the STM version does not scale after 22 threads. In addition, this experiment reveals the same performance drops at the transition points of interest, with the HTM version having also a negative spike at transition point **(2)**. We attribute this to the increased contention between transactions that have to share the same core.

In conclusion, the implementation using HTM as synchronization benefits of more parallelism, in contended scenarios in particular. While the scaling slows down after 66 threads in our configuration, it still shows 30% to 60% higher throughput than the fine-grained locking and, respectively, the lock-free versions.

Latency. We measure the latency of each operation with the `rdtsc` instruction and remove outliers with the `zscore` function. We represent the latency of update operations on 88 threads with a CDF in Figure 5.2. The Y axis shows the fraction of operations that execute in less than $x\mu$s (X axis, logarithmic scale). The vertical lines indicate the maximum measured value for a given implementation.

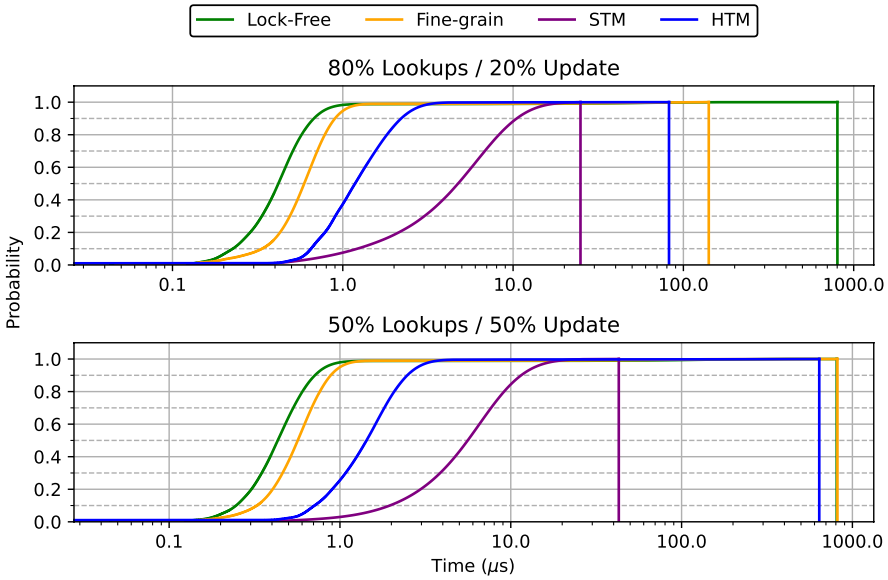


Figure 5.2: Update operation latency (88 threads) for the synchronization mechanisms on a concurrent hash-table.

Lock-free, fine-grained locking, and HTM implementations present a long tail, more prominent for the update-intensive workload (*Update50*). HTM always has lower tail-latency than the classical synchronization mechanisms. The lock-free implementation consistently shows a considerably larger tail-latency than HTM and lock-based versions. Many threads spinning on the same atomic primitive (e.g., CAS) in order to perform update operations can explain this result. Thus, on the *Update20* workload, the tail-latency of the HTM version is 2x lower than that of fine-grained locking, and 10x lower than that of the lock-free implementation. The tail-latency for these three synchronization mechanisms becomes comparable on an update-intensive workload, such as *Update50*.

In contrast, the STM implementation has a short tail, but 99% of operations take one order of magnitude longer than the other three synchronization mechanisms for the same interval. We believe this is due to our STM configuration: transactions abort immediately on conflict, and the write-set implementation facilitates low-overhead aborts and high-overhead commits. Thus, in both

workloads, less than 30% of the STM operations are executed in under 3 µs, as opposed to 97% for the other synchronization methods.

HTM events. Figure 5.3 shows the fraction of committed and aborted transactions on the Y axis and the number of threads on the X axis. The abort rate is further split into multiple abort causes: capacity aborts (physical limitation to the size of the transaction), conflict aborts (concurrent accesses to the same memory address), and other (explicit aborts when encountering an already-acquired lock or system-level interrupts, debug instructions, I/O operations). We only show these fractions for the mixed workloads (*Update20* and *Update50*) because the lookup operations alone do not perform any transactions.

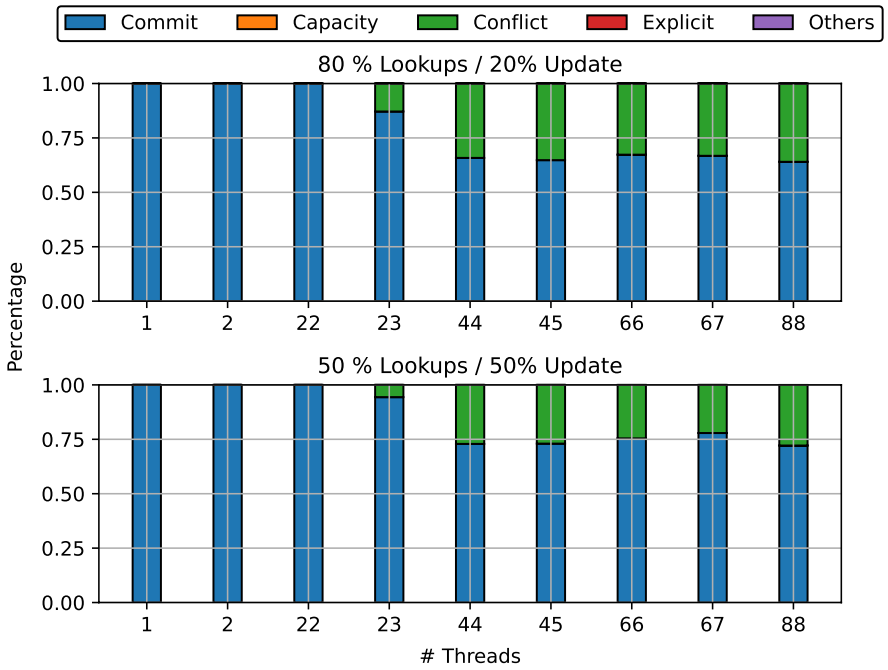


Figure 5.3: HTM events of the different synchronization mechanisms on a concurrent hash-table.

Almost all transactional aborts are due to memory conflicts. Most of them are generated by the interaction with the global lock employed in the fallback path. More precisely, all transactions monitor the state of the global lock; if one transaction repeatedly aborts for any reason and takes the fallback path, it acquires the lock, changing its state; this state change conflicts with the monitored value in all other running transactions, causing them to abort. The ratio of conflict aborts increases with the number of threads in both workloads (e.g., from 12% on 23 threads to 35% on 88 threads for *Update20*). However, we observe a lower abort rate for the update-intensive workload than for *Update20*.

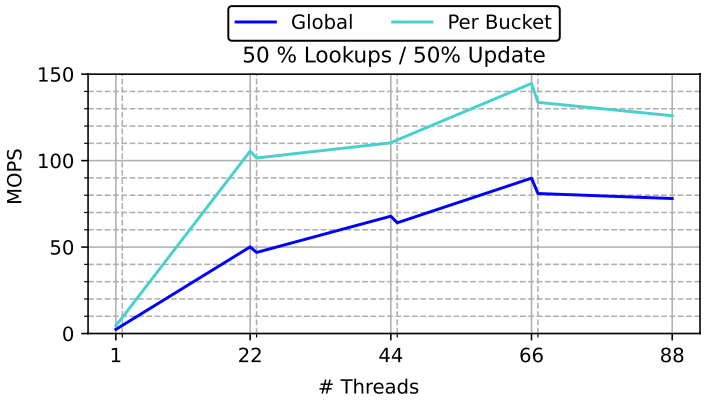


Figure 5.4: Global fallback lock vs. per-bucket fallback lock.

We believe this is due to the fact that part of the delete operations will not hit an existing element in the hash-table, while the insert operations will keep adding elements. Iterating over a larger data structure favors more and longer transaction-free lookups and, thus, fewer opportunities for conflict.

There are two main factors that lead to infrequent aborts in the other categories for this application: first, we avoid overflows by carefully planning the contents of the transactional regions and taking into account HTM size limitations; second, our hash-table implementation follows closely the aforementioned Intel guidelines, thus avoiding aborts due to unfriendly instructions. We further discuss an optimization for the fallback algorithm.

HTM fallback discussion. To avoid restarting all running transactions every time a thread takes the fallback path, a more fine-grained approach can be implemented. This may represent a convenient trade-off between simplicity and performance, depending on the application. In the case of a concurrent hash-table, this optimization is easily achievable by replacing the global lock in the fallback path with a *per-bucket lock*. This reduces the set of conflicting transactions to only the potential few that are accessing the same bucket. An even finer-grained approach (e.g., per-object lock) would significantly increase the complexity of the code with negligible performance improvement.

