Runtime-Assisted Optimizations in the On-Chip Memory Hierarchy

---

Vladimir Dimić

Barcelona, 2020

Advisors: **Miquel Moretó Planas,**
Marc Casas Guix

A thesis submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy

in the Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

to my family

Abstract

Following Moore's Law, the number of transistors on chip has been increasing exponentially, which has led to the increasing complexity of modern processors. As a result, the efficient programming of such systems has become more difficult. Many programming models have been developed to answer this issue. Of particular interest are task-based programming models that employ simple annotations to define parallel work in an application. The information available at the level of the runtime systems associated with these programming models offers great potential for improving hardware design. Moreover, due to technological limitations, Moore's Law is predicted to eventually come to an end, so novel paradigms are necessary to maintain the current performance improvement trends.

The main goal of this thesis is to exploit the knowledge about a parallel application available at the runtime system level to improve the design of the on-chip memory hierarchy. The coupling of the runtime system and the microprocessor enables a better hardware design without hurting the programmability.

The first contribution is a set of insertion policies for shared last-level caches that exploit information about tasks and task data dependencies. The intuition behind this proposal revolves around the observation that parallel threads exhibit different memory access patterns. Even within the same thread, accesses to different variables often follow distinct patterns. The proposed policies insert cache lines into different logical positions depending on the dependency type and task type to which the corresponding memory request belongs.

The second proposal optimizes the execution of reductions, defined as a programming pattern that combines input data to form the resulting reduction variable. This is achieved with a runtime-assisted technique for performing reductions in the processor's cache hierarchy. The proposal's goal is to be a universally applicable solution regardless of the reduction variable type, size and access pattern. On the software level, the programming model is extended to let a programmer specify the reduction variables for tasks, as well as the desired cache level where a certain reduction will be performed. The source-to-source compiler and the runtime system are extended to translate and forward this information to the underlying hardware. On the hardware

level, private and shared caches are equipped with functional units and the accompanying logic to perform reductions at the cache level. This design avoids unnecessary data movements to the core and back as the data is operated at the place where it resides.

The third contribution is a runtime-assisted prioritization scheme for memory requests inside the on-chip memory hierarchy. The proposal is based on the notion of a critical path in the context of parallel codes and a known fact that accelerating critical tasks reduces the execution time of the whole application. In the context of this work, task criticality is observed at a level of a task type as it enables simple annotation by the programmer. The acceleration of critical tasks is achieved by the prioritization of corresponding memory requests in the microprocessor.

Resumen

Siguiendo la ley de Moore, el número de transistores en los chips ha crecido exponencialmente, lo que ha comportado una mayor complejidad en los procesadores modernos y, como resultado, de la dificultad de la programación eficiente de estos sistemas. Se han desarrollado muchos modelos de programación para resolver este problema; un ejemplo particular son los modelos de programación basados en tareas, que emplean anotaciones sencillas para definir los trabajos paralelos de una aplicación. La información de que disponen los sistemas en tiempo de ejecución (runtime systems) asociada con estos modelos de programación ofrece un enorme potencial para la mejora del diseño del hardware. Por otro lado, las limitaciones tecnológicas hacen que la ley de Moore pueda dejar de cumplirse próximamente, por lo que se necesitan paradigmas nuevos para mantener las tendencias actuales de mejora de rendimiento.

El objetivo principal de esta tesis es aprovechar el conocimiento de las aplicaciones paralelas de que dispone el runtime system para mejorar el diseño de la jerarquía de memoria del chip. El acoplamiento del runtime system junto con el microprocesador permite realizar mejores diseños hardware sin afectar negativamente en la programabilidad de dichos sistemas.

La primera contribución de esta tesis consiste en un conjunto de políticas de inserción para las memorias caché compartidas de último nivel que aprovecha la información de las tareas y las dependencias de datos entre estas. La intuición tras esta propuesta se basa en la observación de que los hilos de ejecución paralelos muestran distintos patrones de acceso a memoria e, incluso dentro del mismo hilo, los accesos a diferentes variables a menudo siguen patrones distintos. Las políticas que se proponen insertan líneas de caché en posiciones lógicas diferentes en función de los tipos de dependencia y tarea a los que corresponde la petición de memoria.

La segunda propuesta optimiza la ejecución de las reducciones, que se definen como un patrón de programación que combina datos de entrada para conseguir la variable de reducción como resultado. Esto se consigue mediante una técnica asistida por el runtime system para la realización de reducciones en la jerarquía de la caché del procesador, con el objetivo de ser una solución aplicable de forma universal sin depender del tipo de la variable de la reducción, su tamaño o el patrón de acceso. A nivel de software, el modelo de programación se extiende para

que el programador especifique las variables de reducción de las tareas, así como el nivel de caché escogido para que se realice una determinada reducción. El compilador fuente a fuente (compilador source-to-source) y el runtime ssytem se modifican para que traduzcan y pasen esta información al hardware subyacente, evitando así movimientos de datos innecesarios hacia y desde el núcleo del procesador, al realizarse la operación donde se encuentran los datos de la misma.

La tercera contribución proporciona un esquema de priorización asistido por el runtime system para peticiones de memoria dentro de la jerarquía de memoria del chip. La propuesta se basa en la noción de camino crítico en el contexto de los códigos paralelos y en el hecho conocido de que acelerar tareas críticas reduce el tiempo de ejecución de la aplicación completa. En el contexto de este trabajo, la criticidad de las tareas se considera a nivel del tipo de tarea ya que permite que el programador las indique mediante anotaciones sencillas. La aceleración de las tareas críticas se consigue priorizando las correspondientes peticiones de memoria en el microprocesador.

Resum

Seguint la llei de Moore, el nombre de transistors que contenen els xips ha patit un creixement exponencial, fet que ha provocat un augment de la complexitat dels processadors moderns i, per tant, de la dificultat de la programació eficient d'aquests sistemes. Per intentar solucionar-ho, s'han desenvolupat diversos models de programació; un exemple particular en són els models basats en tasques, que fan servir anotacions senzilles per definir treballs paral·lels dins d'una aplicació. La informació que hi ha al nivell dels sistemes en temps d'execució (runtime systems) associada amb aquests models de programació ofereix un gran potencial a l'hora de millorar el disseny del maquinari. D'altra banda, les limitacions tecnològiques fan que la llei de Moore pugui deixar de complir-se properament, per la qual cosa calen nous paradigmes per mantenir les tendències actuals en la millora de rendiment.

L'objectiu principal d'aquesta tesi és aprofitar els coneixements que el runtime system té d'una aplicació paral·lela per millorar el disseny de la jerarquia de memòria dins el xip. L'acoblament del runtime system i el microprocessador permet millorar el disseny del maquinari sense malmetre la programabilitat d'aquests sistemes.

La primera contribució d'aquesta tesi consisteix en un conjunt de polítiques d'inserció a les memòries cau (cache memories) compartides d'últim nivell que aprofita informació sobre tasques i les dependències de dades entre aquestes. La intuïció que hi ha al darrere d'aquesta proposta es basa en el fet que els fils d'execució paral·lels mostren diferents patrons d'accés a la memòria; fins i tot dins el mateix fil, els accessos a variables diferents sovint segueixen patrons diferents. Les polítiques que s'hi proposen insereixen línies de la memòria cau a diferents ubicacions lògiques en funció dels tipus de dependència i de tasca als quals correspon la petició de memòria.

La segona proposta optimitza l'execució de les reduccions, que es defineixen com un patró de programació que combina dades d'entrada per aconseguir la variable de reducció com a resultat. Això s'aconsegueix mitjançant una tècnica assistida pel runtime system per dur a terme reduccions en la jerarquia de la memòria cau del processador, amb l'objectiu que la proposta sigui aplicable de manera universal, sense dependre del tipus de la variable a la qual

es realitza la reducció, la seva mida o el patró d'accés. A nivell de programari, es realitza una extensió del model de programació per facilitar que el programador especifiqui les variables de les reduccions que usaran les tasques, així com el nivell de memòria cau desitjat on s'hauria de realitzar una certa reducció. El compilador font a font (compilador source-to-source) i el runtime system s'amplien per traduir i passar aquesta informació al maquinari subjacent. A nivell de maquinari, les memòries cau privades i compartides s'equipen amb unitats funcionals i la lògica corresponent per poder dur a terme les reduccions a la pròpia memòria cau, evitant així moviments de dades innecessaris entre el nucli del processador i la jerarquia de memòria.

La tercera contribució proporciona un esquema de priorització assistit pel runtime system per peticions de memòria dins de la jerarquia de memòria del xip. La proposta es basa en la noció de camí crític en el context dels codis paral·lels i en el fet conegut que l'acceleració de les tasques que formen part del camí crític redueix el temps d'execució de l'aplicació sencera. En el context d'aquest treball, la criticitat de les tasques s'observa al nivell del seu tipus ja que permet que el programador les indiqui mitjançant anotacions senzilles. L'acceleració de les tasques crítiques s'aconsegueix prioritzant les corresponents peticions de memòria dins el microprocessador.

Acknowledgments

The road to PhD is not easy and would not be possible without help and support of many people.

First of all, I would like to thank to my advisors Miquel Moretó and Marc Casas for their patience and guidance. They were always there to help and encourage me. I learned a lot from them about many aspects of academic research.

I would also like to thank my colleagues from the RoMol team. The special mention goes to Adrian, Calvin, Cesar, Constan, Dimitrios, Helena, Isaac, Luc and Xubin, who were not only great teammates, but also good friends.

I am grateful to the pre-defense committee members, Vicenç Beltran, Petar Radojković and Sara Royela, and external reviewers, Nikos Nikoleris and Miquel Pericas, for providing valuable comments and suggestions, which helped me improve this thesis.

I would like to thank Francesc Martinez for the support with the TaskSim infrastructure, Vicenç Beltran and Sergi Mateo for sharing their valuable experience with reductions in the context of programming models, Lluc Alvarez for numerous discussions about computer architecture, and Isaac Sánchez Barrera for his help on translating the abstract into Catalan and Spanish.

Захваљујем се својим родитељима, Ивану и Алли, и сестри Наташи на непрестаној подршци током свих ових година, без које остварење овог циља не би било могуће. Посебну захвалност дугујем и Бранкици што је била уз мене у најтежем периоду доктората и што га је испунила лепим заједничким тренуцима.

This thesis has been supported by the Agency for Management of University and Research Grants (AGAUR) of the Government of Catalonia under Ajuts per a la contractació de personal investigador novell fellowship number 2017 FI_B 00855, the RoMoL ERC Advanced Grant GA 321253, by the European HiPEAC Network of Excellence, by the Spanish Ministry of Economy and Competitiveness (contract TIN2015-65316-P), and by Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328).

Contents

Chapter 1

Introduction

Memories have been an integral part of computers ever since the appearance of the first concept for a programmable computing machine in 1837 [16]. Since they store both instructions and data, fast memory accesses are a must in order to achieve the good performance of applications.

Alas, due to differences in technologies used to manufacture processors and memories, the gap between speeds of these two resources has been increasing exponentially. This phenomenon is illustrated in Figure 1.1 by comparing processor and memory performance expressed through SPECint performance and access latency, respectively. The consequence of this behavior is that the relative cost of accessing memory has been increasing exponentially. To bridge this performance gap, the concept of caching the data in the processor has come to light. On-chip cache memories offer shorter access latency and higher bandwidth compared to the main memory. On the other hand, they are more costly in terms of area and power per byte of storage and require a careful design of algorithms that manage their contents. However, it is predicted

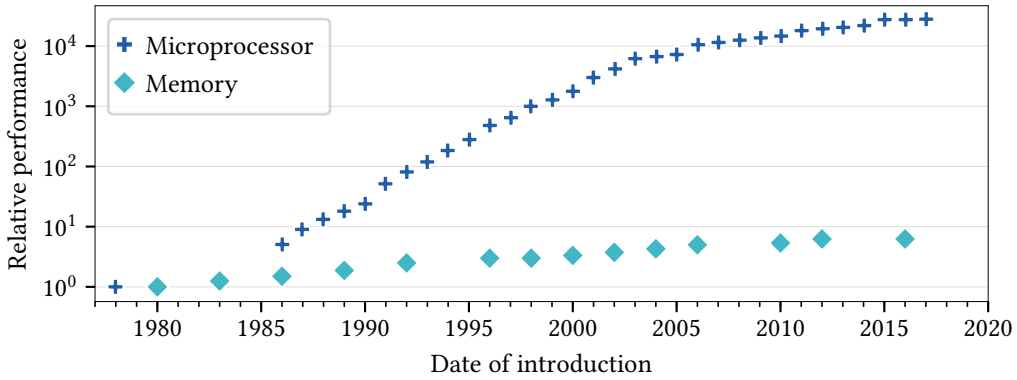


Figure 1.1: Evolution of relative processor and memory performance.

Data collected and plotted by Hennessy and Patterson [74]. Each dot corresponds to a microprocessor or a memory design. Performance metrics are based on the SPECint performance for microprocessors and access latency for memories and are normalized to the oldest shown product of the respective group.

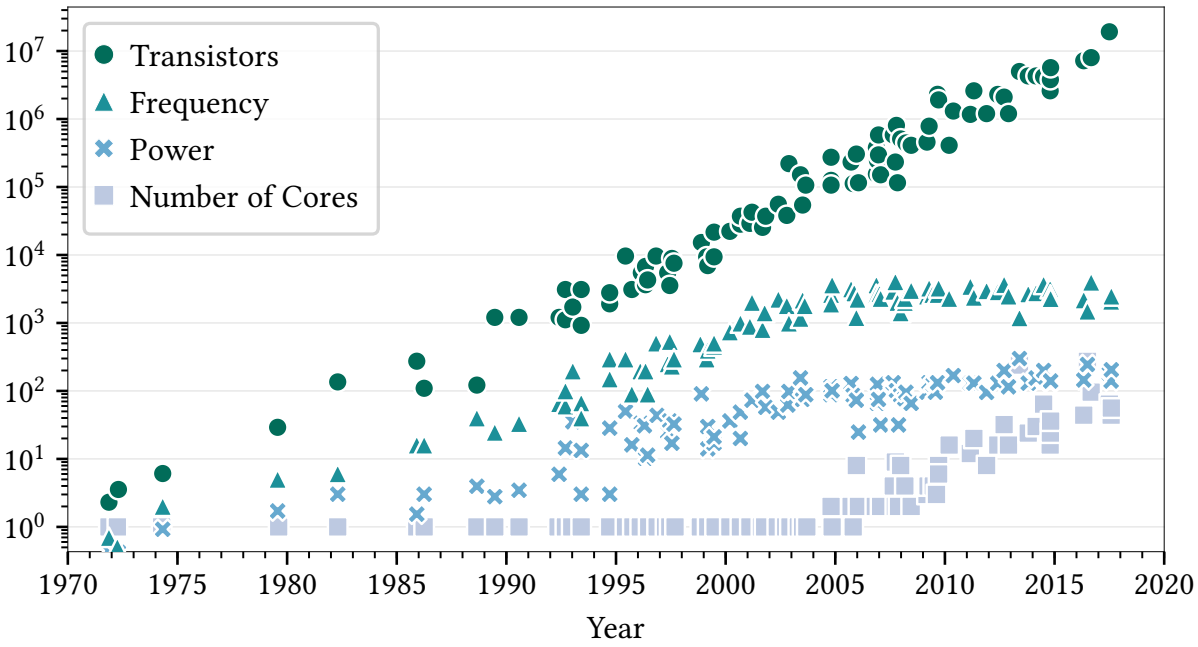


Figure 1.2: Historical trends of important metrics in computing systems.

Transistor count is presented in thousands, frequency in Hz and power in W. Original data up to 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten. Data from 2010 to 2017 collected by Rupp [151].

that current technological trends will eventually lead to the Memory Wall [177] regardless of the efforts put into optimizing the cache design. Nevertheless, improving the design of on-chip memory hierarchies remains an open challenge as one of the ways to delay the arrival to the wall.

Cache design has been influenced by several factors, such as the number of transistors and limitations in power and frequency. The historical trends of these processor properties are shown in Figure 1.2. Following Moore's Law, the number of transistors on chip has been increasing exponentially, which has enabled architects to spend more transistors on improving cache performance. As a direct consequence, caches have become larger, more complex (e.g., set-associative and multi-banked) and split into multiple levels to further offset the trade-off between cost and performance. Further advances were slowed down by the end of Dennard scaling [52] and reaching the Power Wall around 2005, which is observed in Figure 1.2 as the stagnation of the previously increasing trends for processor frequency and power.

The stagnation of performance achieved through increasing the size and complexity of caches has led researchers and architects to better utilize already available cache resources. To that end, significant advances have been made in the design of cache management techniques, improving

the performance in both execution speed and consumed energy. Cache replacement policies are developed to improve cache hit ratios and, thus, reduce unnecessary data movement between the processor and main memory. However, not all cache misses can be avoided[1], even with infinite cache resources and oracle algorithms. To reduce the latency of data access, prefetchers are used to pre-load the lines that are predicted to be accessed in the future. Nevertheless, the memory access patterns are often unpredictable. Moreover, prefetchers do not reduce memory traffic between the processor and the main memory and may pollute the cache by prefetching unnecessary lines, further decreasing the performance.

Data movement is predicted to be the main contributor to power consumption in future systems [23, 47, 101]. In order to further reduce the data movement through the memory hierarchy, the concept of *processing in memory* has been born [141]. The idea relies on placing computing resources near the place where data resides. In general, this enables performing simple operations on data without the need for transferring it to the core. In-memory computation has several positive effects on performance. First, it saves the energy necessary to transport data to the core and back. Second, data do not take space in the cache and therefore, it allows better cache performance both in terms of execution speed and energy. However, in-memory computation has seen limited use in High-Performance Computing (HPC) systems due to long latency and increased pressure on already congested memory buses. Recent proposals [125, 181] avoid the limitations while keeping the benefits of this approach by implementing it in the on-chip memory hierarchy, which is referred to as *near-memory computation*.

Another point of optimization within the memory hierarchy is the interface between microprocessor cores and main memory. The memory controller is a component that manages this interface and, therefore, it plays an important role in the optimal utilization of the limited memory bus. Formerly a part of the north bridge, the memory controller has been an integral part of the processor since the appearance of AMD K8 [172]. In current systems, memory controllers manage the contents of volatile memories, sort the memory requests coming from the cores and issue appropriate DRAM commands following the restrictions defined by DRAM standards [93]. The impact of the memory controller on the performance lies in the optimal scheduling of DRAM commands in order to exploit parallelism inside DRAM chips. In addition, since the beginning of the multi-core era, memory controllers ensure a fair allocation of the memory bandwidth to the co-running threads. Memory controller designs focus on prioritizing critical work at the instruction [73], thread [128, 129, 132] and application level [163].

[1]Different types of cache misses are explained in Section 2.1.2

The growing complexity of the modern parallel processors has increased the difficulty of writing codes that optimally utilize the existing hardware resources. This is especially the case in the HPC domain, where complex applications need to run efficiently on a large number of processors. Parallel programming models are a set of paradigms designed to ease the writing of such codes. Early implementations, such as pthreads [80] for shared memory and Message-Passing Interface (MPI) [64] for distributed memory systems, require manual specification of parallel work, handling of synchronization and ensuring the correct execution of programs. Further advances resulted in more programmer-friendly paradigms, such as the annotation-based approach offered by OpenMP [135]. A programmer employs simple annotations to specify units of parallel work and the appropriate synchronization between threads. The annotations are translated by a compiler to function calls provided by the programming model library. More recent proposals [136] introduce task-based parallelism. Contrary to parallel work units in the fork-join models, tasks can run asynchronously, enabling better load balancing and, therefore, better utilization of hardware resources.

Traditionally, microprocessors are designed to be agnostic to the code running on them. Besides, programmers do not need knowledge of the underlying hardware to write correct applications. This approach enables a high level of code portability, and thanks to this, it is widely adopted in general-purpose computing. However, the codes in the HPC domain are less restricted in that context as they are often optimized for specified machines where they are executed. This implies applying knowledge of micro-architectural details of the underlying hardware to improve the performance of applications. Many modern microprocessor designs [1, 78, 81, 122, 168] offer mechanisms to provide application-level information to the hardware, such as data prefetches, cache block invalidations and flushes, as well as hints to cache replacement policies. While this offers an opportunity to achieve better performance, a great effort is required from a programmer to fulfill that goal.

To exploit performance benefits offered by the hardware-software collaboration while maintaining the ease of programming, Casas et al. [35] propose a new paradigm based on runtime-aware architectures. At the heart of this approach is the runtime system that serves as a link between the parallel application and the underlying hardware. While its purpose is to ensure the correct execution of a task-based parallel code, its main potential lies in the knowledge of both the hardware and application-level details that can be exploited to improve the overall performance. From the software standpoint, knowledge about underlying hardware offers better scheduling decisions to optimally utilize modern heterogeneous systems. From the hardware perspective, the runtime system knowledge offers a view into the future that is

not available with traditional methods. The great potential of this paradigm is proven by many successful applications of the runtime system information on improving the performance of the whole system [9, 32, 36, 41, 44, 118, 119, 138].

1.1 Thesis Objectives and Contributions

The main goal of this thesis is to exploit the knowledge about a parallel application available at the runtime system level to improve the design of the on-chip memory hierarchy. The coupling of the runtime system and the microprocessor enables a better hardware design without hurting the programmability of the parallel systems.

The proposals presented in this thesis rely on modern directive-based parallel programming models, a powerful, programmer-friendly tool for producing well-performing and portable applications. The programming model is extended to allow the user to provide additional information about the parallel code that is not already present in the existing standards. The source-to-source compiler translates these annotations into the calls to the runtime system library. The runtime system exploits existing mechanisms to forward the information to the hardware, which then uses it to optimize the design of various components within the on-chip memory hierarchy.

1.1.1 Runtime-Assisted Insertion Policies for Last-Level Caches

The first contribution of this thesis is a set of insertion policies for shared last-level caches that exploit information about tasks and task data dependencies. The design of these policies is based on re-reference intervals. The intuition behind this proposal revolves around the observation that parallel threads exhibit different memory access patterns. Even within the same thread, accesses to different variables often follow distinct patterns. The goal of these insertion policies is to assign the appropriate re-reference interval to each cache line on its insertion in the last-level cache, taking into account the access pattern and its impact on cache performance of other cache lines.

The first policy considers the data dependencies between tasks defined by a programmer using programming model annotations. Data dependencies are classified into three groups depending if they are read-only (*in*), write-only (*out*), or both (*inout*). The dependency type statically determines whether the corresponding cache lines are treated by the replacement

policy as cache-friendly or trashing access patterns. Depending on their cache friendliness, the cache lines are assigned different re-reference intervals.

The second insertion policy utilizes the notion of task types to dynamically drive the assignment of re-reference intervals to new cache lines. On the software level, the runtime system is extended with a sampling mechanism that tracks the cache performance for each policy configuration. The execution is split into training and stable phases. During the training phase, the runtime system drives the cache configuration and evaluates each configuration's cache performance. The best performing setting is used during the stable phase. To allow for the changes in application behavior during the time, the runtime system periodically switches between the training and the stable phase. On the hardware level, the last-level cache is extended to take into account runtime-provided settings for the re-reference interval assignment.

1.1.2 Implementing Reductions in the Cache Hierarchy

The second contribution of this thesis is a runtime-assisted technique for performing reductions in the processor's cache hierarchy. The notion of near-memory computation [160] is applied to reductions, which are defined as operations where input data is accumulated by applying an operator to generate output data [76]. The goal of this proposal is to be a universally applicable solution regardless of the reduction variable type, size and access pattern.

On the software level, the programming model is extended to let the programmer specify the reduction variables for tasks, as well as the desired cache level where each reduction will be performed. The source-to-source compiler and the runtime system are extended to translate and forward this information to the underlying hardware. On the hardware level, private and shared caches are extended with functional units and accompanying logic to perform reductions at the cache level. This approach avoids the unnecessary data movements to the core and back as the data is operated at the level where it resides. Since this design does not modify the processor's instruction set architecture (ISA), the core is extended to facilitate proper instructions to the caches. Specifically, the load-modify-store chain is converted to a single store instruction that carries all information necessary to perform a reduction correctly. This store instruction is forwarded through the cache hierarchy until it arrives to the cache level that computes the reduction. If the reduction is performed in private caches, a final reduction takes place in the last-level cache once the reduction tasks are completed. The existing synchronization mechanisms in the runtime system manage correct execution and prohibit premature access to partially reduced data.

1.1.3 Criticality-Driven Prioritization in the Memory Hierarchy

The third contribution of this thesis is a runtime-assisted prioritization scheme for memory requests inside the on-chip memory hierarchy. The proposal is based on the notion of a critical path in parallel codes and the well-known impact of accelerating critical tasks on reducing the execution time of the whole application. In the context of this work, task criticality is observed at the level of task types as it enables simple annotation by the programmer. The acceleration of critical tasks is achieved by the prioritization of their corresponding memory requests in the microprocessor.

On the software level, the programming model is extended to allow the programmer to specify critical task types in a parallel code. The source-to-source compiler and the runtime system are responsible for translating and providing this information to the processor. Within the processor, memory requests issued by the core are assigned priorities depending on the corresponding task's criticality. Shared resources belonging to the on-chip memory hierarchy, such as the interconnection network, the last-level cache and the memory controller, consider the priority of memory requests when deciding their ordering. The design is viable for different organizations of caches and memory controllers and various implementations of interconnection networks.

1.2 Thesis Outline

The contents of this thesis are organized as follows:

Chapter 2 reviews the prior work on the topics related to this dissertation, beginning with the background on cache design and following with the algorithms for cache management. Further, it explores the designs of memory controllers and their policies. Next, the concept of near-memory computation is introduced with detailed insights into previous works on this topic. Advancing to the software level, it provides a background on parallel programming models and their runtime systems. Finally, the prior work on the holistic design of runtime systems and microprocessors is briefly explained.

Chapter 3 introduces the simulation infrastructure used for experiments described in the thesis, together with the description of the benchmarks used for the evaluation of the proposed designs.

Chapter 4 proposes two runtime-assisted insertion policies for shared last-level caches which use application-level knowledge to guide the placement of cache lines.

1.2 Thesis Outline

Chapter 5 presents an application of near-memory computation paradigm inside the on-chip memory hierarchy, specifically for performing reduction operations within the caches. The chapter introduces the challenges present in the context of in-memory reductions and proposes the solution that aims to minimize data movements while maintaining programmability.

Chapter 6 introduces the prioritization scheme for memory requests based on task criticality in a parallel application. After describing the context and problems in the previous designs, this chapter introduces the proposed solution's hardware and programming model extensions.

Chapter 7 summarizes the contributions presented in this dissertation and provides the possible directions for future work.

Chapter 2
Background

This chapter presents the background of the relevant hardware and software components in the context of the work developed for this thesis. The design of the on-chip memory hierarchy, i.e., caches and memory controllers, is presented in detail, including the previous work on improving the design of these components. Further, the chapter introduces the state-of-the-art work on optimizing the execution of reductions, which constitute an important algorithmic pattern in the HPC codes. The fourth section of this chapter is dedicated to the programming of parallel systems, specifically the programming models for shared-memory processors and the associated runtime systems. Finally, the fifth section introduces the concept of runtime-aware architectures that are a backbone of the work presented in this thesis, with a brief overview of related work.

2.1 Cache Memories in Microprocessors

In computing, a cache generally refers to a small and fast storage that keeps frequently accessed data. In the context of computer architecture, caches were introduced as a solution to the ever-increasing performance gap between processor and main memory (see Figure 1.1). They offer a lower latency and higher bandwidth compared to the main memory. On the negative side, they require more area and consume more power per unit of storage.

The first commercially available system that uses a data cache is IBM System/360 Model 85 [77] announced in 1968. Since then, the design of cache memories has become more complex as a response to the increasing processor-memory performance gap and enabled by the exponential increase of the number of on-chip transistors following the Moore's Law [123, 124]. Figure 2.1 illustrates the most common memory hierarchy designs encountered in the modern processors. One direction the development of caches has taken is the introduction of a multi-layered cache hierarchy, which employs several caches of increasing sizes and latencies.

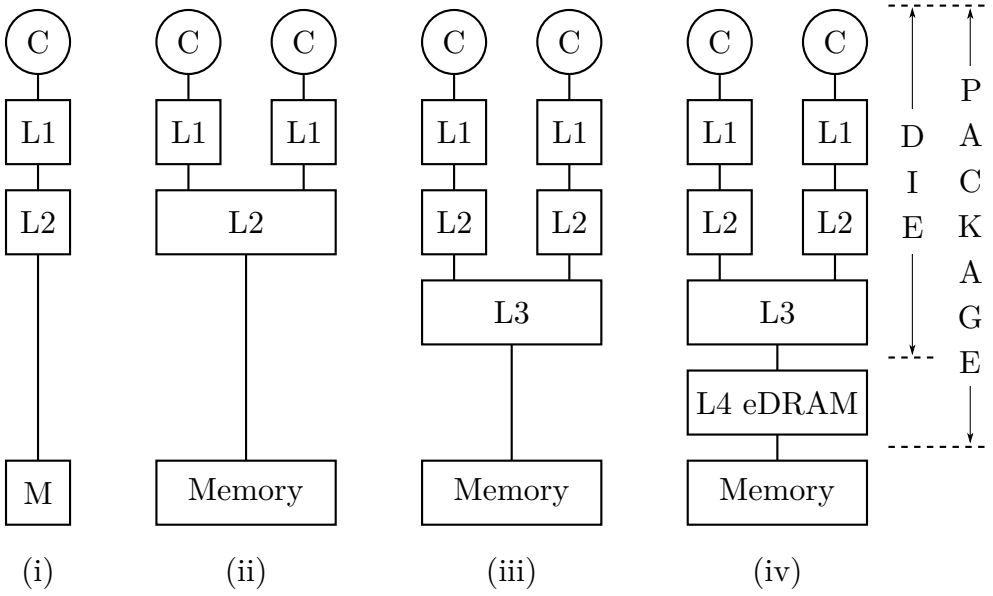


Figure 2.1: Typical memory hierarchy architectures

While most current processors implement a three-level cache hierarchy, designs marketed towards low-power applications use two-level caches (i). More recent chips introduce the fourth cache level, implemented in the DRAM process on a separate die. Another direction of cache design follows the introduction of multi-core processors. In general, the L1 cache is private to the core, while the lower-level cache can be either private or shared. Single-core and low-performance multi-cores usually employ two-level cache hierarchy where the second level is shared (i), (ii). More advanced multi-core designs employ three-level hierarchies (iii), while some of the recent processors add a L4 cache (iv). A cache that lies just before the main memory in the memory hierarchy is often referred to as the last-level cache (LLC).

The remaining of Section 2.1 presents the organization of the most-relevant cache designs and introduces the prior work on the policies for management of the cache contents with a special focus on the cache replacement policies.

2.1.1 Cache Microarchitecture

This section describes the organization of the on-chip caches. Modern microprocessors employ caching techniques for instructions, data, memory page-table, among others. Since the contributions of this thesis target data caches, this section focuses only on that cache type.

A building block of on-chip caches is a SRAM cell which consists of 4 or 6 transistors, depending on the implementation, contrary to a DRAM cell which is typically built of one transistor and one capacitor. As it does not rely on a capacitor to store a charge, a SRAM cell

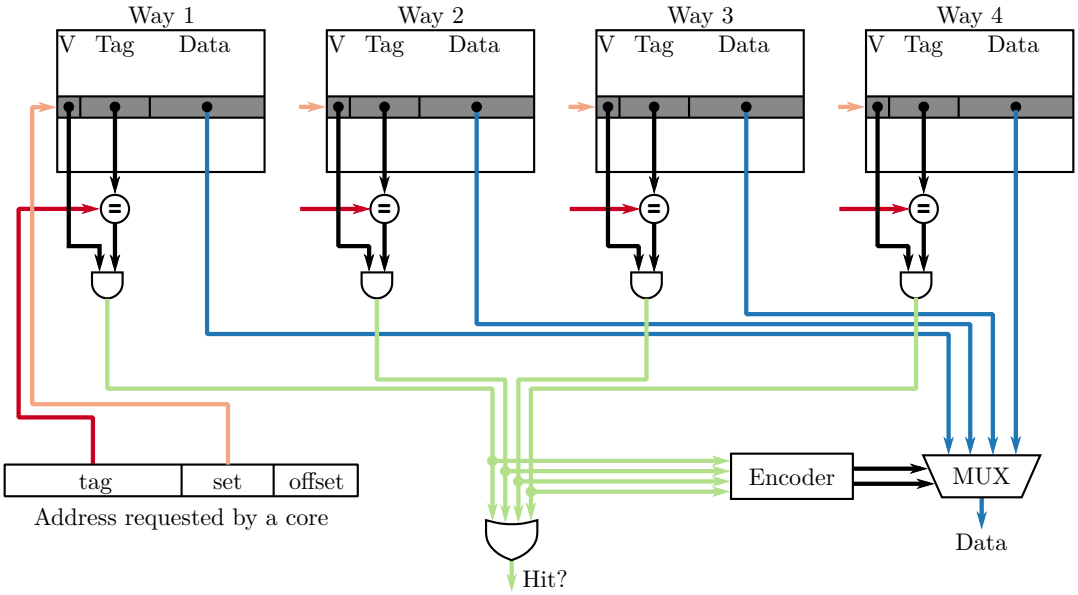


Figure 2.2: Logical organization of a 4-way set-associative cache with a data look-up circuit
The actual cache implementations use separate SRAM arrays for tags and data. Comparators are part of the tag storage which is implemented as content-addressable memory. Some designs split data arrays into banks.

does not require refreshing in order to maintain information. Moreover, it allows a faster access but is larger and consumes more energy that a DRAM cell.