We briefly evaluate the fallback path optimization. Figure 5.4 compares the throughput of the two approaches, using a global and a per-bucket lock, on the *Update50* workload. We observe up to 2x throughput improvement (on 22 threads) and 62% higher throughput on 88 threads. Moreover, while the performance of the two implementations follows the same trend, we note the lack of negative spike at transition point **(2)** for the per-bucket approach. Statistics on HTM events show a drastic reduction of conflict aborts, which are consistently under 1%.

5.3.3 Concurrent Binary Search Tree

We evaluated the BST following the same methodology applied to the hash-table. The BST results confirm the findings from Section 5.3.2 and are illustrated in Figures 5.5, 5.6, and 5.7.

Throughput. Overall, the measurements performed on the BST show more variability than those of the hash-table. The relative standard deviation is 8.5% in average across all workloads, contention levels and implementations of the tree data structure, with a maximum of 32%. We found and excluded a total of 3 outliers over all datasets with the `zscore` function.

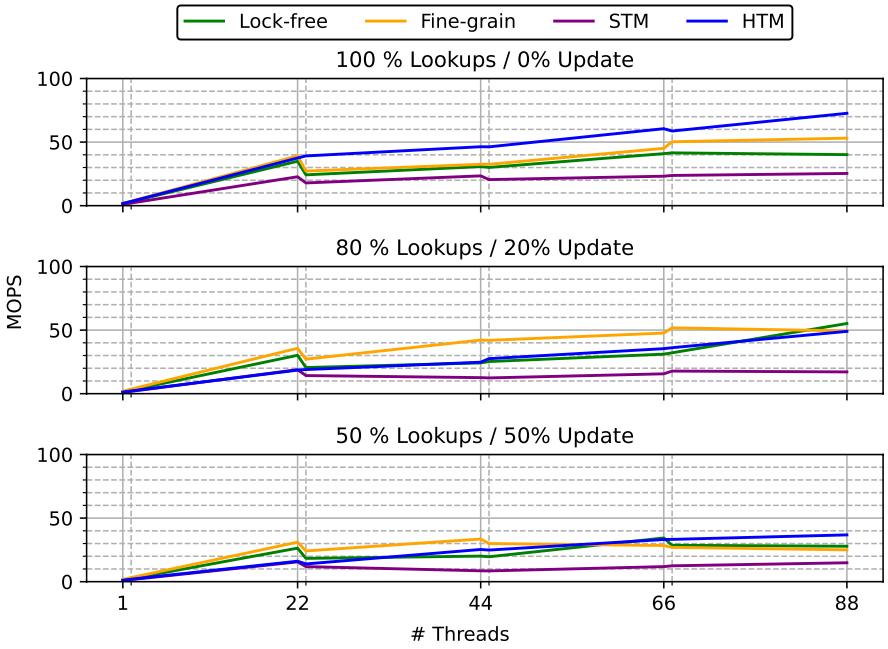


Figure 5.5: Throughput of the different synchronization mechanisms on a concurrent BST.

As before, the STM implementation consistently shows the worst performance, see Figure 5.5. For the *Update20* workload, the fine-grained locking implementation provides the highest throughput across the entire thread range. At transition point **(3)**, the locking implementation's scaling starts to slow down, while the HTM and lock-free versions continue to scale constantly. On 88 threads, all versions have comparable performance. By contrast, the update-intensive workload shows similar throughput for these three implementations throughout the experiment. The locking version stops scaling at transition point **(2)**, when the second logical core is enabled. Similarly, the lock-free implementation does not scale over 65 threads. The HTM version scales constantly

until 88 threads, where it provides 53% better throughput than the lock-based version and 37% better than the lock-free one.

The state at transition points generally follows the same pattern as in the hash-table experiment. It is less prominent for the HTM-based implementation. The classical synchronization versions do not have a visible performance drop at transition point **(3)** for less contended scenarios.

Latency. Consistent with the results in the hash-table experiment, HTM exhibits tail-latency comparable to fine-grained locking, regardless of workload. The tail-latency for the lock-free implementation varies significantly with contention: it is 8x larger than that of the HTM version on the update-intensive workload, and 2x lower on the less contended one. We believe this variation comes from the nature of the lock-free algorithm. Since an updating thread is not blocking the portion of the tree it is working on, it may need to repeatedly return and restart its subtree traversal if other threads manage to change the values and the placement of the nodes before it applies its update. The update-intensive workload increases the chances that multiple back and forth iterations will take place for any update operation, thus resulting in the observed long tail, see Figure 5.6.

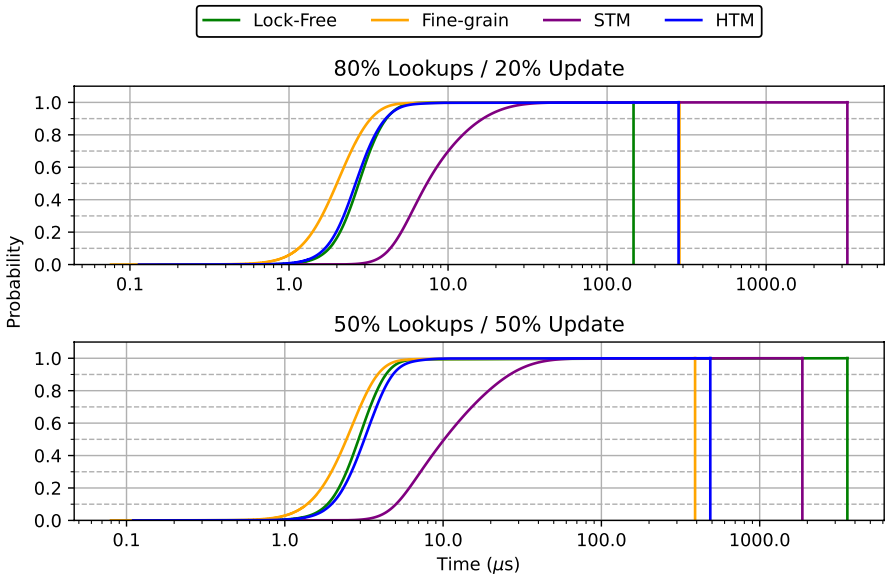


Figure 5.6: Operation latency (88 threads) of the different synchronization mechanisms on a concurrent BST.

Another interesting behavior is presented by STM. Generally, when implementing concurrent data structures with STM, each operation is entirely encapsulated in a software transaction. Thus, operations have higher latency for a more complex tree structure, e.g., if a tree traversal is needed, than for a

hash-table. As such, in this scenario, STM presents between 4x and 10x higher tail-latency than HTM, depending on the workload. Moreover, 90% of STM operations still take at least 3 times longer than those of other versions for the same interval.

HTM events. Figure 5.7 shows the ratios of transactional commits and aborts for different degrees of contention. The breakdown on abort types reveals a majority of conflict aborts. We explain this in the same way as for the hash-table. Similarly, we observe a lower abort rate for the update-intensive workload than for the *Update20* workload. This is consistent with our observations on the concurrent hash-table.

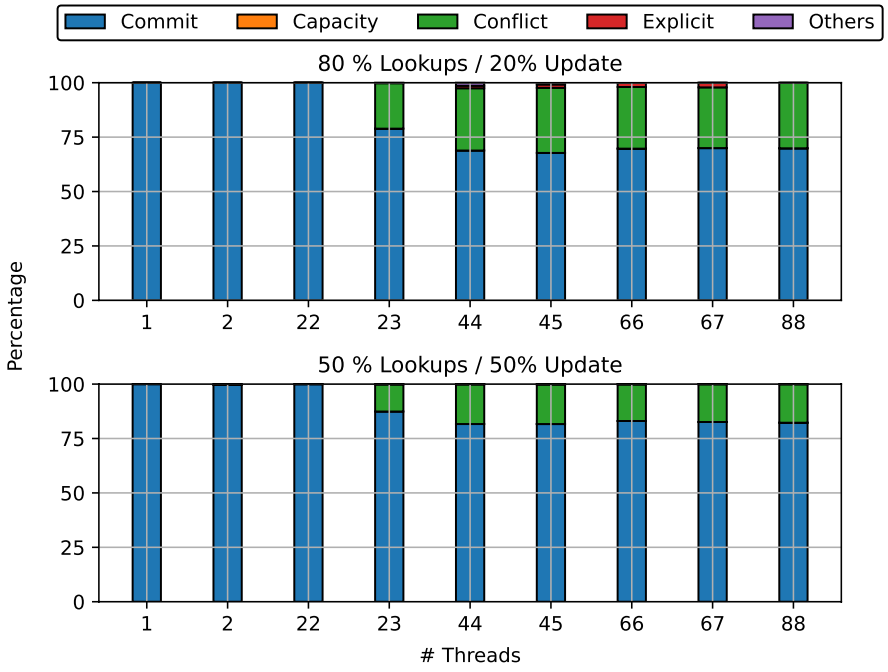


Figure 5.7: HTM events of the different synchronization mechanisms on a concurrent BST.