The unit of storage for most common cache designs is a cache line, or often referred to as a cache block. It contains the data and the metadata necessary for the proper management of the data. The metadata consists of (i) the *tag* portion of the address, which is compared with the requested address during a cache look up, and (ii) flags, which represent status of the line, such as validity bit, dirty bit and bits related to the coherence protocol.

With regards to the organization of a cache, the simplest implementation is a so-called *direct-mapped cache*. In this design, each memory address is always mapped to the same physical cache block. Such design allows a fast access with a low energy cost. However, it can suffer from frequent conflicts when two distinct addresses are mapped to the same block. As a consequence, some access patterns can produce series of evictions and data fetches which slows down the execution and wastes energy. On the other end of the spectrum lies a fully-associative cache, where memory addresses can be mapped to any physical cache block. While they exhibit the best performance in terms of number of successful cache lookups, such cache designs are expensive in area and consume significantly more energy than direct-mapped caches. Set-associative caches combine these two designs to offer a trade-off between the cache performance in terms of cache hits, while keeping low area, access latency and energy consumption. A set-associative cache is split into cache sets. Memory addresses are directly

mapped to cache sets, while within a set, an associative lookup is used. Figure 2.2 shows an organization of a set-associative cache.

Most modern microprocessors employ set-associative caches with varying associativity. Caches with larger degree of associativity within the set observe better hit ratios at the cost of higher implementation complexity, latency and energy consumption.

2.1.2 Cache Management

Processor caches are accompanied with a set of algorithms that define their behavior and ensure an efficient and correct use of cache resources. These algorithms are implemented in silicon and usually are a part of the cache controller. An important goal in the design of these algorithms is improving cache hit ratios, or, in other words, reducing the number of cache misses. Cache misses can be categorized in three categories, usually referred to as three Cs: (i) *compulsory* misses occur on a first reference to a given address; (ii) *capacity* misses are a result of eviction of cache contents due to the limited space; (iii) *conflict* misses happen in direct-mapped and set-associative caches due to the same set-mapping of different addresses. There are cache management algorithms dedicated to reduce the number of misses corresponding to each of the mentioned groups.

Cache *write policies* define how data modifications are handled. *Write back* policy maintains a modified line (also called dirty line) in the cache and writes to the next cache when the line is evicted. *Write through* policy forwards the modifications to the lower-level cache on write. Allocation policies specify whether on a write miss a line is first brought to cache (*write allocate*), or the data is directly forwarded to the lower-level cache (*no write allocate*).

In some cases, a programmer may know that accessed datum does not have temporal locality. To prevent the data from unnecessarily occupying the cache, a hint can be given to the processor. Modern ISAs, like ARMv8 [14] and Intel x86 [81], offer such a mechanism via non-temporal memory operations.

Cache coherence algorithms ensure that changes to the shared data is propagated through the memory hierarchy which guarantees that cores always access the correct value. Coherence protocols are transparent to the programmer, contrary to memory hierarchy organizations like scratch-pad memories that have to be managed manually.

Prefetching schemes are able to significantly improve cache performance by fetching the cache lines that are predicted to be accessed in the near future, and, therefore, avoid some compulsory misses. However, making optimal predictions is not an easy task. Moreover,

wrong prefetching decisions may result in eviction of the useful data from the cache, aside from increasing the pressure on the memory bus.

Cache partitioning is a technique where caches are divided into portions according to certain criteria. This mechanism can be used to ensure fair share of cache resources among co-running threads.

Cache compression is a method to improve cache performance by compressing cache lines and thus increasing the effective capacity of the cache [7, 11, 133]. Future processors are predicted to have a reduced amount of memory per core due to power limitations. This makes compression a viable approach for improving cache performance for the next-generation systems. A drawback of cache compression is an increased hit latency. In programs that do not benefit from compression, this results in a worse performance compared to the systems without compression. Dynamic compression schemes overcome this issue by selectively applying compression taking into account the performance benefits and penalties of compression.

2.1.3 Cache Replacement Policies

Cache replacement policies are a set of algorithms whose main objective is to select a line to be evicted to make space for a newly fetched line. The complexity of the problem has led to its decoupling into multiple algorithms, i.e., insertion (also called placement), promotion and eviction policies. Simple replacement schemes, like First-In, First-Out (FIFO), do not take into account data reuse and, therefore, is not an attractive choice for a cache replacement policy. The optimal replacement policy in terms of miss ratio is the Belady's MIN algorithm [22], which evicts the line that is going to be referenced furthest in the future. However, it requires knowledge about the future, which renders it unusable in real systems.

Many techniques approximate MIN algorithm, the most well-known being the Least-Recently Used (LRU) policy. It uses a recency stack to logically order the cache blocks according to the last time they were accessed. While its implementation for software caches is trivial and efficient, the hardware implementation requires updating several entries of the stack on each access. Pseudo-LRU schemes are proposed to avoid the added energy cost that comes with updating the recency stack, such as Not-Recently Used (NRU) [164], binary tree-based policies [42], etc.

These simple replacement policies are not suitable for all scenarios. For example, LRU policy inserts new lines into the most-recently used position in the recency stack. Therefore, data that is never reused (e.g., streaming access patterns) can push other cache-friendly data to the bottom of the recency stack and cause their premature eviction. Many techniques are

built on top of these simple policies in order to further optimize cache performance for complex memory access pattern, especially in the domain of shared caches.

Qureshi et al. [145] propose several insertion policies that try to reduce trashing of cache lines. LRU Insertion Policy (LIP) inserts new lines in the LRU position, which is an optimal insertion decision for data with zero reuse. Bimodal Insertion Policy (BIP) combines LIP with the standard Most-Recently Used (MRU) insertion policy, so that a tunable percentage of lines is inserted in the LRU, and the other in the MRU positions. The objective of BIP is to prevent cache trashing by inserting randomly-selected lines at the bottom of the recency stack. If such lines are accessed before the next eviction, they are promoted to the MRU position. On the other hand, if the line belongs to streaming accesses, it will not pollute the recency stack. Dynamic Insertion Policy (DIP) chooses between BIP and LRU during the run-time and is able adapt to different workloads and the change of the workload behavior during the execution time. The switching between these two algorithms is achieved via Sampling Based Adaptive Replacement [146]. The cache performance of the two policies is tracked and the cache is set to follow the currently best-performing strategy. The main drawback of this policy is that it randomly assumes the access patterns of incoming lines without using high-level knowledge.

Jaleel et al. [88] propose a cache replacement policy that uses Re-Reference Interval Prediction (*RRIP*) in order to prevent lines, which are not going to be referenced for a long time, polluting the cache. The algorithm predicts whether a cache line is going to be accessed in a near or distant future, and accordingly assigns an *immediate* or *distant* re-reference interval. On eviction, lines with distant prediction are evicted. The predictor learning is performed during the program execution, on a hit to a cache line. The policy is composed of two algorithms that try to solve two different problems in caches. The first, Static RRIP (SRRIP) protects cache lines from being evicted by streaming access patterns. The second, Bimodal RRIP (BRRIP), solves the problem of trashing that appears in some workloads when using SRRIP policy and is analogous to the previously-described BIP. The selection of a better-performing policy during the run-time is done via Set Dueling [144]. Similarly to DIP, DRRIP does not consider the high-level information about memory accesses.

Wu et al. [176] propose Signature-based Hit Predictor (SHiP) to improve the RRIP-based policies by taking into account the signature of the cache reference during the prediction of re-reference interval. SHiP assigns a signature to each cache reference and records the reuse of lines corresponding to each signature. On insertion, lines are assigned a re-reference interval depending on the previous behavior. Cache lines exhibiting no reuse are assigned a *distant* re-reference interval, while the lines recorded to have reuse are assigned an *immediate*

re-reference interval. This policy is adaptable to recency-based policies, such as LRU, and supports different signature schemes that can be based on the program counter of the issuing instruction, the memory region the referenced address belongs to, etc.

Jiménez [96] proposes using arbitrary insertion, promotion and eviction schemes, controlled by an Insertion/Promotion Vector (IPV). The IPV defines how the order of lines in the LRU-MRU stack is changed on promotion. It also defines the position where a new line is inserted and the position from which a victim line is chosen. The main idea is to employ out-of-order promotion of cache lines, based on their position in LRU-MRU stack. The IPV is calculated by the use of a genetic algorithm that tries to maximize overall performance of an application. Dynamic mechanism, based on Set Dueling [144], is applied to choose between multiple predefined vectors, depending on the current workload.

Jain and Lin [87] propose Hawkeye, which applies Belady's algorithm on previous cache references in order to predict the future behavior of cache blocks. Hawkeye observes a set of previous cache accesses to determine whether a certain access would hit or miss following the Belady's algorithm. These observations drive the training of a PC-based predictor that classifies cache lines as *cache-friendly* or *cache-averse*. Cache-friendliness is taken into account by the replacement policy to favor evicting cache-averse blocks. Hawkeye does not consider the reuse of cache-averse lines and always associates them with a distant re-reference interval.

Multiple proposals are based on the idea of marking lines that will not be referenced again (dead blocks) as victims. Since identifying dead blocks requires knowledge of future, all techniques use dead block predictors in order to do so. Kharbutli and Solihin [104] and Khan, Tian, and Jimenez [103] propose hardware techniques for dead block prediction. The first one uses a counter-based approach, which marks lines as dead when number of certain cache events (e.g. number of accesses to a cache line) reaches a preset threshold. The other proposal uses learning based on events from a subset of all cache sets, via sampling of partial tags. Some compiler assisted techniques [154, 173] predict dead blocks during compilation and provide this information to the processor.

Previously described replacement policies take into consideration only information and metrics available at the hardware level. This information is used for the prediction of certain behaviors that drive the replacement policies. Therefore, the achieved cache performance depends on the quality of these predictions. Moreover, obtaining good predictions often requires additional hardware structures that further complicate the cache design. In recent years, there have been proposals that utilize the knowledge about the application at the software level to assist the decisions made in hardware. These approaches exploit high-level information

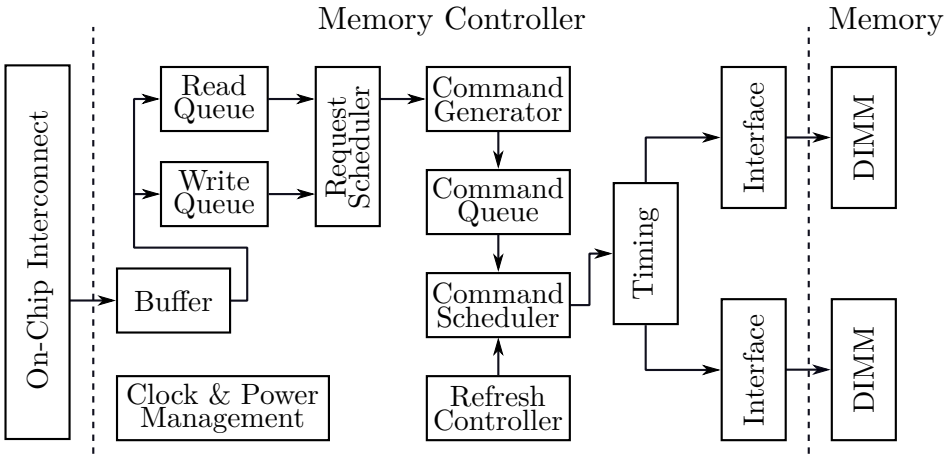


Figure 2.3: The schematics of the memory controller for DRAM memories

unavailable at the hardware level. Manivannan et al. [118] propose RADAR, a runtime-assisted scheme for dead-block management for task-based parallel applications. The proposal consists of two algorithms which are combined to give a better prediction of dead blocks. The first algorithm, *Look-ahead scheme* uses information about task dependencies and current state of the task dependency graph to determine whether certain blocks of data will be accessed again. The second, *Look-back scheme* uses previous outcomes of cache accesses to train branch a predictor-like structure whether certain cache lines will be dead or not. Same authors [119] expand the idea globally to all caches, which further improves the cache performance.

2.2 Memory Controller Design and Optimizations

This section introduces the background on main memories and the associated controllers and presents the state-of-the-art designs of request scheduling policies inside memory controllers.

2.2.1 Memory Controller Design

A memory controller is a component that lies between the processor and the main memory and has the role of transferring data between these two resources. Until early 2010s, memory controller was a part of the North-Bridge. Since Intel Nehalem and AMD Sledgehammer architectures, enabled by the increasing number of transistors on chip, the memory controllers have become a part of the processor die. A DRAM controller has several responsibilities and its internal structure is presented in Figure 2.3. The subcomponents of a memory controller can

be classified into queues and interfaces, scheduling logic, DRAM command generators and control logic. The following paragraphs describe the functionality of the relevant components.

As the memory bus is not bi-directional, read and write requests cannot be interleaved. In addition, the switching between read and write modes takes time, so the controllers generally issue larger bursts of read and write requests. Consequently, memory controllers are equipped with separate queues for read and write requests. The request scheduler has an objective to order memory requests coming from the processor in a way that maximizes the bandwidth and minimizes the access latency. Section 2.2.3 provides a detailed background on this topic and reviews the relevant state-of-the-art proposals.

Once the ordering of requests is determined, the controller translates each request into a set of DRAM commands necessary to fulfill that request. A DRAM command is a high-level abstraction of the signals to the DRAM chip that need to be set in order to perform a desired action. For example, a read request is translated into `Activate` and `Read with Auto-Precharge` commands, according to the JEDEC DDR4 [93] specification. The commands are then scheduled and issued to the DRAM chips through the respective interfaces, while following the timing constraints specified by the standard. Section 2.2.2 presents in detail an organization of DDR4 memory. DRAM is a volatile memory and therefore needs to be refreshed periodically in order to preserve the stored data. This is another task done by a memory controller. The Refresh Controller maintains the status of each memory block and periodically issues `Refresh` commands. Recent designs of memory controllers introduce power management schemes that reduce energy consumption. Some architectures provide mechanisms for improving security.

2.2.2 DRAM organization

A full understanding of various design choices inside a memory controller is not possible without a basic knowledge of the modern DRAM architectures. This section provides a brief background of the DDR4 design. A common organization of the main memory in the modern systems is shown in Figure 2.4. Processors offer multiple memory interfaces (channels). Each channel can hold one or two DIMM modules, each of which can have up to two ranks. Within a rank, the memory is further divided into banks, and banks into sub-arrays. Sub-arrays within the bank can operate simultaneously. However, since all components on the same channel share physical command, address and data buses, the accesses need to be serialized. Devices residing on different channels can function independently from each other.