5.4 Case Study: HTM Ease-of-Use

In this section, we compare the performance of fine-grain locks and HTM on realistic workloads. PARSEC 3.0 [14] is one of the most widely-used benchmark suites for evaluating multicore systems. It consists of 13 scientific real-world parallel applications.

The benchmarks being originally synchronized with a locking mechanism, the strategy we adopt for HTM synchronization is lock-elision, to allow for

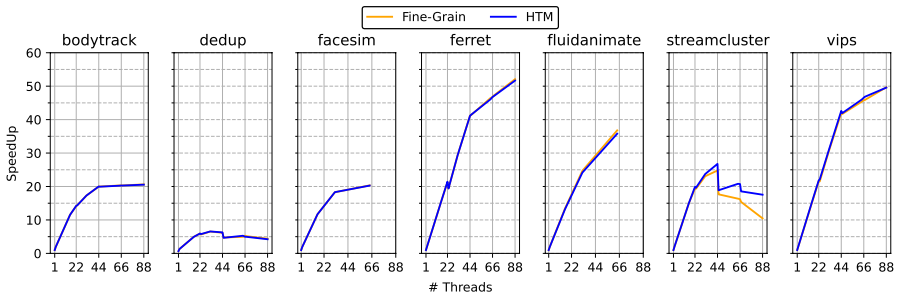


Figure 5.8: HTM and fine-grained locking evaluation on PARSEC 3.0 benchmarks.

backward compatibility and minimal code modifications. Since we want to compare lock performance against HTM, we use glibc GNU/Linux glibc library. The glibc library of GNU/Linux has a readily integrated version of HTM lock-elision, which can be enabled by setting the `GLIBC_TUNABLES` environment variable to `glibc.elision.enable=1`. The glibc implementation uses Intel RTM for performance and flexibility reasons [73]. This mechanism allows us to evaluate HTM in the context of a highly-optimized benchmark suite without minimal code modifications. We also modify the glibc library to retrieve the abort reason of failed transactions.

We select seven benchmarks from the PARSEC 3.0 suite to drive our experiments: `bodytrack`, `dedup`, `facesim`, `ferret`, `fluidanimate`, `streamcluster`, and `vips`. We choose this subset because the benchmarks use numerous lock-based synchronization points (> 100) [15]. These locks are then being elided in the HTM version, providing us with sufficient information to analyze the HTM behavior. We exclude the `x264` benchmark due to runtime errors on our setup. All benchmarks are compiled with the recommended flags. We execute each benchmark three times using the native input files and take the median value.

Figure 5.8 shows the speed-ups achieved by our subset of benchmarks for different numbers of threads. We set the maximum value on the Y axis at 60 for all plots in order to facilitate a direct comparison between the graphs in the figure. All benchmarks are executed on up to 88 threads, except for `facesim` and `fluidanimate` which only allow powers of two as input for the number of threads. Overall, HTM performance is on par with highly-optimized fine-grained locking. Only three of the PARSEC 3.0 benchmarks in our subset show a non-negligible difference between the standard version (fine-grain locks) and the HTM version (lock-elision). More precisely, in `fluidanimate` fine-grain locks perform slightly better (around 2% better on 64 threads) than the HTM version. In contrast, the `vips` benchmark exhibits better speed-up (1% on 66 threads) when using HTM. `Streamcluster` shows a more acute difference in HTM's favor (7% better speed-up on 44 threads, going up to 71% on 88 threads).

5.5 SMT Impact on HTM Capacity Aborts

Most HTM implementations use the cache hierarchy to track their read and write sets. With SMT enabled, threads running on the same core share the private cache resources. When transactions are involved, this translates to a significant reduction in the working-set size limit of the transactions [164]. This is typically reflected by an increase in capacity aborts. Moreover, the impact is more prominent with the increasing number of hardware threads per core.

In this work, we aim to quantify the SMT impact on real-world applications. To this end, we measure the capacity aborts of PARSEC 3.0 benchmarks using lock-elision. As opposed to the applications in Section 5.3, manually optimized with respect to the contents of the transactions, this analysis addresses transactional behavior in realistic scenarios, where transactional contents cannot be controlled or planned. While the number of capacity and random aborts was negligible in the concurrent data structures execution, we expect an increase in the rate of these categories when running real applications. We start with the subset presented in Section 5.4. We use two SMT threads per core in our experiments. The applications are executed without and with SMT (we pin one and two threads per core, respectively). We analyze the distribution of aborts with the no-SMT and SMT versions.

5.5.1 Evaluation

Figure 5.9 shows the number of HTM aborts in the PARSEC 3.0 benchmarks. The abort reasons are split in three categories: capacity, conflict, and other. The latter contains explicit aborts and random aborts due to unfriendly instructions or OS interference. Most benchmarks are dominated by conflict aborts. Two applications show a significant amount of capacity aborts: `dedup`, with up to 21% and `vips`, with up to 11%, of all transactional aborts. These benchmarks have one particular property in common: they access large memory areas in the critical sections, i.e., inside transactions. More precisely, they process chunks of data for (de)compression and image transformation. In contrast, the other benchmarks in the suite employ HTM mainly for synchronizing access to shared flags and variables. As such, we will further focus on the subset of two applications that need large cache capacity to store their transactional working set.

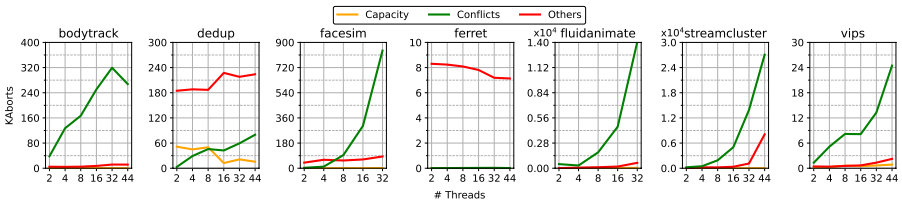


Figure 5.9: Number of aborts per category in the Parsec 3.0 suite with SMT.

Figure 5.10 shows the number of capacity aborts per giga-instruction for different numbers of threads, with and without SMT. We illustrate the ratio between the two versions (SMT and no-SMT) with numbers above the bars. When using SMT, both benchmarks show a considerable increase in capacity aborts regardless of thread count. For `dedup` the maximum impact is recorded on 16 threads, with 16x more capacity aborts when using SMT. `Vips` shows a dramatic 69x increase in capacity aborts on two threads for the SMT version. This ratio decreases with the increasing number of threads down to 3x. We believe this is due to increased thread contention that favors transactional conflict aborts before the tracking structures overflow.

In conclusion, we have observed that SMT can cause a significant increase in the number of aborted transactions due to lack of resources. This holds true even for carefully designed applications such as those of the PARSEC 3.0 suite. This issue can be much more critical in other areas. Wang et al. [164] have shown that for in-memory databases capacity aborts are a limiting factor.

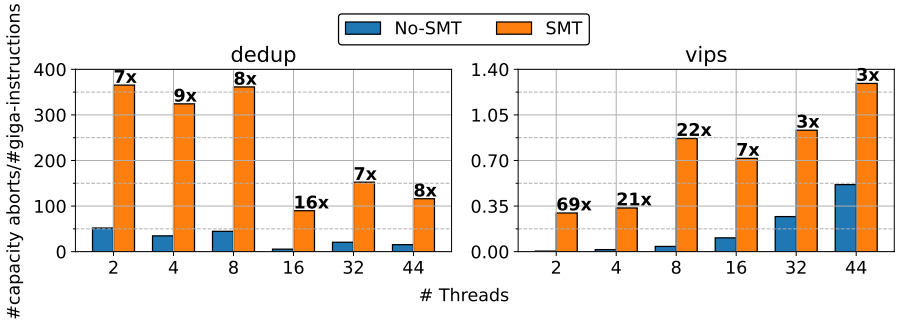


Figure 5.10: Capacity aborts in the PARSEC 3.0 with and without SMT and their ratio.

5.5.2 Transaction-Aware Replacement Algorithm

We present *Transaction-Aware (TA)* replacement algorithm, a novel mechanism that mitigates the negative effects of SMT on capacity abort rate. We build our solution on top of the *Least Recently Used* (LRU) replacement algorithm and call it *TA-LRU*. As shown in Section 5.5.1, SMT increases the number of capacity aborts. The main idea of TA-LRU is to protect the cache lines involved in transactions, and thus mitigate the SMT effect.