Due to the technological properties of a DRAM cell, the content of a row first needs to be loaded into a buffer before it can be accessed. Every memory request is decomposed into three

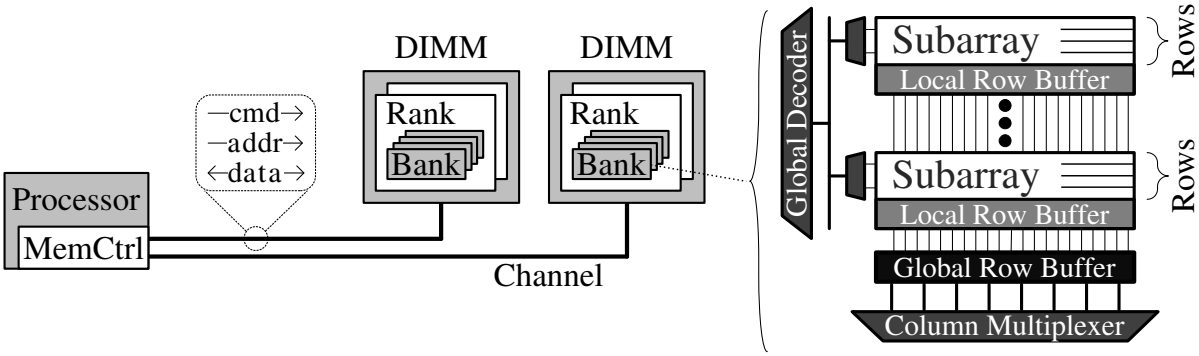


Figure 2.4: The layers of parallelism in modern DRAM designs. Left: The organization of the DRAM. Right: The logical architecture of a DRAM bank. The illustrations adapted from the work by Kim et al. [107]

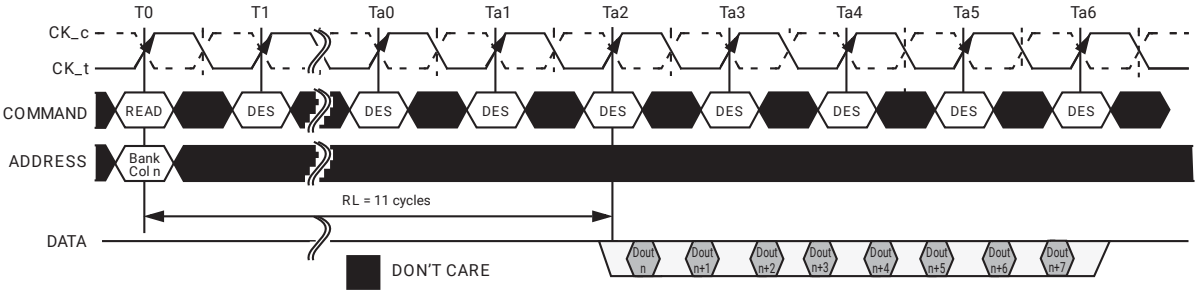


Figure 2.5: A timing diagram for a read command [93]

commands: (i) `Activate` reads a row from a sub-array into the row buffer; (ii) `Read/Write` accesses the selected column inside the row buffer, and (iii) `Precharge` writes the contents of the row buffer back into the corresponding row of the sub-array. Since `Activate` command destroys the original data in the row, the row buffer always needs to be written back before a new row is activated. Depending on the row buffer status, the latency of a memory access can vary significantly. For example, if a requested row buffer already holds the necessary data, only `Read` command needs to be issued. On the other hand, if the row-buffer holds the content of another row, the executed command sequence is `Precharge - Activate - Read`. In a Micron DDR4 SDRAM model MT40A [121] operating on 2400 MHz, each of these commands take 17 DRAM cycles.

A memory block corresponding to a cache line is distributed across banks in a rank. Therefore, to serve a LLC miss, memory controller simultaneously issues appropriate commands to the banks within the selected channel, device and rank. Figure 2.5 illustrates a timing diagram of signals on the command, address and data buses corresponding to a `Read` command. At time `T0`, the memory controller puts the address on the address bus and issues the command `Read`. After a certain number of cycles, the data can be sent via the bus to the memory controller

(cycle `Ta2`). Since the cache line size is 64B-long and data bus is 8B-wide, the data is sent in 8B chunks. DDR4 memories send the data on the rising and falling edge of the clock (*dual rate*) and, thus, the transfer takes 4 cycles.

This section has shown the great level of parallelism offered by the modern DRAM designs. However, to achieve high bandwidths, it is necessary to exploit the offered parallelism while respecting the timing restrictions. A careful design of memory controller schedulers is of utmost importance in order to achieve this task. Many techniques are developed with this goal in mind [140, 149, 169].

2.2.3 Memory Request Prioritization

The importance of proper scheduling of DRAM commands in order to exploit the parallelism offered within DRAM modules is demonstrated in Section 2.2.2. This section turns the focus onto the scheduling of memory requests that takes place before they are converted into DRAM commands. The proper request scheduling aims to exploit row buffer locality, guarantee fair share of the memory bandwidth across co-running threads, prioritize time-sensitive requests, etc. The following paragraphs provide a brief description of the previous work on this topic.

First-Come, First-Served (FCFS) is the most basic scheduling algorithm used in many fields such as computing and telecommunications. Requests are processed in the order they arrive at the resource. FR-FCFS (First-Ready FCFS) [149] extends the FCFS algorithm by taking into account the locality of the row buffers inside DRAM chips and, thus, it reduces the number of expensive *Activate* and *Precharge* commands.

Mechanisms that work well for single-thread systems may not be suitable for multi-core processors due to various issues such as a lack of fairness and starvation. FQM [132] is a scheduling algorithm for chip multi-processors that tries to achieve fairness among the threads by applying QoS mechanisms from the networking domain. Each thread is assigned a percentage of the total memory bandwidth. Excess bandwidth from one thread can be distributed to other threads depending on the previous traffic intensity of each thread. STFM [129] tries to achieve a fair use of memory resources by threads by taking into account the performance losses of each thread when executed in parallel with other threads compared to the solo execution. PAR-BS [128] prevents thread starvation by creating temporal batches of memory requests and prioritizing requests belonging to the oldest batch. The common drawback of these proposals is the unawareness of the thread's access pattern and its variability during the execution.

ATLAS [106] defines a new metric of accumulated memory service time by each thread. Similarly to PAR-BS, time is split into periods and, in each time period, the threads with lower

value of the metric in the previous time blocks are given a higher priority. In addition, the design is sensitive to the changes of the memory behavior of each thread by tracking the history of the accumulated service time per thread.

Since static schemes do not always achieve the best performance for a wide range of applications, researchers have proposed many adaptive scheduling algorithms. Hashemi et al. [73] identify that, so called, *dependent* cache misses are an important contributor to performance degradation when on-chip contention is present. This scenario occurs when an instruction causing a cache miss also depends on another instruction that results in a miss. They propose a scheme where some operations resulting in misses are off-loaded to the Enhanced Memory Controller. As a result, serialized dependent misses are issued earlier which reduces the waiting time for the second miss. BLISS [163] splits co-running applications into two groups depending on their sensitivity to memory interference. The sensitive group is given more priority in order to reduce the performance degradation caused by interference. Ipek et al. [85] propose a memory controller scheduling scheme based on reinforced learning.

However, the mentioned proposals focus only on the information visible to the core, which is generally observed on relatively short time intervals of several thousands of CPU cycles. Such fine-grained observations cannot capture the macro trends in the whole application as well as the impact of prioritization on the overall performance.

2.3 Reductions and Near-Memory Computing

2.3.1 Reductions: A Brief Overview

In the context of parallel programming, reductions are operations where input data is accumulated by applying an operator to generate output data [76]. In the context of this thesis, a reduction variable is defined as a data structure that holds the output of the reduction. Addition and multiplication are commonly used as reduction operators in scientific computing. Reductions can be parallelized because these operations are associative and commutative [142]. Based on the reduction variable's size, reductions can be classified into two categories:

(i) Reductions over scalar-types. This type of reductions occurs in a wide range of application domains including combinatorics (e.g. satisfiability problems such as n-Queens), scientific computing (e.g., normalized residuals to verify convergence or code correctness), to implement hardware performance counters, etc. On current mainstream architectures, updates to shared

data should be avoided, since they may result in high cache coherence traffic that significantly impacts execution performance.

(ii) Reductions over vector-types. Such reductions are usually present in more complex scientific codes that accumulate results on arrays or higher-dimensional matrices. Depending on how the reduction variable is accessed during the reduction, two sub-categories are identified: near-linear access patterns and irregular access patterns. Near-linear reductions typically take place in scientific codes where operations access just the neighboring grid elements, such as in LULESH [99] and SPECFEMD [109]. Irregular array-type reductions are frequently found in n-body codes, histogram computations, as well as in applications where a data structure representing a physical domain is accessed in an irregular manner. Concurrent execution of vector-reductions requires solutions that avoid unnecessary data privatization, prevent data races due to concurrent updates to overlapping memory regions and effectively reduce memory bandwidth and latency requirements.

The following sections present a detailed view into current solutions for an efficient reduction computation. It starts with software-based approaches and continues to hardware-accelerated solutions. Since the reductions are a special case of operations performed in modern computing systems, this chapter also introduces the paradigm of near-memory computing, which is applied by some of hardware-based solutions to optimize reductions.

2.3.2 Software Support for Reductions

There are two intuitive software-based techniques to parallelize reductions. The first approach, called *privatization* [26, 174], consists in having each of the threads involved in the parallel execution performing a partial reduction over its portion of input data. The partial reduced data is stored in private per-thread copies of the reduction variable. Partial results are combined in the final result after the parallel reduction tasks are completed. Privatization works well for reductions over scalars and small arrays. For larger reduction variables, however, privatized data increases cache pollution as the multiple copies of the same data occupy scarce cache resources.

As an alternative, private threads directly update the shared reduction variable. To ensure correctness, the update operation is guarded using *atomic* instructions that are commonly implemented in modern processors [1, 14, 78, 81]. This method performs worse than privatization for small reduction variables due to frequent cache misses caused by the invalidations of cache lines in the private caches as well as the increased coherence traffic.

More advanced schemes combine privatization and direct accesses to preserve benefits and minimize the drawbacks of these solutions. LocalWrite [71] avoids data races by reordering iterations among different threads. A compile-time analysis determines data segments accessed by each iteration of a parallel loop. During the execution, the reduction variable is split among threads. Iterations are assigned to threads in a way that each thread updates only the data it owns. This technique suffers from load-balancing issues when a single memory location is updated by many iterations. PAE [70] reduces the overhead of privatization by using it only for conflicting updates, while non-conflicting accesses are performed to the global space protected by a lock. Yu and Rauchwerger [179] applies different reduction techniques depending on the properties of the access pattern to the reduction variable. Depending on the compile-time characterization of the parallel code, one of the existing reduction parallelization approaches, such as privatization, selective privatization or iteration reordering, is used. These solutions require the knowledge of the iteration space, making them applicable only to algorithms with a static iteration space [72]. PIBOR [45] combines privatization and redirection to achieve linear updates to private copies of reduction variable. OmpSs-RM [46] formalizes support of the software techniques for declarative parallel programming models.

Transactional Memory (TM) [75] offers mechanisms that can be used for implementing reductions [68]. Software TM implementations utilize existing lock and atomic operations and, thus, have similar drawbacks as other previously discussed solutions that use these operations. Massively parallel processors (GPUs) offer primitives used by algorithmic proposals for efficient execution of reductions [51, 61]. While these approaches are effective on GPUs owing to efficient synchronization and lock-step execution inside a warp, they are not applicable to general purpose processors, where these mechanisms are not present.

2.3.3 In-Memory and Near-Memory Computation

In the context of the current HPC systems, in-memory computing corresponds to performing calculations directly in the memory silicon. The benefit of this approach is the direct access to high concurrency offered by the DIMMs (Dual Inline Memory Modules) via bank-, subarray- and rank-level parallelism. However, the incompatibility between the fabrication processes of DRAM and logic circuits [108] makes it unfeasible to efficiently incorporate a complex computing logic inside DRAM chips. As a consequence, in-memory operations often rely on exploiting the properties of the DRAM design, which limits the range of supported operations.

RowClone [156] performs data initialization and bulk-copy directly within the DRAM without the need to transfer the data outside of the memory chip. Intra-subarray copying is

achieved by reading the source data into the row-buffer and, then, writing it in the destination row of a DRAM array. To copy the data between different subarrays, RowClone exploits the shared internal bus inside the DRAM chip. Ambit [157], DRISA [116] and NAND-Net [105] implement the support for performing simple logic operations, such as AND, OR, NOT and NAND, directly in the DRAM. These simple operators can be used to construct more complex logic bitwise operations. ComputeDRAM [65] improves the previous work by supporting copy and Boolean AND and OR operators with commodity DRAM chips.

Near-memory computing represents a middle-ground between the traditional Von-Neumann architecture and the architectures employing processing in memory (PIM). Computing logic is placed near memory to exploit low latency and high bandwidth of near-memory data accesses. Moreover, implementing the computing hardware on a dedicated logic layer enables a support for complex operations. Consequently, a wide range of applications can be directly supported without the need to translate complex operations to basic operators supported by the PIM designs.

The concept of processing near memory appears in the literature as early as 1960s [100, 162]. V-IRAM [110] is an example of an architecture for near-memory processing that places a dual-core processor and a vector functional unit next to a DRAM silicon. One of the challenges encountered in these early designs is a limited amount of memory on a single chip, which restricted the applicability of such solutions to only less memory-intensive applications. The technological advances of chip fabrication helped to overcome this challenge by introducing 3D-stacked memory designs that can pack significantly more DRAM cells on a single chip. HBM [94, 114] and HMC [139] are commercial implementations of a 3D-stacked memory connected to a logic layer via through-silicon vias (TSV) [86]. Many proposals exploit the near-memory computation capabilities offered by these designs. Active Memory Cube [131] uses HMC as the base of its design. The logic layer incorporates the controller with functional units able to perform a wide set of atomic operations such as floating-point addition and logic operations. Graph-PIM [130] and Tesseract [5] are other examples based on HMC that apply the near-memory computing paradigm in the context of graph applications. Singh et al. [160] evaluate previous works on the topic of processing near-memory and identify that certain challenges currently prevent a wide adoption of these designs. While they provide notable performance improvements over the traditional paradigms, a lack of programming model support and the resulting increase in application complexity are still open issues in the current state of the art.

2.3.4 Computation in On-Chip Memory Hierarchy

Computing in the cache hierarchy is a natural evolution of the PIM paradigm applied to the CPU's on-chip memory hierarchy. Early designs modify SRAM cells to support simple operations directly in the cache [95, 98]. One of such approaches is Compute Caches [3] that implements operations such as copy, search, compare and logical operators in the caches. To be able to benefit from these hardware capabilities, applications need to be rewritten. Specifically, complex operations have to be decomposed into the supported simple operands.

Many previous designs offer out-of-core computation capabilities for efficient execution of atomic operations. Scatter-Add in data parallel architectures [6] targets reductions in SIMD/vector/stream memory systems [48]. ARM Cortex A75 [13] adds support for *far atomics* in the last-level cache. Power9 [63, 79, 182], Cray T3D [102], Cray T3E [155] and SGI Origin [112] implement atomic operations inside the memory controllers. NYU supercomputer [69] goes beyond the computing node by supporting atomics inside network switches. The operations are restricted to integer additions and logic operations. This mechanism allows an efficient implementation of synchronization primitives, such as barriers and locks. However, the performance is limited for more complex reduction operations encountered in codes from the HPC domain.

Some proposals address this issue by offering efficient in-cache computation targeted to a wider set of applications. Often-encountered programming patterns in HPC applications are reductions. Reductions apply an operator on input data to produce output data. Challenges in efficient implementation of reductions come from the need to concurrently update a single memory location by different co-running threads. PCLR [67] offers a solution for this problem by implementing a hardware privatization inside caches. The concurrently updated data is allocated to private cache lines not covered by the coherence protocol. This avoids conflicts caused by the coherence protocol due to simultaneous stores to the same location. The partially reduced data is combined in the last-level cache on eviction or by an explicit flush. PCLR requires inserting appropriate configuration and synchronization calls in the application. COUP [181] is a more general approach that targets any concurrent type of operation and requires less modifications in the application. As a result, it does not require synchronization within the application as the coherence protocol takes care of the correct execution. CommTT [180] improves COUP by supporting wider range of commutative operations in a speculative context of hardware transactional memories. PHI [125] is another hardware-based approach for coalescing and buffering of scattered updates in private caches targeted to graph applications. It relies on

functional units inside cache controllers to perform the computation and does not require coherence protocol modifications, contrary to COUP.

Current hardware support for reductions is constantly improving. However, current designs offer solutions targeting specific application domains. In addition, they lack a proper programming model support and adaptability to different scenarios. RICH [56] is a proposal that addresses these issues and is presented in detail in Chapter 5.

2.4 Parallel Programming for Shared-Memory Systems

This section provides a brief overview of parallel programming targeted to the shared-memory computer systems. First, it introduces challenges that come with programming modern parallel processors. Then, it presents background on parallel programming models with a special attention to task-based programming models. Finally, it describes OmpSs, a data-flow task-based programming model used as a basis for the work developed in this thesis.

2.4.1 Parallel Processors

As a result of the end of Dennard's scaling around 2005, the improvements of the single-thread performance have hit a plateau (see Figure 1.2). As a consequence, and enabled by the Moore's Law, the first multi-core processors were introduced not long afterwards. Further following of technological advances in microprocessor manufacturing has led to the current situation where parallel processors contain tens of cores, such as 58-core Intel Xeon Platinum [83] and 64-core AMD EPYC [2]. Some recent designs employ heterogeneous architectures, which combine low-power, slower cores with high-performance cores. ARM big.LITTLE [12] architecture is an example of heterogeneous processor designs.

Most multiprocessors employ a shared memory on chip, and even between multiprocessors on the same multi-socket node. Using a shared memory enables a direct access to the data by any core. This also means that any core can modify the data, so proper synchronization techniques are necessary in order to avoid race conditions. Even though all data in a shared memory system is accessible by all cores, the accesses to a distant memory (e.g., another socket) take more than local accesses, which is known as Non-Uniform Memory Access (NUMA). There is a great number of factors that need to be accounted for when designing parallel applications. As a consequence, producing a well-performing parallel code that efficiently uses available hardware resources is not a trivial task.

2.4.2 Parallel Programming Models

Early approaches to programming shared-memory parallel machines were based on the concept of threads. Some implementations were provided as a part of the operating system, such as POSIX threads [80] in Linux and Win32 threads [24] in Windows OS. In addition, industry-standard programming languages like C++ [84] and Java [134] have a built-in support for threading. Thread-based parallelization requires a programmer to create threads and manage their lives, to ensure correct synchronization via locks, semaphores and monitors. This is achieved by inserting in the code appropriate function calls defined in the API of a threading library. Another approach to parallel programming has come from a distinct effort as a directive-based programming. OpenMP [135] is a de facto standard programming model from the directive-based family. The programmer uses pragmas to define the units of parallel work which are translated to the functions implemented in the supporting library. Such an approach makes the directive-based programming model easier to use compared to the library-based solutions.

The first implementations of the OpenMP standard (until version 2.5) were focused on the fork-join paradigm, where threads are spawned and destroyed at the same time, effectively dividing the application in sequential and parallel regions. This is achieved by manually defining parallel sections using `pragma omp parallel`. Inside the threads, the work can be distributed by means of work-sharing constructs, such as the *for construct*. In the case of for loops, this is achieved using `#pragma omp parallel for` annotation before a for loop. The supporting library automatically creates threads and distributes the loop iterations among them. At the exit of the for loop, the execution is synchronized by an implicitly added barrier. This guarantees that the main thread does not continue with the execution until all threads have completed. To control the visibility of the local variables to other threads executing the same loop, the programming model offers `shared` and `private` annotations. To support mutual exclusion between threads, programmers can use `atomic` constructs to guard the access to a shared variable. The OpenMP specification offers customizing the scheduling policy to achieve the best load balancing among threads using a directive `schedule`. The behavior of OpenMP programs can be modified without re-compilation and before the execution using environment variables. The most used such mechanism is manual selection of the number of threads used for executing the parallel work. This can be achieved using `OMP_NUM_THREADS` environment variable, `num_threads` clause or `omp_set_num_threads` runtime routine.

2.4.3 Task-Based Parallel Programming

As an alternative to fork-join parallelism, some programming models use a notion of task as a unit of parallel work. Tasks are viewed as portions of the serial code that can execute asynchronously with other tasks while respecting the synchronization points between them. The programmer splits the sequential code into tasks and defines the dependencies between them. In the case of directive-based programming models, a source-to-source compiler translates the annotations to the function calls implemented in the supporting library.

The initial task is an implicit task that starts at the beginning of the application. During the execution, the initial task creates user-defined tasks until it arrives to a explicit synchronization primitive, such as `taskwait` in OpenMP, which pauses the initial task until all the tasks that were created by the initial task (children tasks) complete. Upon its creation, a task is added into a task dependency graph (TDG) as *pending* task. A TDG is an acyclic directed graph whose nodes represent tasks and edges correspond to dependencies between tasks. When all dependencies of a task are fulfilled, meaning that all the predecessor tasks completed their execution, the task is considered as a *ready* task. Parallel threads are assigned a task from the ready queue according to scheduling criteria, such as data locality, user-defined task priorities, scheduling policy, etc. Upon finishing the execution of each task, the runtime system updates the TDG and the pending tasks that have all dependencies satisfied are promoted to the ready-task queue. This process continues as long as the TDG has pending tasks or until the initial task comes to an end.

Examples of task-based programming models are OpenMP 3.0 [135] and Cilk [27] for shared-memory systems and Charm++ [97] for distributed-memory systems. Contrary to these implementations that are based on the extensions to the existing programming languages, Chapel [38] is a novel language designed specifically for the development of task-based parallel applications.

Data-flow task-based programming models, a subset of previously described programming models, employ the idea of defining the dependencies between tasks by specifying the data they produce and consume. A task is observed as a code that consumes certain input data to produce the output data. This mechanism is another step in the evolution of programming languages and further separates the application code from the task management primitives. Instead of specifying direct dependencies between tasks, a programmer defines the input and the output data for each task by specifying memory regions corresponding to this data. The runtime system constructs the TDG of an application by respecting the input and output dependencies

of tasks. From this point, the tracking and scheduling of tasks function in the same manner as previously described for the general case of task-based programming models. OpenMP extends the tasking constructs with the support for data dependencies in the standard 4.0 [136].

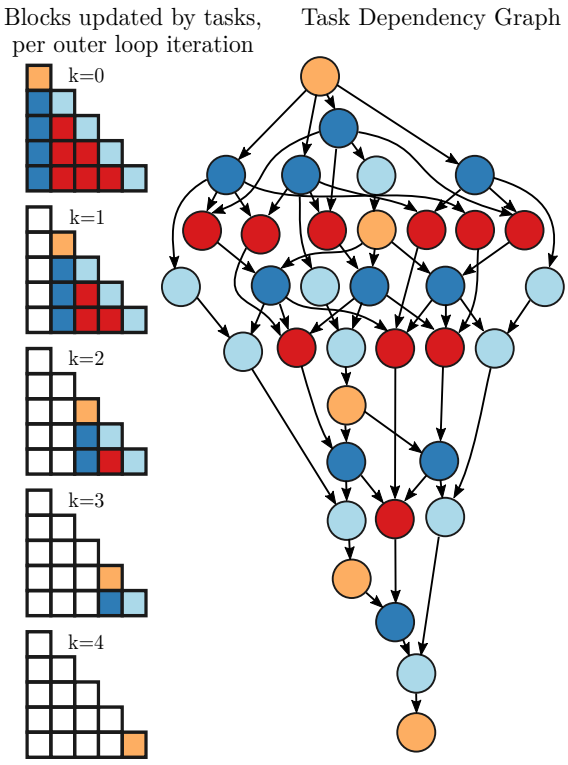
2.4.4 OmpSs Programming Model

OmpSs [20, 60, 167] is an example of a data-flow task-based parallel programming language. It is developed as a research-oriented programming model and several of its important features have been introduced in the OpenMP standard, such as the notion of tasks and data dependencies. A programmer makes use of simple annotations offered by OmpSs to define tasks in a sequential code. The annotations are translated with Mercurium [18], a source-to-source compiler, to function calls implemented in the runtime system library, Nanos [19]. Some of the capabilities implemented by these functions are task creation, calculation and tracking of their dependencies using a task dependency graph, scheduling of ready tasks, synchronization, etc.

Figure 2.6 (i) shows a source code of Cholesky factorization of a 2D matrix implemented in OmpSs. Algebraic functions `potrf`, `trsm`, `gemm` and `syrk` are defined as tasks using the `#pragma omp task` annotations. For a given task, the data it consumes is denoted with `in` directive, while the results of the computation are denoted using `out` directive. The data that is both consumed and produced in a given task is annotated as `inout`. For example, in case of `gemm` task, which performs a matrix-matrix multiplication, its input matrices `Ah[k][i]` and `Ah[k][j]` are denoted with `in`, while the resulting matrix, `Ah[j][i]`, is annotated with `inout`. A part of the dependency specification is the size of dependency, which in the case of matrices accessed by `gemm` task (painted in red) is `ts`×`ts`. As explained in Section 2.4.3, the information about task data dependencies is used by the runtime system library during the execution to dynamically determine the dependencies between tasks using a TDG. For example, `potrf` (painted in orange) from the iteration corresponding to `k = 1` depends on `syrk` (painted in light blue) from the previous iteration as it consumes the matrix `A[1][1]` produced by the task `syrk`. Figure 2.6 (ii) shows the mapping of the tasks to the blocks of matrix for each iteration (left) and the TDG containing all the tasks (right). This example clearly illustrates an asynchronous execution of tasks. The tasks belonging to different iterations can execute concurrently if they do not depend on each other, which is not possible in a fork-join programming model without a significant code reorganization. For example, `potrf` (orange) does not depend on `gemm` (red) from the previous iteration and, therefore, they can execute simultaneously.

```
1  for (int k = 0; k < nt; k++) {
2    // Diagonal Block factorization
3  #pragma omp task
       ↪ inout([ts][ts](Ah[k][k]))
4    potrf(Ah[k][k], ts, ts);
5
6    // Triangular systems
7    for (int i = k + 1; i < nt; i++)
8    #pragma omp task
       ↪ inout([ts][ts](Ah[k][i]))
       ↪ in([ts][ts](Ah[k][k]))
9      trsm(Ah[k][k], Ah[k][i], ts, ts);
10
11   // Update trailing matrix
12   for (int i = k + 1; i < nt; i++) {
13     for (int j = k + 1; j < i; j++)
14     #pragma omp task
       ↪ inout([ts][ts](Ah[j][i]))
       ↪ in([ts][ts](Ah[k][i]),
       ↪ [ts][ts](Ah[k][j]))
15       gemm(Ah[k][i], Ah[k][j],
             ↪ Ah[j][i], ts, ts);
16
17   #pragma omp task
       ↪ inout([ts][ts](Ah[i][i]))
       ↪ in([ts][ts](Ah[k][i]))
18     syrk(Ah[k][i], Ah[i][i], ts, ts);
19   }
20 }
21 #pragma omp taskwait
```

(i) Source code with annotated OmpSs data-flow tasks.



(ii) Mapping of tasks on blocks of a 5-by-5 blocked matrix (left). An example TDG with color-coded tasks (right).

Figure 2.6: Cholesky factorization parallelized with OmpSs. Source: Jaulmes [89]

2.5 Runtime-Aware Architectures

Historically, the design of hardware and software has been decoupled in order to ease the programmability and provide code portability. However, in applications where achieving a good performance is crucial, it is important to fully exploit hardware resources. To that end, the hardware implementation details need to be known at the software level. In addition to this requirement, the hardware designs is becoming more complex which makes efficient programming of such systems more difficult.

Contrary to software, current hardware designs, in general, do not require high-level information about the software. Most hardware optimizations take into account only the information already available as a direct consequence of the software execution, such as sequence of executed instructions in case of instruction caches [150], the execution flow in case of branch predictors [161] or data access patterns in case of hardware prefetchers.

The limiting factor for further performance improvements at the hardware level is the availability of high-level information. The information about software execution currently used in the processors is of limited time range and cannot provide a precise view into the future. In addition, such approach complicates the hardware design. In order to use high-level information about the application, an appropriate mechanism for transferring that information from software to hardware level needs to exist. Processor manufacturers have been reluctant to bridge the gap between hardware and software due to portability issues, among others.

Valero et al. [170] propose a concept of Runtime-Aware Architectures where the hardware and software are designed in a holistic manner. The bridge between these two layers is a runtime system, which takes over the responsibility of managing the hardware and software components, without directly exposing them to each other. This can enable the implementation of a broader set of optimization techniques that are not feasible in the current computer designs.

Casas et al. [35] further develop the idea and explore the potential use of runtime system-level information in the hardware and software design. From the software side, this approach allows better programmability, which in the current era of multi- and many-core heterogeneous systems with different ISAs [171] is surely an issue. It enables quicker development cycles and a better performance of parallel codes. From the hardware point of view, this approach opens new horizons in the context of processor design. The information about a parallel application can be used to improve decisions at the hardware level. This may ultimately lead to a better overall performance, lower energy consumption and reduced complexity of future computers.

A large amount of work has been done on the hardware-software co-design. At the software layer, Chronaki et al. [43, 44] propose task scheduling algorithms for heterogeneous systems. Chasapis et al. [39] develop a runtime-guided approach to model manufacturing power differences and propose a job scheduling algorithm for power-restricted NUMA systems [41]. Castillo et al. [37] design a runtime-assisted management of the cores' frequency depending on the criticality of running tasks. Brumar et al. [30] introduce memoization of task data dependencies to predict the outputs of a task without loosing accuracy. Sánchez Barrera et al. [152, 153] perform partitioning of the task-dependency graph in order to reduce the data movement in NUMA systems.

The runtime system-level knowledge has seen a great utility in the optimization of cache memories. Garcia et al. [66] and Papaefstathiou et al. [138] propose prefetcher schemes for task-based parallel programming models. The runtime system instructs a hardware prefetcher to fetch the data necessary for the execution of the next task. RADAR [118] is a dead-block prediction scheme that exploits the information about task lives to evict the data that will

not be accessed in future. Manivannan et al. [119] expand this idea and apply it to all cache levels. Caheny et al. [31, 33] propose to reduce cache coherence traffic in NUMA systems and to deactivate coherence for data that does not need it, which is determined by a runtime system [32]. A runtime-guided management of scratchpad memories is introduced by Alvarez et al. [9, 10]. Runtime systems can also aid in handling of the modern memory designs, such as stacked DRAM [8] and hybrid DRAM/NVM memory architectures [117]. Jaulmes et al. [90–92] explore a usefulness of runtime systems for reliability. Caminal et al. [34] study the effectiveness of manual and automatic vectorization in task-parallel programs. Castillo et al. [36], Etsion et al. [62], Kumar, Hughes, and Nguyen [111], and Tan et al. [165, 166] develop hardware accelerators for selected components of the runtime systems for task-based parallel programming models.

The enormous potential of the runtime aware architectures is proven by a wide body of previous work on this topic. The attractiveness of this approach lies in the fact that it enables improving the performance of current and future systems using simple designs while maintaining the ease of programming of such systems. The goal of this thesis is to exploit the runtime system-level information to optimize the design of on-chip memory hierarchies.

Chapter 3

Experimental Methodology

This chapter presents the methodology followed in this thesis. The first section describes the simulation infrastructure, including the environment used for tracing applications, the simulators used for the evaluation and the details of simulated architectures. The second section introduces the benchmarks used for the evaluation. It further provides the changes made to the benchmarks and the input parameters used for tracing. Finally, the third section briefly presents the metrics used to evaluate proposals developed in this thesis.

3.1 Simulation Infrastructure

3.1.1 Simulators

Figure 3.1 shows the tool-chain used in the evaluation of this thesis' proposals. The left-hand side of the figure illustrates the trace generation of a parallel application, while the right-hand side shows the process of obtaining performance metrics using the generated trace and hardware description. The remaining of this section describes these tools in detail.