The classical LRU algorithm evicts the least recently used cache line when no more space is available in the cache. It does not take into account whether the selected cache line is used in a transaction. By contrast, our TA-LRU algorithm avoids evicting cache lines involved in a transaction unless it is strictly necessary. TA-LRU works as follows: let us assume that the cache contains lines L_1 to L_n, with L_1 being the most recently used and L_n, the least recently used. L_n is also used in a transaction. When a new line has to be brought into the cache,

classical LRU would evict L_n, therefore causing a capacity abort for the running transaction. TA-LRU, on the other hand, will evict the least recently used line that is not used in a transaction, keeping L_n stored in the cache.

Our solution does not need any extra hardware. Typically, in HTM implementations, the cache lines are annotated with two bits to track read/write accesses. TA-LRU simply ORs the two bits. If the result of this operation is 0, i.e., none of the bits is set, it means that the current cache line is not involved in a transaction and can be evicted without any side effects on performance. If any of the bits is set, the algorithm moves to the next candidate in LRU order. Our solution can be integrated with any cache replacement algorithm.

While the TA-LRU selection process for an eviction candidate does not have any overhead compared to standard LRU, the algorithm may not be as effective at choosing the best victim. Some of the least-recently used entries will potentially be marked as transactional and avoided, leading to the eviction of more recent entries. In other words, some of the cache entries that may have been reused in the future with standard LRU, may be evicted with TA-LRU. In the worst case scenario, all lines can be involved in transactions, leading to the eviction of the least recently used transactional line and the subsequent abort of the corresponding transaction. We perform a preliminary overhead assessment in Section 5.5.3 and leave the complete evaluation on real-world applications as future work.

5.5.3 TA-LRU Prototype

We evaluate our proposal in gem5 [16] using the only HTM implementation currently available in the simulator, the Transactional Memory Extension (TME) for the ARM architecture. The incipient state of the TME implementation in gem5 significantly limits the performance of any executed transactional workload and hinders the evaluation. Presented with these limitations, we propose an initial TA-LRU prototype, with the goal of assessing its potential.

We implement and test our prototype in a one-core out-of-order superscalar ARM processor that can execute two threads simultaneously. We use the DerivO3CPU type. The threads share 32 KiB L1 data and instruction caches (8-way), a 1 MiB L2 cache (16-way), and a 2 MiB LLC (16-way). All cache levels are inclusive.

We evaluate our proposal on a synthetic microbenchmarks designed to stress stress transactional capacity and generate overflows aborts. The microbenchmark executes two concurrent threads that access two arrays: one smaller and, respectively, one larger in size than the L1 data cache. One thread starts a transaction, iterates over the smaller array, reading and updating the elements, and commits the transaction. The other thread simply iterates over the larger array, thus polluting the cache and affecting the working-set size limit of the transactional workload.

With TA-LRU we observe a 16x reduction in capacity aborts compared to

classical LRU. More precisely, only 3% of the started transactions abort due to capacity overflow when using TA-LRU, as opposed to 53% for LRU. We further measure the performance in *cycles per instruction (CPI)*. The thread that executes the transactional workload shows 2% performance improvement with TA-LRU. We also look at the potential overhead of our solution, by measuring the slow-down of the non-transactional thread. We find that the performance loss is under 1%. Finally, we analyze the number of cache misses in L1, as indicator for the impact of TA-LRU on caching. We measure an overhead of 6.25% for TA-LRU.

5.6 Related Work

There is extensive literature on synchronization mechanisms scalability. Guiroux et al. [56] compare the performance of 27 lock algorithms on 35 real-world large-scale applications on many-core systems. On the same note, Rico et al. [139] present an extensive scalability study on 4 STM libraries. Similarly, Brown et al. [18] analyze HTM performance on a many-core NUMA system (up to 72 threads), formulating guidelines on efficient use of HTM in a many-socket setup. All these works focus on a single synchronization strategy and make a deep-dive into its scalability performance and issues. By contrast, our work compares all major classes of synchronization mechanisms in order to provide a broader picture on their performance in a many-core context (up to 88 threads).

Only a few studies perform a direct comparison between synchronization methods. Park et al. [128] and Schindewolf et al. [143] experiment with HTM, locks and atomic primitives on a 64-thread setup, both using a synthetic microbenchmark suite that emulates HPC applications. Our work also brings STM into the equation and evaluates scalability on widely-used concurrent data structures. Yoo et al. [176] address the same synchronization mechanisms as us, but on a very low thread count. In addition, we analyze the evolution of HTM events with the increasing number of threads and provide a detailed breakdown on commits and various abort types.

Nakaike et al. [110] and Dice et al. [34] provide an in-depth characterization of HTM for different HTM implementations. While their analyses go into great architectural detail, they do not study the SMT impact on the working-set size limit of transactions. Hasenplaugh et al. [65] and Wang et al. [164] briefly look at capacity aborts in SMT systems, but do not go as far as proposing a solution to mitigate SMT effects on HTM performance. The recent work of Cai et al. [19] aims to shed some light on the way in which transactional structures are tracked in hardware and on the impact of the replacement policy on capacity aborts. They find that flushing or warming the cache maximizes the read-set capacity of a transaction. They do not investigate the impact of SMT in this context.

5.7 Concluding Remarks

This chapter presents an extensive study on the scalability of various synchronization mechanisms. It departs from the state-of-the-art by: including all major forms of synchronization, from typical locking schemes to emerging technologies like HTM; evaluating them in the context of many-core systems; and quantifying the impact of SMT on HTM performance.

For the scalability evaluation, we experiment with two concurrent data-structures, a hash-table and a binary search tree, and workloads with various degrees of contention. We find that, in terms of throughput and latency, STM is lagging behind because of its considerable instrumentation overhead. In contrast, HTM matches lock-free and fine-grained locking performance and scales better as the number of threads increases. Since our HTM-based implementation uses the simplest and most common version of a fallback path, relying on a global lock, these results represent a lower bound for HTM scalability. We note that careful planning of the fallback contents can substantially reduce transactional conflicts and boost the performance.

Going further, we compare the performance of fine-grain locks and HTM on a widely-used benchmark suite consisting of real-world scientific applications, namely PARSEC 3.0. At the same time, we make the case of HTM adoption, showing on the PARSEC benchmarks how easy it is to obtain comparable performance to that of a highly-optimized locking scheme by simply flipping the value of a flag in the glibc library.

Finally, we analyze the impact of SMT on HTM performance. We find that enabling SMT for applications that access large blocks of memory inside their critical sections, considerably affects HTM commit-rate. SMT reduces the available resources per transaction, resulting in repeated capacity overflow aborts. We propose *Transaction-Aware LRU (TA-LRU)*, a novel cache replacement algorithm that aims to mitigate the negative effects of SMT on transactional capacity abort rate. Our prototype reduces aborts in this category by a factor of 16.

Chapter 6

Conclusions and Future Work

This last chapter summarizes the main conclusions of this dissertation and suggests future lines of research based on the contributions presented herein.

6.1 Conclusions

This dissertation explores new approaches to improve the efficiency in the use of shared resources in the memory hierarchy of a multicore processor, from different levels ranging from hardware to application.

First, we have evaluated the memory hierarchy performance for two SPEC suites, CPU2006 and CPU2017, on an Intel Xeon Skylake-SP. We have drawn the following conclusions from this work:

- A significant number of the benchmarks have very low miss ratios in the second and third level caches, even when reducing their available space in the LLC and disabling hardware prefetch. It is remarkable that the resource demand of the CPU2017 memory hierarchy is lower than that of the CPU2006.
- Space utilization in the SLLC is uneven among applications. Increasing the available capacity translates into reductions in cache misses in very different magnitudes depending on the application. Likewise, reductions in miss rates translate into very different performance improvements.
- Hardware prefetching significantly reduces cache misses and increases performance for most benchmarks, even when cache capacity is limited.
- Current methodologies for obtaining simulation points do not guarantee representative workloads regarding the use of the memory hierarchy.

Next, we characterize the relationship between cache occupation, hardware prefetch and memory bandwidth consumption to understand their interactions. This characterization has allowed us to obtain several new interesting findings:

- Some applications hardly change their performance when their allocated space in LLC decreases, but significantly increase their memory bandwidth consumption, negatively affecting system performance.

- When running multiprogrammed workloads, it is common for the traffic generated by memory requests to congest the DRAM channels. This results in high memory latencies, which in turn affects application execution time.

- Main memory traffic is a poor indicator for assessing memory access contention, especially when reaching the congestion point. Memory access latency is a better indicator.

From this characterization work, we have proposed Balancer, a mechanism that imposes limits on the use of LLC space and memory traffic to specific applications. These restrictions improve the performance and/or fairness of multiprogrammed workloads compared to an uncontrolled system. Balancer does not require hardware or operating system modifications. It takes advantage of AMD QoSE, a feature of AMD Rome processors that allows the user to distribute SLLC and main memory bandwidth among different threads.

As we have highlighted in the previous characterizations, data prefetching is a technique that plays a crucial role in modern high-performance processors by hiding long latency memory accesses and improving performance. However, these prefetchers load a large number of useless blocks, ranging from 22.6% to 35.1% for SPEC CPU2017, and reaching 80.2% for programs with more irregular patterns such as those of the GAP suite. This results in an unnecessary increase in the consumption of shared and scarce resources such as cache space and memory bandwidth. In this dissertation, we show that the detection of timely local deltas along with a precise mechanism to compute the local coverage of the detected deltas leads to a high accurate prefetcher that outperforms the state-of the art prefetchers presented in the recent prefetching championships, both for SPEC CPU2017 and GAP workloads. The proposal, Berti, is an L1D prefetcher that orchestrates its requests across the whole cache hierarchy. Due to its high accuracy (almost 90%), Berti neither pollutes the caches nor wastes memory hierarchy bandwidth. Finally, Berti incurs a storage overhead of only 2.55 KB.

Synchronization between threads of the same application is another context in which there may also be a high demand for shared resources in the memory hierarchy as the number of cores per processor increases. Classical solutions, such as fine-grained locks or lock-free algorithms, provide good performance in systems with a small number of cores, in exchange for high complexity. Transactional memory was proposed to achieve similar or better performance than classical synchronization solutions with ease of use. However, the increase in the number of cores per processor demands an analysis of the scalability of all these strategies. This dissertation presents an extensive study on the scalability

of various synchronization mechanisms, from which the following conclusions can be drawn:

- In terms of throughput and latency, software transactional memory is lagging behind the other strategies because of its considerable instrumentation overhead. In contrast, hardware transactional memory matches lock-free and fine-grained locking performance and scales better as the number of threads increases.
- Careful planning of the fallback contents can substantially reduce transactional conflicts and boost the performance.
- HTM adoption is easy in scientific applications and it obtains comparable performance to that of a highly-optimized locking scheme by simply flipping the value of a flag in the glibc library.
- Enabling simultaneous multithreading (SMT) for applications that access large blocks of memory inside their critical sections considerably affects HTM commit-rate.
- *Transaction-Aware LRU (TA-LRU)*, a novel cache replacement algorithm that aims to mitigate the negative effects of SMT on transactional capacity abort rate, can reduce aborts in this category by a factor of 16.

Our proposals improve the existing knowledge and enable more efficient use of the memory hierarchy at different levels of a computer system: application, runtime, and microarchitecture.

6.2 Future work

From this thesis new and promising lines of research are opening up:

- The IBM Power family of processors follows a different organization than the ones analyzed in this work, since it uses a smaller number of cores but with a much higher number of threads per core. This shift implies increased pressure on shared resources that were previously considered private, such as the first levels of cache. Characterizing the behavior of the applications and memory hierarchy on these new processors may allow us to identify potential bottlenecks.
- Balancer uses the mechanisms available in AMD EPYC as actuators: limiting the available space in SLLC and limiting the bandwidth with main memory. Intel processors allow to selectively switch off the data prefetchers while IBM processors allow to modify different parameters of the hardware prefetching, such as its aggressiveness. This opens up the possibility of developing new system resource management mechanisms using these tools.

- Hardware prefetching, despite being a topic that has been extensively studied by the community for years, still has potential to be exploited. TLB prefetching or the use of several types of prefetchers simultaneously from L1D can push the limits of hardware prefetching.

6.3 Conclusiones y Trabajo Futuro

Este último capítulo resume las principales conclusiones de esta tesis y sugiere futuras líneas de investigación basadas en las aportaciones aquí presentadas.

6.3.1 Conclusiones

Esta tesis explora nuevos enfoques para mejorar la eficiencia en el uso de recursos compartidos en la jerarquía de memoria de un procesador multinúcleo, en distintos niveles que van desde el hardware hasta la aplicación.

En primer lugar, hemos evaluado el rendimiento de la jerarquía de memoria para dos suites de SPEC, CPU2006 y CPU2017, en un Intel Xeon Skylake-SP. De este trabajo hemos extraído las siguientes conclusiones:

- Un número significativo de los benchmarks tienen tasas de fallo muy bajas en las caches de segundo y tercer nivel, incluso cuando se reduce su espacio disponible en la LLC y se desactiva la prebúsqueda hardware. Cabe destacar que la demanda de recursos de la jerarquía de memoria de CPU2017 es inferior a la de CPU2006.
- La utilización del espacio en la SLLC es desigual entre aplicaciones. El aumento de la capacidad disponible se traduce en reducciones de los fallos de cache en magnitudes muy diferentes según la aplicación. Del mismo modo, las reducciones en las tasas de fallos se traducen en mejoras de rendimiento muy dispares.
- La prebúsqueda hardware reduce significativamente los fallos en cache y aumenta el rendimiento en la mayoría de los benchmarks, incluso cuando el tamaño de la cache es limitado.
- Las metodologías actuales para obtener puntos de simulación no garantizan cargas de trabajo representativas en cuanto al uso de la jerarquía de memoria.

A continuación, caracterizamos la relación entre la ocupación de la cache, la prebúsqueda hardware y el consumo de ancho de banda de memoria para comprender sus interacciones. Esta caracterización nos ha permitido obtener nuevos hallazgos interesantes:

- Algunas aplicaciones apenas modifican su rendimiento cuando disminuye su espacio asignado en la SLLC, pero aumentan significativamente su consumo de ancho de banda con memoria, afectando negativamente al rendimiento del sistema.
- Cuando se ejecutan cargas de trabajo multiprogramadas, es habitual que el tráfico generado por las peticiones de memoria congestionen los canales de la DRAM. Esto se traduce en elevadas latencias de memoria, lo que a su vez afecta al tiempo de ejecución de las aplicaciones.

- El tráfico con memoria principal es un mal indicador para evaluar la contención en el acceso a la memoria, especialmente cuando se alcanza el punto de congestión. La latencia de acceso a la memoria es un mejor indicador.

A partir de este trabajo de caracterización, hemos propuesto Balancer, un mecanismo que impone límites en el uso del espacio de la LLC y el tráfico de memoria a aplicaciones específicas. Estas restricciones mejoran el rendimiento y/o la equidad de las cargas de trabajo multiprogramadas en comparación con un sistema no controlado. Balancer no requiere modificaciones del hardware ni del sistema operativo. Aprovecha AMD QoSE, una característica de los procesadores AMD Rome que permite al usuario distribuir el ancho de banda de la SLLC y de la memoria principal entre distintos hilos.

Como hemos destacado en las caracterizaciones anteriores, la prebúsqueda hardware de datos es una técnica que desempeña un papel crucial en los procesadores modernos de alto rendimiento al ocultar los accesos a memoria de latencia alta y mejorar el rendimiento. Sin embargo, estos prebuscadores cargan un gran número de bloques inútiles, que oscilan entre el 22.6% y el 35.1% para SPEC CPU2017, y alcanzan el 80.2% para programas con patrones más irregulares como los de la suite GAP. Esto se traduce en un aumento innecesario del consumo de recursos compartidos y escasos como el espacio de cache y el ancho de banda con memoria. En esta tesis, demostramos que la detección puntual de deltas locales junto con un mecanismo preciso para calcular la cobertura local de los deltas detectados conduce a un prebuscador de alta precisión que supera a los prebuscadores hardware de última generación presentados en los recientes campeonatos de prebúsqueda, tanto para cargas de trabajo SPEC CPU2017 como GAP. La propuesta, Berti, es un prebuscador hardware de L1D que orquesta sus peticiones a través de toda la jerarquía de cache. Gracias a su gran precisión (casi el 90%), Berti no contamina las caches ni desperdicia ancho de banda de la jerarquía de memoria. Por último, Berti incurre en una sobrecarga de almacenamiento de sólo 2.55 KB.