A shared-memory multiprocessor system used as a baseline system and the architecture implementing designs proposed in this thesis are simulated using TaskSim, a trace-driven cycle-accurate architecture simulator [147, 148]. TaskSim simulates in detail the execution of parallel applications with OpenMP and OmpSs pragma primitives [60] on parallel multi-core environments. An input to a simulation is a trace of an execution of a parallel code and a specification of the simulated system's microarchitecture. A trace is obtained in a two-step process using a serial execution of a parallel code. The first step captures the structure of the parallel code, such as tasks, their dependencies and synchronization events, by recording instrumented calls to the runtime system library. The second step records the sequence of the executed instructions for each parallel task using a DynamoRIO-based tool [28, 29] as the

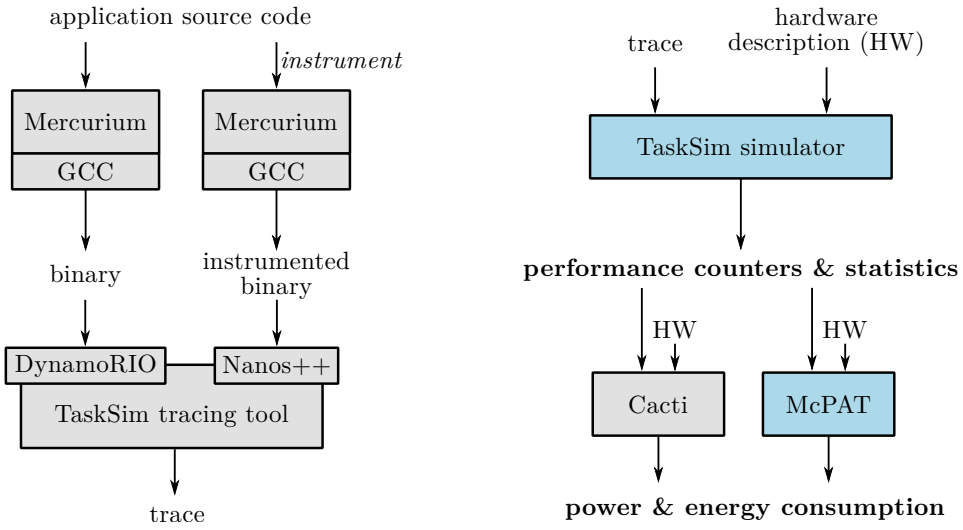


Figure 3.1: Overview of tools used in the evaluation. Text in bold represents the metrics used to evaluate the proposals. Tools colored in blue are modified to support the proposed designs.

back-end. During simulation, TaskSim interfaces Nanos++, a runtime system library of OmpSs, and exposes the simulated architecture as a pool of processing elements. The runtime system performs the task creation, dependence tracking and scheduling, while TaskSim replays the trace of a given task on the core assigned by the runtime system. To enable the development of the work presented in this thesis, the following improvements to the TaskSim simulator have been implemented:

- The possibility to forward information from the runtime system level to the simulated hardware. This feature introduces a special instruction that carries the runtime-provided payload to any simulated component. It enables simulating the overhead introduced by the hardware setup performed by the runtime system. As an alternative, existing instructions are extended to carry special flags, such as the issuing task's name.

- A complete refactoring of the cache replacement policies implementation to allow dynamic switching of the policies during the execution. In addition, the support for set dueling is added by allowing each cache set to follow a distinct replacement policy and by implementing the mechanism for performance tracking and policy switching.

- Support for execution of atomic instructions in the simulated hardware. The unmodified code relies on the runtime system events to guard the atomic access using a lock, which introduces a significant overhead.

The energy consumption and microprocessor area are evaluated using the McPAT models [115]. McPAT is an integrated power, area and timing simulator for multi-core architectures build on top of CACTI [17, 126, 127]. It models various processor components, such as cores, including the functional units, caches, on-chip interconnections and memory controllers. The accuracy of the built-in models is improved by incorporating the changes suggested by Xi et al. [178]. The processor simulated in this thesis uses a 22 nm transistor technology with the voltage of 1.2 V and the default clock gating scheme. The hardware structures proposed in this thesis are modeled with CACTI 7 [17] using the same process technology. TaskSim is extended with appropriate counters to record the necessary statistics corresponding to these components.

3.1.2 Baseline Architecture

The baseline architecture used for the evaluation in this thesis is based on a multi-core processor connected to the main memory. The cores follow a simple model of a superscalar out-of-order processor with a detailed three-level cache hierarchy. Each core has private L1 and L2 cache, while the L3 cache is shared among all cores. The L2 and the L3 caches are connected via a simple interconnection network. Main memory is simulated using a constant-latency model. The first proposal, which introduces cache insertion policies, is targeted to small processors with slow memories. The second proposal, which performs reductions in the cache hierarchy, targets HPC machines based on Intel Xeon processor having a high bandwidth interface to the main memory. The third proposal, which introduces criticality-driven prioritization of memory requests, is evaluated on a multi-core processor with smaller caches. In addition, the memory

Table 3.1: Parameters of the simulated systems for each proposal.

	Replacement	Reductions	Prioritization
CPU	out-of-order superscalar cores, 128-entry ROB, frequency 2.4 GHz, issue width 4		
Cores	4	16	16
Caches	64 B line, non-inclusive, write-back, write-allocate, split I/D, LRU replacement		
L1	32 KB, 4-way, 4 cycles	32 KB, 8-way, 4 cycles	16 KB, 8-way, 4 cycles
L2	256 KB, 8-way, 10 cycles	256 KB, 8-way, 12 cycles	64 KB, 16-way, 13 cycles
L3	8 MB, 16-way, 24 cycles	32 MB, 16-way, 36 cycles	16 MB, 16-way, 68 cycles
Memory	200 ns latency 9.6 GB/s bandwidth	126 ns latency 85 GB/s bandwidth	120 ns latency 17.2 GB/s bandwidth

bandwidth is tuned to a value that results in a realistic number of in-flight memory requests. A summary of the relevant configuration parameters is presented in Table 3.1.

3.1.3 Environment

The tracing of benchmarks is performed on different systems. Traces used in the evaluation of the first and the third proposal are generated on an IBM dx360 M4 node equipped with two 8-core E5–2670 SandyBridge-EP processors running at 2.6GHz with a 20MB LLC and 32GB main memory. The node runs SUSE Linux Enterprise Server 11 SP3. Tracing for the evaluation of the second proposal is performed on a machine with a dual-core Intel Core i7-5600U running at 2.60GHz with a 4MB LLC and 16GB main memory. The operating system running on this machine is openSUSE Leap 42.3. Benchmarks are compiled using Mercurium [18] source-to-source compiler, which translates the OmpSs annotations to the function calls defined in Nanos++ [19] runtime system library. GCC is used as a back-end compiler for generating the final code. All codes are compiled using the -O3 optimization flag. To enable proper capturing of function boundaries, the benchmarks are compiled with `-fno-optimize-sibling-calls` flag. The versions of the compilers and the runtime system used in the evaluation of the first proposal are as follows: Mercurium 1.99.0, Nanos++ 0.7a and GCC 4.9.1. The last two proposals use the following versions: Mercurium 2.1.0, Nanos++ 0.15a and GCC 7.2.0.

3.2 Benchmarks

The benchmarks used for the evaluation of the proposals in this thesis are selected among HPC applications and kernels to cover a wide range of algorithms used in scientific codes. All the codes are written using the OmpSs programming model using tasks-based parallelization. Most of the benchmarks are chosen from larger collections, such as PARSECSs [40], an OmpSs port of PARSEC Benchmark suite [25], and BSC Application Repository [21]. Kernels dot product [54], stencil histogram [54] and array scan [53] are developed by the author of this thesis. The remaining of this section describe the benchmarks corresponding to each proposal, including the proposal-specific changes introduced in each code, the input parameters used for tracing and the selected code properties relevant to each contribution.

Table 3.2: Benchmarks used to evaluate the proposal about cache replacement policies.

Benchmark	Description	Input Parameters
Conjugate Gradient (CG) [90, 159]	An iterative method for solving large systems of linear equations that have form $Ax = b$.	matrix qa8fm [50], 16 blocks, 97 iterations
Facesim [40]	Intel RMS workload which takes a model of a human face and a time sequence of muscle activation and computes a visually realistic animation of the modeled face.	simlarge: 80,598 particles, 372,126 tetrahedra, 1 frame
Ferret [40]	A content-based similarity search of feature-rich data such as audio, images, video and 3D shapes.	simlarge: 34,973-image database, 256 queries, find top 10 images.
Specfem3D	Simulates global and regional (continental-scale) seismic wave propagation.	147,645 global points, 2160-element mesh, 125 Gauss-Lobatto-Legendre integration points per element
Stap [113]	Space-time adaptive processing applied on moving target indication by an airborne radar.	136,808 stimuli

The benchmarks used for the evaluation of the first proposal are shown in Table 3.2, including a description of each code and the input parameters used for tracing. These benchmarks do not require any source-code modifications.

For the evaluation of the second proposal, the eligible codes are those that have reduction-like computation, such as histograms, matrix multiplications, etc. Table 3.3 presents the description of a set of benchmarks having these properties. The benchmarks are split into two groups defined in Section 2.3.1 depending on the size of the reduction variable. The source-code of the benchmarks is modified by annotating the reduction variable and the corresponding reduction operator in each reduction task. In addition, updates to the reduction variable are made to be direct (i.e. not using privatization) and *atomic* guards are removed, as the atomicity is provided by the hardware support for reductions. The benchmark versions used to evaluate the baseline solutions (i.e. privatization and atomics) are identical to the original. Table 3.4 specifies the input parameters used for tracing and selected properties of reduction tasks and reduction operations for each code. The workload size of reduction task corresponds to the reduction variable size (out) and size of the data that is being reduced (in).

Table 3.3: Description of the benchmarks used to evaluate the proposal about reductions.

	Benchmark	Short Name	Description
Scalar Reductions	Dot Product	DotP	Calculates the sum of the products of the corresponding entries of the two sequences of numbers.
	KnightsTour	KT	Calculates a number of sequences of moves of a knight on a chessboard such that the knight visits every square exactly once.
	NBinaryWords	NB	Recursively calculates how many arrays of length N can be constructed from a dictionary of 2 letters.
	NQueens	NQ	Determines in how many ways one could place N queens on a $N \times N$ chessboard so that none of the queens attack each other.
	PowerSet	PS	Calculates the size of the power set of a set with N elements. Power set is defined as a set of all subsets of a given set.
	Vector Reduction	VectR	Calculates a sum of elements in a vector.
Vector Reductions	2D Convolution	2DC	Calculates a convolution of two functions represented with two-dimensional matrices.
	Conjugate Gradient [90, 159]	CG	An iterative method for solving large systems of linear equations that have form $Ax = b$.
	Dense Matrix Matrix Multiply	DGEMM	Multiplies two dense square matrices.
	Dense Matrix Vector Multiply	DGEMV	Multiplies a dense square matrix with a vector.
	2D Explicit Hydrodynamics Fragment [120]	EHF	A kernel from a 2D explicit hydrodynamics code.
	Stencil Histogram	Hist	Calculates a histogram of weighted averages using a 3D 27-point stencil over a $N \times N \times N$ cube represented by a dense 3D matrix of floating point numbers.
	Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics [99]	LULESH	Solves the single-material Sedov blast wave problem.
	Molecular Dynamics	MD	A simulation using a Lennard-Jones potential. The forces between particles in a 3D space are iteratively calculated and their positions integrated over time.
	N-body Simulation	NBody	Simulates the positions of N bodies trough the time taking into account the gravitational forces among them.
	1D Particle in Cell [120]	PIC	An kernel from a 1D particle-in-cell code used for kinetic simulations in plasma physics and astrophysics.
	Sparse Matrix Vector Multiply	SpMV	Multiplies a sparse matrix with a vector.

Table 3.4: Input parameters and the properties of the benchmarks used for the evaluation of reductions.

The Data/Op column shows the type of the reduction variable and the reduction operator. The last two columns show the dynamic ratio of reduction instructions compared to the total number of executed instructions and the time spent on reduction instructions compared to the overall execution time, respectively

	Benchmark	Input Parameters	Reduction task workload size	Data Op	Reduction instruction ratio	
					Count	Time
Scalar Reductions	DotP	256K elem. 100 iterations	in: 2MB out: 8B	FP ADD	8.96 %	7.95 %
	KT	5×5 chessboard	in: 304K elem. out: 4B	INT ADD	0.74 %	1.88 %
	NB	word length: 24	in: 2^{24} elem. out: 4B	INT ADD	14.09 %	9.50 %
	NQ	12 queens on a 12×12 chessboard	in: 12! elem. out: 4B	INT ADD	0.45 %	0.30 %
	PS	set size: 24 elements	in: 2^{24} elem. out: 4B	INT ADD	14.37 %	11.05 %
	VectR	256K elem. 50 iterations	in: 2MB out 8B	FP ADD	22.91 %	43.24 %
Vector Reductions	2DC	image 1024×1024 pixels, stencil 16×16	in: 16MB out: 1MB	FP ADD	9.87 %	28.09 %
	CG	matrix qa8fm [50], 16 blocks, 97 iterations	in: 529.5KB out: 516KB	FP ADD	5.82 %	1.82 %
	DGEMM	matrix 1024×1024, block 64×64	in: 16MB out: 8MB	FP ADD	1.28 %	0.10 %
	DGEMV	matrix 2048×2048, block 128×128	in: 32MB out: 16KB	FP ADD	14.21 %	16.48 %
	EHF	array 16×64K elem.	in: 9MB out: 6MB	FP ADD	0.89 %	11.14 %
	Hist	input: 4MB, 512K bins, 27-point stencil	in: 4MB out: 2MB	INT ADD	1.31 %	11.77 %
	LULESH	cube 20^3, 11 regions, 10 iterations	in: 1500KB out: 187.5KB	FP ADD	0.35 %	3.00 %
	MD	2k atoms, periodic space, stretch phase change	in: 326KB out: 163KB	FP ADD	2.34 %	0.91 %
	NBody	4096 bodies, 10 iterations	in: 24MB out 96KB	FP ADD	5.88 %	12.32 %
	PIC	array 128K elem, 8K histogram, 1000 iter.	in: 6.5MB out: 64KB	FP ADD	17.87 %	9.99 %
	SpMV	matrix bcsstk32 [50]	in: 12.9MB out: 357KB	FP ADD	10.63 %	15.88 %

3.2 Benchmarks

The evaluation of the third proposal requires benchmarks with more than one task type and that instances of different task types execute simultaneously. The benchmarks that fulfill these conditions are listed in Table 3.5 together with their descriptions. Table 3.6 provides the input parameters used for tracing and certain measures regarding the amount of critical tasks and memory requests. The source-code of each benchmark is modified by annotating critical task types, which are determined by a static analysis of the critical path. This analysis is performed using TaskSim set up to run in a burst-mode which replays only recorded task duration instead of simulating the instructions. For a given task type, we evaluate how reducing the task duration by a certain amount impacts the overall execution time. This process is repeated for all task types. Tasks with largest impact on the overall performance are defined as critical tasks. In some benchmarks, several task types are selected as critical. In the case of fluidanimate, the four critical task types represent the calls to the same function from different locations in the code. The critical tasks in SMI are `gemm` tasks, which perform matrix-matrix multiplications. In the case of LU decomposition, the critical tasks are `fwd` and `bmod`. The second to last column in the table represents the number of critical task instances compared to the total number of tasks. The last column shows the percentage of all memory request that are flagged as critical.

Table 3.5: Benchmarks used to evaluate the proposal about memory request prioritization.

Benchmark	Short Name	Description
Array scan	scan	A kernel that performs a scanning access to an input array.
Bodytrack	body	An application from the computer vision domain that tracks a pose of a human body in 3 dimensions from a sequence of images.
Cholesky	chol	Decomposes of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose.
Dedup	dedup	Compresses of a data stream with a combination of global compression and local compression to achieve high compression ratios.
Ferret	ferret	A content-based similarity search of feature-rich data such as audio, images, video and 3D shapes.
Fluidanimate	fluid	An Intel RMS application that uses an extension of the smoothed particle hydrodynamics method to simulate an incompressible fluid for interactive animation purposes.
QR Decomposition	QR	Decomposes of a matrix into a product of an orthogonal matrix Q and an upper triangular matrix R.
Symmetric Matrix Inverse	SMI	Calculates of an inverse of a symmetric matrix.
Sparse LU Decomposition	LU	Factorizes a sparse matrix as the product of a lower triangular matrix and an upper triangular matrix.

Table 3.6: Input parameters and properties of the benchmarks used for the evaluation of the proposal about memory request prioritization.