La sincronización entre hilos de una misma aplicación es otro contexto en el que también puede haber una gran demanda de recursos compartidos en la jerarquía de memoria a medida que aumenta el número de núcleos por procesador. Las soluciones clásicas, como los *fine-grain locks* o los algoritmos *lock-free*, ofrecen un buen rendimiento en sistemas con un número reducido de núcleos, a cambio de una elevada complejidad. La memoria transaccional se propuso para lograr un rendimiento similar o superior al de las soluciones de sincronización clásicas con facilidad de uso. Sin embargo, el aumento del número de núcleos por procesador exige un análisis de la escalabilidad de todas estas estrategias. Esta tesis presenta un amplio estudio sobre la escalabilidad de diversos mecanismos de sincronización, del que se pueden extraer las siguientes conclusiones:

- En términos de rendimiento y latencia, la memoria transaccional por

software va por detrás de las demás estrategias debido a la considerable sobrecarga de su instrumentación. Por el contrario, la memoria transaccional hardware iguala el rendimiento de los *fine-grain locks* y *lock-free*, y se adapta mejor a medida que aumenta el número de hilos.

- Una planificación cuidadosa del fallback puede reducir sustancialmente los abortos por conflicto y aumentar el rendimiento.
- La adopción de HTM es fácil en aplicaciones científicas y obtiene un rendimiento comparable al de un esquema de fine-grain locks altamente optimizado simplemente cambiando el valor de una bandera en la biblioteca glibc.
- Habilitar el multithreading simultáneo (SMT) para aplicaciones que acceden a grandes bloques de memoria dentro de sus secciones críticas afecta considerablemente a la tasa de retiro de HTM.
- *Transaction-Aware LRU (TA-LRU)*, un novedoso algoritmo de remplazo de cache que pretende mitigar los efectos negativos de SMT en la tasa de abortos por capacidad, puede reducir los abortos en esta categoría en un factor 16.

Nuestras propuestas mejoran los conocimientos existentes y permiten un uso más eficiente de la jerarquía de memoria en los distintos niveles de un sistema informático: aplicación, *runtime* y microarquitectura.

6.4 Future Work

A partir de esta tesis se abren nuevas y prometedoras líneas de investigación:

- La familia de procesadores IBM Power sigue una organización diferente a la de los analizados en este trabajo, ya que utiliza un número menor de núcleos pero con un número mucho mayor de hilos por núcleo. Este cambio implica una mayor presión sobre recursos compartidos que antes se consideraban privados, como los primeros niveles de cache. Caracterizar el comportamiento de las aplicaciones y la jerarquía de memoria en estos nuevos procesadores puede permitirnos identificar posibles cuellos de botella.
- Balancer utiliza como actuadores los mecanismos disponibles en AMD EPYC: limitar el espacio disponible en SLLC y limitar el ancho de banda con la memoria principal. Los procesadores Intel permiten desactivar selectivamente los prebuscadores hardware de datos, mientras que los procesadores IBM permiten modificar diferentes parámetros de la prebúsqueda hardware, como su agresividad. Esto abre la posibilidad de desarrollar nuevos mecanismos de gestión de recursos del sistema utilizando estas herramientas.

- La prebúsqueda hardware, a pesar de ser un tema ampliamente estudiado por la comunidad desde hace años, aún tiene potencial por explotar. La prebúsqueda de TLB o el uso simultáneo de varios tipos de prebuscadores en L1D pueden llevar al límite a la prebúsqueda hardware.

Bibliography

[1] The 1st Data Prefetching Championship (DPC-1). https://jilp.org/dpc, 2009. Accessed on: 17 May 2020.

[2] The 2nd Cache Replacement Championship. http://crc2.ece.tamu.edu, 2017. Accessed on: 03 April 2019.

[3] The 2nd Data Prefetching Championship (DPC-2). https://comparch-conf.gatech.edu/dpc2, 2015. Accessed on: 17 May 2020.

[4] The 3rd Data Prefetching Championship (DPC-3). https://dpc3.compas.cs.stonybrook.edu/, 2019. Accessed on: 17 May 2020.

[5] 7-CPU. Sunnycove microarchitecture latency, 2018.

[6] Jorge Albericio, Rubén Gran, Pablo Ibáñez, Víctor Viñals, and Jose María Llabería. Abs: A low-cost adaptive controller for prefetching in a banked shared last-level cache. *taco*, 8(4):19:1–19:20, 2012.

[7] Advanced Micro Devices (AMD). Software optimization guide for amd epyc™ 7003 processors. whitepaper, 2020.

[8] Neelu Shivprakash Kalani and. Instruction criticality based energy-efficient hardware data prefetching. *IEEE Comput. Archit. Lett.*, 20(2):146–149, 2021.

[9] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Bingo spatial data prefetcher. In *25th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 399–411, 2019.

[10] Scott Beamer, Krste Asanović, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[11] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A customizable hardware

prefetching framework using online reinforcement learning. In *54th Int'l Symp. on Microarchitecture (MICRO)*, pages 1121–1137, 2021.

[12] Rahul Bera, Anant V. Nori, , Onur Mutlu, and Sreenivas Subramoney. Dspatch: Dual spatial pattern prefetcher. In *52th Int'l Symp. on Microarchitecture (MICRO)*, pages 531–544, 2019.

[13] Eshan Bhatia, Gino Chacon, Seth H. Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. Perceptron-based prefetch filtering. In *46th Int'l Symp. on Computer Architecture (ISCA)*, pages 1–13, 2019.

[14] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81. ACM, 2008.

[16] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, 2011.

[17] L. F. Bonnichsen, C. W. Probst, and S. Karlsson. Hardware transactional memory optimization guidelines, applied to ordered maps. In *9th Int'l Symp. on Parallel and Distributed Processing with Applications (ISPA)*, pages 124–131. IEEE, 2015.

[18] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. Investigating the performance of hardware transactions on a multi-socket machine. In *28th Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 121–132. ACM, 2016.

[19] Zixian Cai, Stephen M. lackburn, and Michael D. Bond. Understanding and utilizing hardware transactional memory capacity. In *Int'l Symp. on Memory Management (ISMM)*. ACM, 2021.

[20] I. Calciu, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*. ACM, 2014.

[21] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[22] ChampSim. ChampSim simulator. http://github.com/ChampSim/ChampSim, 2020.

[23] Shuang Chen et al. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proc. of the Twenty-Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[24] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. Leaking control flow information via the hardware prefetcher. *CoRR*, abs/2109.00474, 2021.

[25] CloudSuite. Cloudsuite traces for champsim. https://www.dropbox.com/sh/pgmnzfr3hurlutq/AACciuebRwSAOzhJkmj5SEXBa/CRC2_trace?dl, 2017.

[26] Henry Cook et al. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *40th Int'l Symp. on Computer Architecture (ISCA)*, page 308–319, 2013.

[27] Jeanine Cook, Jonathan Cook, and Waleed Alkohlani. A statistical performance model of the opteron processor. *SIGMETRICS Perform. Eval. Rev.*, pages 75–80, 2011.

[28] Kanter D. Skylake-sp scales sever systems. Microprocessor Report, 2017.

[29] DDR. Ddr standards.

[30] Arnaldo Carvalho. De Melo. The new linux 'perf' tools., 2010.

[31] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[32] Advanced Micro Devices. Amd64 technology platform quality of service extensions. Pub. 56375, rev 1.01, 2018.

[33] Advanced Micro Devices. Processor programing reference (prr) for amd family 17h model 20h, revision a1 processors. Rev 3.07, 2020.

[34] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. *SIGARCH Comput. Archit. News*, 37(1):157–168, 2009.

[35] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *15th Int'l Conf. on Distributed Computing (DISC)*, pages 194–208. Springer, 2006.

[36] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *32th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 3–14. ACM, 2014.

[37] Jack Doweck. Inside intel core microarchitecture and smart memory access. In *Intel whitepaper*, 2006.

[38] Jack Doweck. Inside intel core microarchitecture and smart memory access. In *Intel whitepaper*, pages 1–13, 2006.

[39] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54(4):70–77, 2011.

[40] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *SIGPLAN Not.*, 44(6):155–165, 2009.

[41] Eiman Ebrahimi et al. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *15th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, page 335–346, 2010.

[42] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *42nd Int'l Symp. on Microarchitecture (MICRO)*, pages 316–326, 2009.

[43] EEMBC. Embedded microprocessor benchmarks. https://www.eembc.org/about/index.php, 1997. Accessed on: 18 January 2019.

[44] N. El-Sayed et al. Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *24th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 104–117, 2018.

[45] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Sørensen, Peter Wägemann, and Simon Wegener. Taclebench: a benchmark collection to support worst-case execution time research. In *16th Int'l Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 55, pages 2:1–2:10, 2016.

[46] P. Felber, C. Fetzer, T. Riegel, and P. Marlier. Time-based software transactional memory. *IEEE Trans. Parallel Distrib. Syst.*, 21(12):1793–1807, 2010.

[47] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *17th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, page 37–48, 2012.