Benchmark	Input Parameters	Critical / Total Task Types	Critical (% of total)	
			tasks	requests
scan	174 arrays of 256 KB; in total 68.5 MB.	1 / 2	13.8	13.8
body	simlarge: 4 cameras, 4 frames, 4000 particles, 5 annealing layers.	1 / 6	47.0	66.0
chol	16×16 blocks of 256×256 elements; total matrix size 4096×4096 elements.	1 / 6	3.2	27.6
dedup	native: 672 MB data.	1 / 5	49.7	77.8
ferret	simlarge: database of 34,973 images, 256 queries, find top 10 images.	1 / 7	16.6	6.1
fluid	native: 5 frames, 500,000 particles	4 / 23	21.8	45.5
QR	16×16 blocks of 512×512 elements; total matrix size 8192×8192 elements.	1 / 7	42.0	90.2
SMI	8×8 blocks of 1024×1024 elements; total matrix size 8192×8192 elements.	3 / 20	18.9	78.6
LU	12 blocks of 512×512 elements	2 / 5	15.6	93.2

3.3 Metrics

The evaluation of the proposals in this thesis is performed by analyzing several performance metrics. The execution time is obtained from TaskSim simulations in terms of cycles and is converted to seconds by taking into account the frequency of the simulated processor. TaskSim also provides metrics that measure cache behavior, such as cache misses. Cache misses are combined with the instruction count to form a compound metric of misses per kilo instructions (MPKI) using the following formula:

$$MPKI = \frac{Cache\ misses}{\frac{Total\ executed\ instructions}{1000}} \tag{3.1}$$

The performance in terms of power is obtained from McPAT, and is used to compute the energy-delay product (EDP) using the following formula:

$$EDP = Power \times Execution\ time^2 = Energy \times Execution\ time \tag{3.2}$$

The comparison of the performance relative to the baseline architectures is done by calculating speedups in case of execution time or improvement as a percentage of the reference value in case of other metrics using formulas 3.3 and 3.4, respectively. In some cases, the metrics are normalized to the baseline values using Formula 3.5.

$$Speedup = \frac{Execution\ time_{baseline}}{Execution\ time_{proposal}} \tag{3.3}$$

$$Performance\ improvement = 100\% \times \frac{Performance_{baseline} - Performance_{proposal}}{Performance_{baseline}} \tag{3.4}$$

$$Metric_{normalized} = \frac{Metric_{proposal}}{Metric_{baseline}}. \tag{3.5}$$

To compare the proposals in the general case, the metric values corresponding to different benchmark are aggregated to provide a single measure of performance. For the metrics defined as ratios, such as speedup, geometric mean is used (Formula 3.6). Metrics that represent absolute values are averaged using arithmetic mean (Formula 3.7).

$$Geometric\ mean = \sqrt[n]{value_1 \times value_2 \times \cdots \times value_n} \tag{3.6}$$

$$Arithmetic\ mean = \frac{value_1 + value_2 + \cdots + value_n}{n} \tag{3.7}$$

Chapter 4

Last-Level Cache Insertion Policies

This chapter explores the utility of the runtime system level information in the design of cache replacement policies. Of a special interest is a shared cache as it comes with additional challenges compared to private caches, such as mixed access patterns coming from different cores. We propose two insertion policies based on re-reference intervals. Each policy targets a different challenge arising in the shared last-level caches. The first policy, Task-Type-aware Insertion Policy (TTIP), is a dynamic insertion policy that assigns re-reference interval to a cache line on insertion depending on the task type that issued the corresponding request. The second, data-type aware insertion policy (DTIP), is based on the observation that the access pattern of accesses to task data dependencies often depend on the dependency type. In the context of the used programming model, the dependency types correspond to whether the data is consumed (*in*), produced (*out*) or updated (*inout*). DTIP is similar to TTIP in the fact that it discriminates the cache lines when assigning a re-reference interval. Specifically, DTIP uses the dependency type of the variable the memory request refers to. Both DTIP and TTIP are guided by the runtime system library by accessing simple hardware structures inside the LLC that define the insertion policy behavior.

This chapter makes the following contributions:

- A dynamic insertion policy based on task types that is able to automatically identify the best configuration during the execution time.
- A static insertion policy based on the data dependency types.
- Hardware components that control the behavior of the insertion policy and are exposed to the runtime system library via memory-mapped registers.

- Runtime system extensions that dynamically determine the best-performing configuration for each task type and control the insertion policy in the hardware using the previously-described interface.

4.1 Challenges in the Design of Replacement Policies for Shared Caches

Memory access pattern is the most important factor that drives the design of cache replacement policies. Unfortunately, the access patterns observed at the last-level cache are rather complex. The complexity is caused by several factors. The locality of accesses is hidden by the private caches which makes it harder to predict the access pattern only by an analysis of the application code. In addition, accesses coming from the different cores are interleaved at the LLC, which further complicates their analysis. However, by discriminating the memory access by the issuing core, it is possible to separately observe the access patterns corresponding to each core. Furthermore, differentiating the accesses from a single core by the variable they belong to can make it easier to predict the access pattern at the LLC.

Simple cache replacement policies such as LRU, NRU, DRRIP do not discriminate the memory accesses. As a result, this can lead to sub-optimal cache performance due to workload trashing. More recent designs consider the access patterns and are able to achieve better hit ratios. For example, SHiP relies on the access signature to estimate to which code segment it belongs to. However, such heuristic-based method is not precise at identifying the variable accessed with a given memory request. The semantics of the task-based parallel programming models, such as OpenMP and OmpSs, can provide an accurate identification of the relevant variables (i.e., task data dependencies) and code segments (i.e., tasks).

To illustrate the design opportunities provided by the accurate knowledge of the access patterns at the LLC level, we use the example in Figure 4.1. It shows a simple case of a combined memory access pattern consisting of one trashing and one cache-friendly access pattern denoted with letters (A, B, C and D) and numbers (1 and 2), respectively. The two access patterns can correspond to different threads or different memory regions within the same thread. The top of the figure shows the order of accesses throughout the time for two address ranges that map to the same four-entry cache set. Accesses to the cache friendly and trashing region are colored in green and blue respectively. The bottom of the figure shows (i) the outcome of the above accesses to the cache set following two replacement policies as

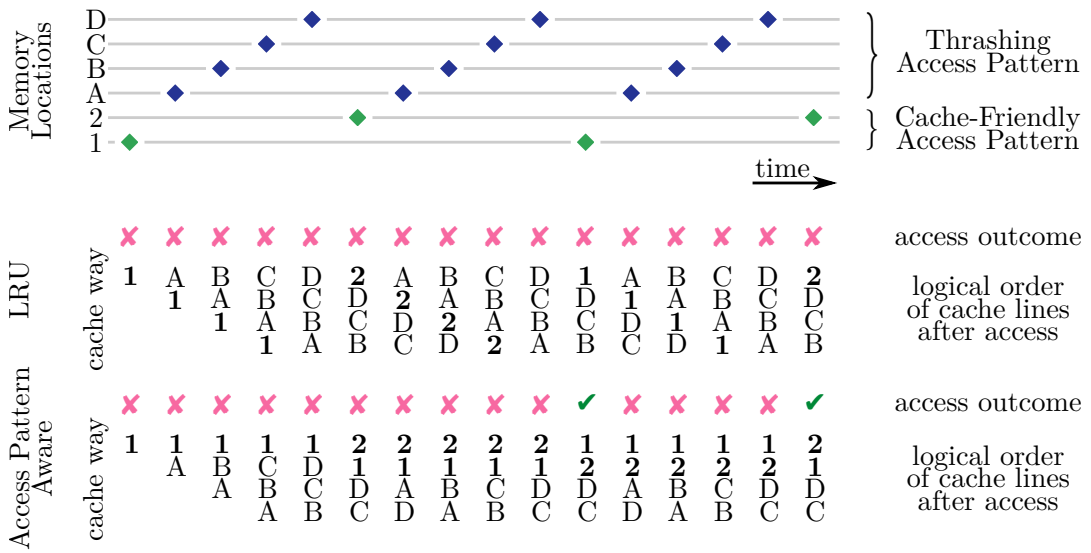


Figure 4.1: Comparison of LRU and access-pattern aware insertion.
LRU inserts new line at the top of the recency stack, while the lines are evicted from the bottom of the stack. The access-aware policy inserts cache friendly lines at the top of the recency stack, while trashing lines are inserted in the middle.

ticks for hits and crosses for misses, and (ii) the state of the cache set after each access. In the case of LRU, the thrashing access pattern pushes to the bottom of the recency stack the lines belonging to the cache-friendly region, which results in their eviction. Therefore, all of the accesses result in a cache miss. The second policy inserts the trashing lines in the middle of the recency stack and uses the same promotion mechanism as LRU. This approach preserves the position of the lines corresponding to the cache-friendly region. Consequently, the third and the fourth accesses to that region result in cache hits.

The conclusion of this analysis is that it is possible to achieve a better cache performance by applying different replacement strategies to accesses belonging to distinct memory regions. In the context of this work, the differentiation can be done on a task level as well as on a task data dependency level. The previous example shows a deterministic insertion policy. To perform finer adjustments, it is possible to employ a probabilistic insertion policy similar to BRRIP. Probabilistic insertion uses a probability parameter that controls how often the lines are inserted into the bottom or middle of the recency stack. In addition, by assigning a different probability to tasks, it is possible to indirectly control the cache space allocated to each task. In summary, the use of the application-level knowledge can help with the design of a better-performing cache replacement policy.

4.2 Runtime-Assisted LLC placement policies

An important factor that affects the LLC performance is memory access pattern. Runtime systems that support task-based data-flow programming models have information about task types and task data-dependencies of the application. We aim to utilize this information at the hardware level to improve the LLC performance by optimizing its insertion policy.

4.2.1 Task Type Aware Probabilistic Insertion

TTIP is a runtime-guided task type-aware last-level cache insertion policy. It uses a notion of re-reference intervals to maintain the logical order of cache lines, which is used to select the eviction candidates. TTIP targets parallel applications written in task-based programming models, such as OpenMP and OmpSs. TTIP is a dynamic policy that automatically selects the best-performing policy for each task type. A runtime system library is responsible for the training process and configuring the hardware to use the selected policy. The last-level cache is extended to support the manual selection of an insertion policy for each task type, which is performed by the runtime system using memory-mapped registers.

TTIP relies on the following extensions:

- Runtime system extensions to allow performance tracking of different insertion policy configurations.
- A training algorithm implemented in the runtime system that monitors the cache performance and selects the optimal configuration for a given time interval.
- Microarchitectural extensions in the LLC that allow the runtime system to control the insertion policy.

TTIP is a policy based on re-reference intervals (RRI) and uses a two-bit representation of the RRI. RRIP policies usually observe three special RRI values: (i) *immediate*, (ii) *long* and (iii) *distant*, which correspond to the RRI values of `00`, `10` and `11`, respectively. In the context of the work presented in this section, the insertion policy determines the assigned re-reference interval (RRI) to each fetched line. Specifically, the insertion policy is defined by a probability to assign a *long* RRI to a cache line. For example, the probability of 0.2 means that one in five lines is assigned a *long* RRI value, while the other four are assigned a *distant* RRI value.

TTIP uses an independent probability for each task type. During the run-time, the probabilities are assigned according currently running task type. That is, write-backs of

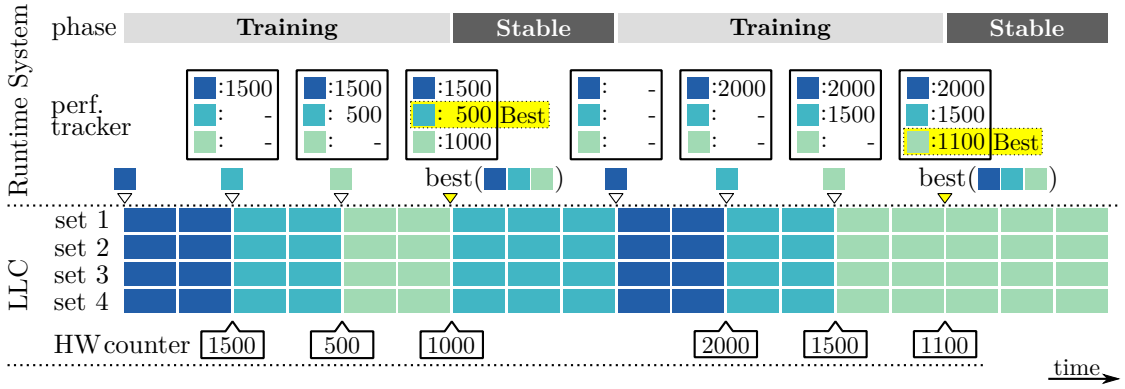


Figure 4.2: TTIP probability training process for a certain task type.
Colors represent different policy configurations (probabilities). Each rectangle in the LLC sets represent a task instance and is colored according to the configuration used for that instance. Hardware counter records the number of misses.

previous task's cache lines are handled as if they belong to the currently running task. To determine the best probability per task type, TTIP employs a sampling-based training mechanism, as displayed in Figure 4.2. The figure shows a snapshot of an execution of a parallel code on a system with a 4-set LLC implementing TTIP. During the training phase, the runtime system configures the LLC's insertion policy to use probabilities from a predefined pool $\mathcal{P}$. The policy is switched at a task boundary and each policy configuration is used for K task instances. Probabilities are selected sequentially from the pool until all probabilities are evaluated. Upon switching to a new policy, the runtime system samples the values of selected hardware counters to get an estimation of the cache performance during the previous period, such as the number of cache misses. When all configurations are exhausted, the best-performing policy is calculated by comparing the recorded cache performance metrics. In total, $K \times |\mathcal{P}|$ task instances are used for training during one training phase for one task type. Once the training phase is complete, the stable phase begins. The LLC is configured to run the best-performing insertion policy during a period that corresponds to N task instances. In Figure 4.2, the values that correspond to K and N are 2 and 3, respectively. Once the N task instances are executed in the stable phase, the training process continues from the beginning, switching again to the training phase. As a consequence, TTIP is able to adapt to the changes in application behavior in terms of memory access patterns. The described training process is performed independently for all the task types.

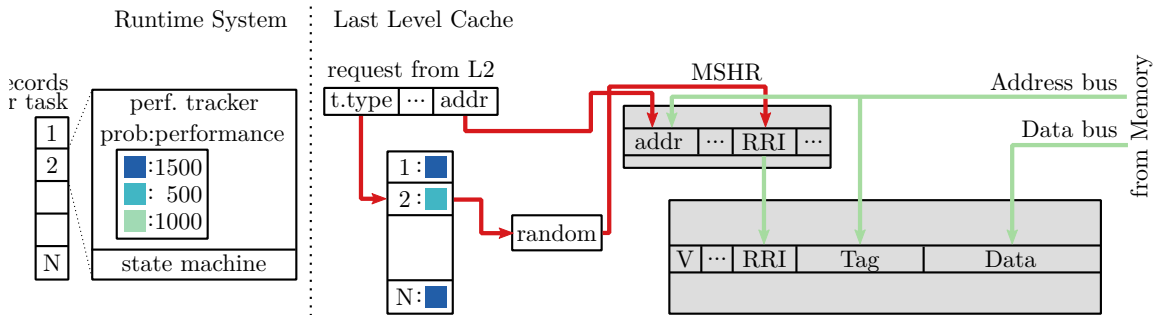


Figure 4.3: Runtime and microarchitectrual extensions for TTIP.

4.2.1.1 Hardware Extensions

This section describes hardware components necessary to implement TTIP, which are shown in Figure 4.3. To be able to use a different insertion probability for each task type, TTIP employs a small and fast hardware structure in the LLC that maps the task type ID to the assigned probability. It is designed as an SRAM memory containing the probabilities and is addressed by the task type ID. The mapping table is accessed on a cache miss and the returned probability is fed to a random number generator, which returns one of the two RRI values, *long* or *distant*. The selected RRI value is written into the Miss Status Handling Register (MSHR) entry corresponding to the request. When a memory request resulting from a miss is served, the cache line is inserted into the cache tag and data arrays. The RRI value to be assigned to the newly inserted line is fetched from the MSHR. The mapping table is modified by the runtime system library using memory-mapped registers, which allows to maintain the original ISA. To track the performance of each insertion probability, TTIP uses one hardware counter per task type that records the number of requests from each task type that result in a cache miss. The counters are exposed to the runtime as a set of memory-mapped registers.

4.2.1.2 Runtime System Extensions

This section describes the required extensions to the runtime system library, as shown on the left-hand side in Figure 4.3. The runtime system extensions can be classified into the two components that control the dynamic insertion policy: (i) a module for performance tracking and (ii) a module that selects which policy is used for each task type at a given moment in time. The performance tracker holds the cache performance statistics per probability, for each task type. This structure is updated every time the probability for a certain task is switched. The policy selector consists of a set of state machines that hold the current phase (i.e., training or stable), currently selected policy and a set of policies not yet evaluated during the training

phase. A simple algorithm selects the next policy depending on the phase. During the training phase, policy selector traverses policy configurations from the predefined set of probabilities $\mathcal{P}$, while at the beginning of the stable phase, the best-performing policy is selected.