[48] Ricardo Filipe, Shady Issa, Paolo Romano, and João Barreto. Stretching the capacity of hardware transactional memory in IBM POWER architectures. In *24th Principles and Practice of Parallel Programming (PPoPP)*, page 107–119. Association for Computing Machinery, 2019.

[49] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers, 2020. Available at https://www.agner.org/optimize/microarchitecture.pdf.

[50] Fujitsu. A64FX® microarchitecture manual. Ver. 1.6, September 2021.

[51] Liran Funaro et al. Ginseng: Market-driven llc allocation. In *USENIX Annual Technical Conference (USENIX ATC)*, page 295–308, 2016.

[52] GAP. GAP traces for champsim. https://utexas.app.box.com/s/2k54kp8zvrqdfaa8cdhfquvcxwh7yn85/folder/132804598561, 2021.

[53] Adrian Garcia-Garcia, Juan Carlos Saez, et al. Lfoc: A lightweight fairness-oriented cache clustering policy for commodity multicores. In *48th Int'l Conf. on Parallel Processing (ICPP)*, 2019.

[54] Brian Grayson, Jeff Rupley, Gerald D. Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. Evolution of the samsung exynos cpu microarchitecture. In *47th Int'l Symp. on Computer Architecture (ISCA)*, pages 40–51, 2020.

[55] Hugo Guiroux. LiTL source code and data sets. https://github.com/multicore-locks, 2016.

[56] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. Multicore locks: The case is not closed yet. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 649–662. USENIX, 2016.

[57] Linley Gwennap. Skylake-sp scales server systems. *Microprocessor Report*, 2017.

[58] Linley Gwennap. Amd rome ruins intel hegemony. *Microprocessor Report*, 2019.

[59] Linley Gwennap. AMD Rome ruins Intel hegemony. *Microprocessor Report*, 2019.

[60] Linley Gwennap. Ibm power10 triples efficiency. *Microprocessor Report*, 2020.

[61] Linley Gwennap. IBM Power10 triples efficiency. *Microprocessor Report*, 2020.

[62] Linley Gwennap. Neoverse advances simd for hpc. *Microprocessor Report*, 2020.

[63] Linley Gwennap. Amd milan extends server lead. *Microprocessor Report*, 2021.

[64] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *15th Int'l Conf. on Distributed Computing (DISC)*, pages 300–314. Springer, 2001.

[65] William Hasenplaugh, Andrew Nguyen, and Nir Shavit. Quantifying the capacity limitations of hardware transactional memory. In *7th Int'l Workshop on the Theory of Transactional Memory (WTTM)*, 2015.

[66] Wim Heirman, Kristof Du Bois, Yves Vandriessche, Stijn Eyerman, and Ibrahim Hur. Near-side prefetch throttling: Adaptive prefetching for high-performance many-core processors. In *27th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 28:1–28:11, 2018.

[67] Steve Heller, , and Maurice Herlihy. A lazy concurrent list-based set algorithm. In *10th Int'l Conf. on Principles of Distributed Systems (OPODIS)*, pages 3–16. Springer, 2006.

[68] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2017.

[69] A. Herdrich et al. Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family. In *22th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 657–668, 2016.

[70] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Int'l Symp. on Computer Architecture (ISCA)*, page 289–300, 1993.

[71] Derek R. Hower et al. Pabst: Proportionally allocated bandwidth at the source and target. In *23th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 505–516, 2017.

[72] Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *24th Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 161–171. ACM, 2012.

[73] hpa. Lock elision in the GNU C library. https://lwn.net/Articles/535519/, 2021.

[74] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. Tcp: Tag correlating prefetchers. In *9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 317–326, 2003.

[75] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G. Abraham. Effective stream-based and execution-based data prefetching. In *18th Int'l Conf. on Supercomputing (ICS)*, pages 1–11, 2004.

[76] IBM. POWER9 processor user's manual (version 2.0), 2018.

[77] Intel. Intel® 64 and IA-32 architectures software developer's manual, combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d and 4., 2019.

[78] Intel. Intel® resource director technology (intel® RDT). https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html, 2019. Accessed on: 18 January 2019.

[79] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. "", 2020.

[80] Intel. *Intel Architecture Instruction Set Extensions and Future Features*. "", 2022.

[81] Intel. *Intel Intrinsics Guide*. "Last access 19 Sep. 2022", 2022.

[82] Engin Ipek et al. Self-optimizing memory controllers: A reinforcement learning approach. In *35th Int'l Symp. on Computer Architecture (ISCA)*, pages 39–50, 2008.

[83] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM System Z. In *45th Int'l Symp. on Microarchitecture (MICRO)*, pages 25–36. IEEE, 2012.

[84] Akanksha Jain. *Exploiting Long-Term behavior for Improved Memory System Performance*. PhD thesis, University of Texas at Austin, 2016.

[85] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *46th Int'l Symp. on Microarchitecture (MICRO)*, pages 247–259, 2013.

[86] Akanksha Jain and Calvin Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. In *43thInt'l Symp. on Computer Architecture (ISCA)*, pages 78–89, 2016.

[87] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation. http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf, 2010. Accessed on: 12 May 2022.

[88] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 197–206, 2001.

[89] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *24th Int'l Symp. on Computer Architecture (ISCA)*, pages 252–263, 1997.

[90] Chris Karamatas. AMD EPYC 7003 series microarchitecture overview. Pub. 57075, rev 3.0, March 2022.

[91] Anre Kashyap. High performance computing: Tuning guide for AMD EPYC™ 7002 series processors. Pub. 56827, rev 1.0, January 2020.

[92] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A. L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chisht. Path confidence based lookahead prefetching. In *49th Int'l Symp. on Microarchitecture (MICRO)*, pages 60:1–60:12, 2016.

[93] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A. L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. Path confidence based lookahead prefetching. In *49th Int'l Symp. on Microarchitecture (MICRO)*, pages 60:1–60:12, 2016.

[94] Jinchun Kim, Elvira Teran, Paul V. Gratz, Daniel A. Jiménez, Seth H. Pugsley, and Chris Wilkerson. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. In *22nd Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 737–749, 2017.

[95] Seongbeom Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *13th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, page 111–122, 2004.

[96] Y. Kim et al. Application performance prediction and optimization under cache allocation technology. In *DATE*, pages 1285–1288, 2019.

[97] Wendy Korn and Moon S. Chang. Spec cpu2006 sensitivity to memory page sizes. *SIGARCH Comput. Archit. News*, 35(1):97–101, 2007.

[98] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM J. Res. Dev.*, 59(1):8:1–8:14, 2015.

[99] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *2011 Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 694–701, 2011.

[100] Ankur Limaye and Tosiron. Adegbija. A workload characterization of the spec cpu2017 benchmark suite. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158, 2018.

[101] David Lo et al. Improving resource efficiency at scale with heracles. *ACM Trans. Comput. Syst.*, 34(2), 2016.

[102] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995.

[103] Collin McCurdy and Charles Fischer. Using pin as a memory reference generator for multiprocessor simulation. *SIGARCH Comput. Archit. News*, 33(5):39–44, 2006.

[104] Pierre Michaud. Best-offset hardware prefetching. In *22th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 469–480, 2016.

[105] Micron. Micron dram power calculator. https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf, 2015.

[106] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114—117, 1965.

[107] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *12th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 254–265. IEEE, 2006.

[108] Onur Mutlu et al. Stall-time fair memory access scheduling for chip multiprocessors. *40th Int'l Symp. on Microarchitecture (MICRO)*, pages 146–160, 2007.

[109] Samuel Naffziger et al. Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families : Industrial product. In *48th Int'l Symp. on Computer Architecture (ISCA)*, pages 57–70, 2021.

[110] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *42th Int'l Symp. on Computer Architecture (ISCA)*, pages 144–157. ACM, 2015.

[111] A. Natarajan and N. Mittal. False conflict reduction in the Swiss Transactional Memory (SwissTM) system. In *IEEE Int'l Symp. on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[112] Agustin Navarro-Torres. Perf++, 2017.

[113] Agustín Navarro-Torres, Jesús Alastruey-Benedé, Pablo Ibáñez-Marín, and Maria Carpen-Amarie. Synchronization strategies on many-core smt systems. In *33th Int'l Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 54–63. IEEE, 2021.

[114] Agustín Navarro-Torres, Jesús Alastruey-Benedé, Pablo Ibáñez-Marín, and Víctor Viñals-Yúfera. Memory hierarchy characterization of spec cpu2006 and spec cpu2017 on the intel xeon skylake-sp. *PLOS ONE*, pages 1–24, 2019.

[115] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñals-Yúfera, and Alberto Ros. Berti: an accurate local-delta data prefetcher. In *55th Int'l Symp. on Microarchitecture (MICRO)*, pages 975–991. IEEE, 2022.

[116] Kyle J. Nesbit et al. Fair queuing memory systems. In *39th Int'l Symp. on Microarchitecture (MICRO)*, pages 208–222, 2006.

[117] Konstantinos Nikas et al. Dicer: Diligent cache partitioning for efficient workload consolidation. In *48th Int'l Conf. on Parallel Processing (ICPP)*, 2019.

[118] SPEC Organization. Spec cpu 2006. https://www.spec.org/cpu2006/, 2006. Accessed on: 25 January 2019.

[119] SPEC Organization. Spec cpu 2017. https://www.spec.org/cpu2017/, 2017. Accessed on: 18 January 2019.

[120] SPEC Organization. The SPEC organization. https://www.spec.org/spec/, 2019. Accessed on: 18 January 2019.

[121] Celal Öztürk, Ibrahim Burak Karsli, and Resit Sendag. An analysis of address and branch patterns with patternfinder. In *iiswc*, pages 232–242, 2014.

[122] Samuel Pakalapati and Biswabandan Panda. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *47th Int'l Symp. on Computer Architecture (ISCA)*, pages 118–131, 2020.

[123] Biswabandan Panda. SPAC: A synergistic prefetcher aggressiveness controller for multi-core systems. *ie3tc*, 65(12):3740–3753, 2016.

[124] Biswabandan Panda and Shankar Balachandran. CAFFEINE: A utility-driven prefetcher aggressiveness engine for multicores. *taco*, 12(3):30:1–30:25, 2015.

[125] Biswabandan Panda and Shankar Balachandran. Expert prefetch prediction: An expert predicting the usefulness of hardware prefetchers. *cal*, 15(1):13–16, 2016.

[126] Reena Panda, Shuang Song, Joseph Dean, and Lizy K John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In *24th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 271–282, 2018.

[127] Victor Pankratius and Ali-Reza Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *23th Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 43–52. ACM, 2011.

[128] J. Park and W. Baek. Quantifying the performance and energy-efficiency impact of hardware transactional memory on scientific applications on large-scale NUMA systems. In *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, pages 804–813. IEEE, 2018.

[129] Jinsu Park et al. Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers. In *27th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2018.

[130] Jinsu Park et al. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *14th European Conference on Computer Systems (EuroSys)*, 2019.

[131] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems*, pages 318–319, 2003.

[132] L. Pons et al. Phase-aware cache partitioning to target both turnaround time and system performance. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2556–2568, 2020.

[133] Ricardo Quislant, Eladio Gutierrez, Emilio L. Zapata, and Oscar Plata. Enhancing scalability in best-effort hardware transactional memory systems. *J. Parallel Distrib. Comput.*, 104(C):73–87, 2017.

[134] Moinuddin K. Qureshi et al. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *39th Int'l Symp. on Microarchitecture (MICRO)*, pages 423–432, 2006.

[135] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *34th Int'l Symp. on Microarchitecture (MICRO)*, pages 294–305. IEEE, 2001.

[136] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 5–17. ACM, 2002.

[137] Luis M. Ramos, José Luis Briz, Pablo E. Ibáñez, and Víctor Viñals. Multi-level adaptive prefetching based on performance gradient tracking. *jilp*, 13:1–14, 2011.

[138] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *32nd Int'l Symp. on Microarchitecture (MICRO)*, pages 16–27, 1999.

[139] Timoteo M. Rico, Mauricio L. Pilla, Andre R. Du Bois, and Rodrigo M. Duarte. Energy consumption and scalability evaluation for software transactional memory on a real computing environment. In *27th Int'l Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 7–12. IEEE, 2015.

[140] Alberto Ros. Berti: A per-page best-request-time delta prefetcher. In *The 3rd Data Prefetching Championship*, 2019.

[141] Alberto Ros and Alexandra Jimborean. A cost-effective entangling prefetcher for instructions. In *47th Int'l Symp. on Computer Architecture (ISCA)*, pages 99–111, 2021.

[142] Karl Rupp. Data repository for my blog series on microprocessor trend data. https://github.com/karlrupp/microprocessor-trend-data, 2020. Accessed on: 16 February 2022.

[143] Martin Schindewolf, Barna Bihari, John Gyllenhaal, Martin Schulz, Amy Wang, and Wolfgang Karl. What scientific applications can benefit from hardware transactional memory? In *Conf. on Supercomputing (SC)*, pages 1–11. IEEE, 2012.

[144] V. Selfa et al. Application clustering policies to address system fairness with intel's cache allocation technology. In *26th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 194–205, 2017.

[145] Mehran Shakerinava, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Multi-lookahead offset prefetching. In *The 3rd Data Prefetching Championship*, 2019.

[146] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 45–57, 2002.

[147] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *48th Int'l Symp. on Microarchitecture (MICRO)*, pages 141–152, 2015.

[148] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. A hierarchical neural model of data prefetching. In *26th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 861–873, 2021.

[149] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *36th Int'l Symp. on Computer Architecture (ISCA)*, pages 69–80, 2009.

[150] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *33th Int'l Symp. on Computer Architecture (ISCA)*, pages 252–263, 2006.

[151] G. Southern and J. Renau. Analysis of PARSEC workload scalability. In *16thInt'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 133–142. IEEE, 2016.

[152] SPEC. SPEC CPU 2017 traces for champsim. https://hpca23.cse.tamu.edu/champsim-traces/speccpu/index.html, 2019.

[153] SPEC. Spec cpu 2006. https://www.spec.org/cpu2006/, Last access: 05-30-2021.

[154] SPEC. Spec cpu 2017. https://www.spec.org/cpu2017/, Last access: 05-30-2021.

[155] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *13th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 63–74, 2007.

[156] Standard Performance Evaluation Corporation. SPEC CPU2017, 2017.

[157] William J. Starke, Brian W. Thompto, Jeff A. Stuecheli, and José E. Moreira. Ibm's power10 processor. *IEEE Micro*, 41(2):7–14, 2021.

[158] G. Sun et al. Combining prefetch control and cache partitioning to improve multicore performance. In *33th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, pages 953–962, 2019.

[159] Karthik Sundaram and Arun Radhakrishnan. Address re-ordering mechanism for efficient pre-fetch training in an out-of-order processor. U.S. Patent 9542323B2, 2014.

[160] Priyanka Tembey et al. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In *5th IEEE Int'l System-on-Chip Conference (IEEE SOCC)*, page 1–14, 2014.

[161] Dennis Antony Varkey, Biswabandan Panda, and Madhu Mutyam. RCTP: Region correlated temporal prefetcher. In *35th Int'l Conf. on Computer Design (ICCD)*, pages 73–80, 2017.

[162] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *21th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, page 127–136. ACM, 2012.

[163] Xiaodong Wang et al. Swap: Effective fine-grain management of shared last-level caches with minimum hardware support. In *23th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 121–132, 2017.

[164] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. Using restricted transactional memory to build a scalable in-memory database. In *9th European Conference on Computer Systems (EuroSys)*, pages 1–15. ACM, 2014.

[165] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. Simflex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[166] WikiChip. Sunnycove microarchitecture, 2018.

[167] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 24–36, 1995.

[168] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Temporal prefetching without the off-chip metadata. In *52th Int'l Symp. on Microarchitecture (MICRO)*, pages 996–1008, 2019.

[169] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Efficient metadata management for irregular data prefetching. In *46th Int'l Symp. on Computer Architecture (ISCA)*, pages 449–461, 2019.

[170] Yaocheng Xiang et al. Dcaps: Dynamic cache allocation with partial sharing. In *13th European Conference on Computer Systems (EuroSys)*, 2018.

[171] Yaocheng Xiang et al. Emba: Efficient memory bandwidth allocation to improve performance on intel commodity processor. In *48th Int'l Conf. on Parallel Processing (ICPP)*, 2019.

[172] Jun Xiao et al. Cppf: A prefetch aware llc partitioning approach. In *48th Int'l Conf. on Parallel Processing (ICPP)*, 2019.

[173] M. Xu et al. vcat: Dynamic cache management using cat virtualization. In *23th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–222, 2017.

[174] M. Xu et al. Dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *13th European Conference on Computer Systems (EuroSys)*, pages 1–13, 2018.

[175] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *13th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 261–272. IEEE, 2007.

[176] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *Conf. on Supercomputing (SC)*, pages 1–11. ACM, 2013.

[177] Haishan Zhu et al. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. In *21th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, page 33–47, 2016.

[178] Huaiyu Zhu, Yong Chen, and Xian-He Sun. Timing local streams: improving timeliness in data prefetching. In *24th Int'l Conf. on Supercomputing (ICS)*, pages 169–178, 2010.