These actions are performed on the task boundaries, inside the runtime system functions. Once a task is scheduled to execute and before its user code starts executing, the policy selector decides which probability to use for that task instance and writes it in the probability table at the position corresponding to the task type ID of the said task. At the end of the execution of a task, the number of misses produced by that task in the LLC is read and stored in the performance tracker module of the runtime system.

4.2.2 Dependency Type Aware Insertion

This section describes a dependency type-aware insertion policy (DTIP) for a shared last-level cache. As the previously described policy, DTIP also targets parallel codes written in data-flow task-based programming models. Its design is based on two observations. (i) Input dependencies are read-only data accessed by the currently-running task instance. (ii) Output dependencies are generated by a task in order to be consumed by its successor tasks. Therefore, it may be beneficial to insert cache lines belonging to outputs in higher positions of the recency stack, thus giving them more chances to stay in the cache until the moment they are required by the consumer task. A similar reasoning applies to dependencies denoted as *inouts*, as they are also inputs of future tasks. Non-dependencies are the local variables of the current task and the global variables that are not specified as task dependencies. In certain evaluated benchmarks, like CG, they are predominantly accesses to large global variables that have streaming-like access patterns. In other benchmarks, where this is not true, decisions that we make for non-dependencies do not harm the performance.

DTIP assigns a RRI value to the new cache line depending on the dependency type of the variable that the cache line corresponds to. The policy configuration can be formally defined as a 4-tuple (RRI_{input}, RRI_{output}, RRI_{inout}, $RRI_{non-dependency}$), where RRI_d is a RRI assigned to lines corresponding to the dependency type d and $RRI \in \{immediate, long, distant\}$. For simplicity, the following naming conventions are adopted:

- Dependency types and RRIs are annotated in the short form, i.e., *in*, *out*, *inout* and *non-dep* for dependency types and `I`, `L` and `D` for RRIs.
- Configurations are designated with a four-letter acronym, where each letter corresponds to the RRI assigned to *in*, *out*, *inout* and *non-dep*, in that order. For example, a configuration

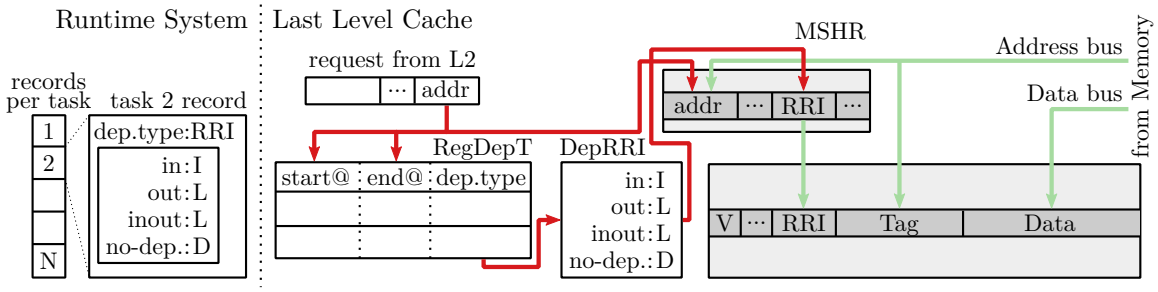


Figure 4.4: Runtime and microarchitectrual extensions for DTIP.

that assigns long RRI to *in* and *out*, distant RRI to *inout* and immediate RRI to *non-dep*, formally defined as (L, L, D, I), is designated as LLDI.

- Configurations where one or more mappings do not matter mark the corresponding positions with x, e.g., LxxI.

4.2.2.1 Hardware Extensions

To identify a re-reference interval (RRI) for a cache line on a cache fill, DTIP uses two simple hardware structures, as shown on Figure 4.4. RegDepT is a table that maps a memory region to the dependency type associated with that region. A memory region is defined by its starting and ending physical address, while a dependency type is encoded with two bits. RegDepT is designed similarly to a content-addressable memory (CAM). The address of a cache reference that resulted in a miss is fed to the structure where it is simultaneously compared with all the start and end addresses. The output of this action is an entry i that fulfills the following condition: $startAddr_i >= sourceAddr >= endAddr_i$. In the case of a no-match, the returned value encode a non-dependency. The second component is a DepRRI, a simple array that maps a dependency type to the associated RRI. It is designed as an array of four entries addressed by the two-bit representation of a dependency type.

The RegDepT and DepRRI are accessed on every occurrence of a cache miss in the LLC to determine the RRI for the missing line. First, the address of the missing line is fed to the RegDepT which returns its dependency type. Then, the obtained dependency type is used to address DepRRI and the resulting RRI is stored in the corresponding entry in the MSHR. This process done in parallel with issuing the request to the main memory and does not introduce any additional latency. When a request for the missing line is served by the memory, the RRI is read from the MSHR and the cache line is updated accordingly. Figure 4.4 illustrates the described actions on a cache miss and cache fill using dark red and light green arrows, respectively.

The described hardware structures are updated by the runtime system on every task switch. If the mapping information corresponding to the previous and current task are the same, the runtime system does not perform the redundant update action. As RegDepT is physically addressed, the translation of new entries is automatically performed by the hardware. In case of a non-continuous mapping to physical address space, multiple entries are created for a single variable. There may be several tasks using the same region at the same time. However, this cannot result in a conflicting mapping of one region to two dependency types, as the runtime system inherently guarantees that a region accessed by more than one task is always accessed in the same manner (i.e., as *in*, *out* or *inout* dependency). The runtime system does not prevent a region that is a non-dependency for a certain task to be accessed as other type of dependency by another co-running task. However, this scenario is not covered by this proposal as it is considered an invalid programming pattern in the context of a data-flow task-based programming model.

The RegDepT and DepRRI are implemented in the LLC and are exposed to the software layer as a set of memory-mapped registers. This interface is supported in most modern architectures and does not require changes to the processor's ISA.

4.2.2.2 Runtime System Extensions.

DTIP relies on several runtime system extension, as shown on the left-hand side of Figure 4.4. The extensions consist of: (i) a data structure that holds the mapping of the dependency type to the associated re-reference interval (RRI) for each task type (ii) a function that updates the hardware structures described in Section 4.2.2.1, taking into account the data dependencies of the current task already present in the task-dependency graph and the dependency type-to-RRI mapping described above. When a task is scheduled to execute on a core, the runtime system updates the RegDepT and DepRRI with the information corresponding to the new task by issuing store instructions to the memory-mapped registers. If there are several consecutive dependency regions of the same type, the runtime may perform two optimizations to reduce the storage requirements in the mapping table. The first optimization merges the consecutive dependency regions of the same dependency type into one. The second does not insert the region if it already exists in the table, which happens if two or more tasks are sharing the same region. Since the mapping table is not readable by the runtime to simplify the hardware design, the runtime keeps a software copy of the mapping information.

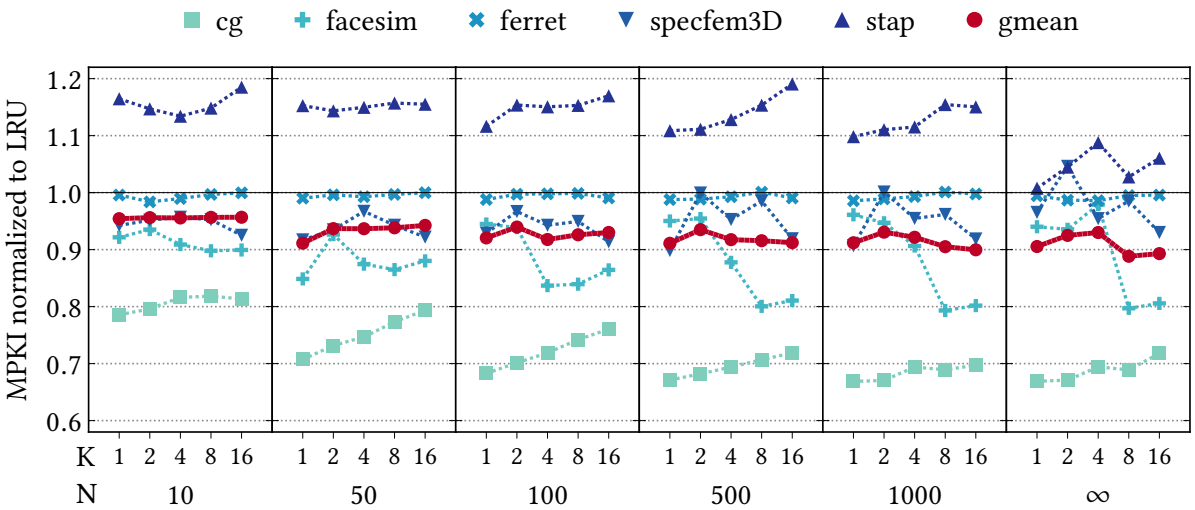


Figure 4.5: TTIP sensitivity to $N \in \{10, 50, 100, 500, 1000, \infty\}$ and $K \in \{1, 2, 4, 8, 16\}$

4.3 Design Space Exploration

This section explores the behavior of TTIP and DTIP depending on their configuration parameters. The result of these analyses are two configurations used in the evaluation of both policies.

4.3.1 TTIP Parameters Space Exploration

TTIP's performance depends on two parameters, K and N, which are described in Section 4.2.1. These parameters determine how many task instances per probability are used in training, and how many instances for running with the best probability in the stable phase, respectively. We explore the set of configurations (N, K) where $N \in \{10, 50, 100, 500, 1000, \infty\}$ and $K \in \{1, 2, 4, 8, 16\}$. Configurations where $N = \infty$ have only one training phase which is followed by one stable phase that lasts until the end of execution. Intuitively, choosing a larger K offers better precision by having more time to evaluate one probability. However, too large K can hurt the overall performance if certain probabilities perform badly. Configurations with larger N use the best probability for a longer period of time, but are less able to adapt to potential changes in application behavior. Using a smaller N can be bad for the final performance because a larger percentage of the execution is spent in the training phase.

Figure 4.5 shows the performance of TTIP in terms of MPKI depending on the choice of parameters K and N. The MPKI shown in this figure is normalized to the configuration using LRU replacement policy, which achieves MPKI 3.50, 3.40, 1.23, 1.45 and 1.44, respectively

for cg, facesim, ferret, specfem3D and stap. For most benchmarks except specfem3D we can observe a performance improvement as N increases. This is due to the fact that, in the majority of benchmarks, instances of the same task type have similar behavior. For cg, we can notice the trend of performance degradation when increasing K for a constant N. Similar behavior can be noticed for stap. Stap highly benefits from configurations where $N = \infty$ due to having a large number of task instances. Having many training phases in case of stap means repeatedly evaluating sub-optimal probabilities, thus hurting the overall performance. Ferret does not show significant sensitivity to K and N. Facesim obtains better performance with larger K due to having a lot of small task instances and, therefore, needing more instances per probability to properly evaluate the performance of each probability. The configuration that performs the best on average for all our benchmarks is $(N, K) = (\infty, 8)$, which we will use for further evaluation of TTIP in the remaining of the chapter.

4.3.2 DTIP Design Space Exploration

The behavior of DTIP depends on the four parameters that control the RRI assigned to each dependency type as defined in Section 4.2.2. To determine the impact of mapping different RRIs to dependency types, we perform an exhaustive design space exploration where we simulate all possible configurations. Number of different policy configurations per benchmark is $|\mathcal{I}|^{|\mathcal{D}|} = 3^4 = 81$, where $\mathcal{I}$ and $\mathcal{D}$ are sets defined in Section 4.2.2. For all five benchmarks, we run $81 \times 5 = 405$ simulations.

Figure 4.6 shows the results of the design space exploration. The results are grouped per benchmark and are shown in form of a heat map. The colors in a heat map correspond to the MPKI achieved by DTIP normalized to the MPKI obtained with LRU policy. The results show a significant sensitivity of cache performance on the selected policy for all benchmarks with the exception of ferret. CG and facesim obtain the best performance improvements when the *non-dep* cache lines are assigned a *distant* RRI. These benchmarks do not exhibit sensitivity to other DTIP parameters. Specfem3D, on the contrary, achieves the best performance for `DIxx` and `DLxx` configurations. In other words, it is important that *in* lines are assigned distant RRI, the *out* lines are assigned long or immediate RRI, while the configuration for other dependency types does not notably impact the performance. Similarly, stap performs best for `DIxx` configurations.

On average, the best performing policy is the one that assigns the *distant* RRI to *inputs* and *non-dependencies* and *long* or *immediate* to *outputs* and *inouts*. This is consistent with reasoning explained at the beginning of this section.

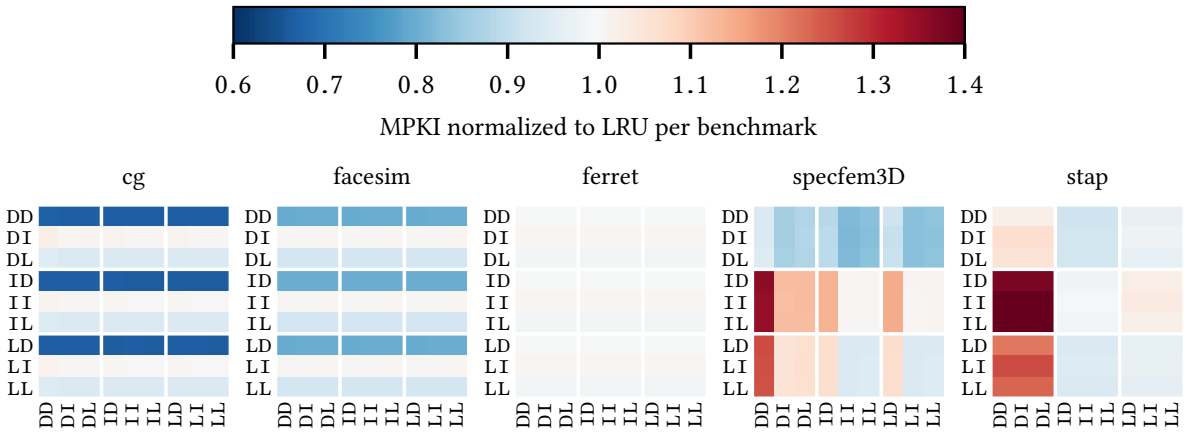


Figure 4.6: MPKI of DTIP normalized to LRU per benchmark.
Y-axis shows the insertion policy for *in* and *non-dep*. X-axis shows the insertion policy for *out* and *inout*. For example, a configuration `ABCD` is shown at the intersection of `x=BC` and `y=AD`. Lower, blue values correspond to better performance compared to LRU.

To further evaluate the performance of different insertion policies, we analyze the per-task behavior in terms of cache misses. For each DTIP configuration, we record a number of misses in the LLC produced by each task type. Figure 4.7 shows the results of this analysis for stap and specfem3D. The results are displayed only for a selected set of tasks, which are chosen among the largest tasks in terms of number of misses. Results show that tasks in stap observe the same performance trends for configurations `Ixxx` and Lxxx. In both of these configuration groups, the configurations with RRI_{out} = `I` and RRI_{out} = `L` observe similar number of misses, while the configurations with RRI_{out} = `D` achieve notably worse performance. This trend is not present for task `Calc.Filter` in the third set of configurations, `Dxxx`, as its performance does not suffer when RRI_{out} = `D`. In fact, this task achieves the best performance in the said set of configurations, and specifically for `DDxD`.

Contrary to stap, in the case of specfem3D, RRI_{inout} has a notable impact on performance, especially for task `update.disp.vel`. This task achieves best performance when RRI_{inout} takes values `I` or `L`. Task `scatter` observes the opposite behavior for configurations `Dxxx`, which clearly illustrates the fact that the optimal configuration for one task type may not be the optimal for another task type. Both `process.element` and `scatter` achieve a notably better performance for `Dxxx` compared to other configurations. This is also the case for `update.disp.vel`, but the performance improvements are minimal.

The conclusions of this analysis is that different task types achieve the best performance for different DTIP configurations, depending on their memory access patterns and dependency sizes. However, using the same, on average best-performing, configuration for all tasks achieves

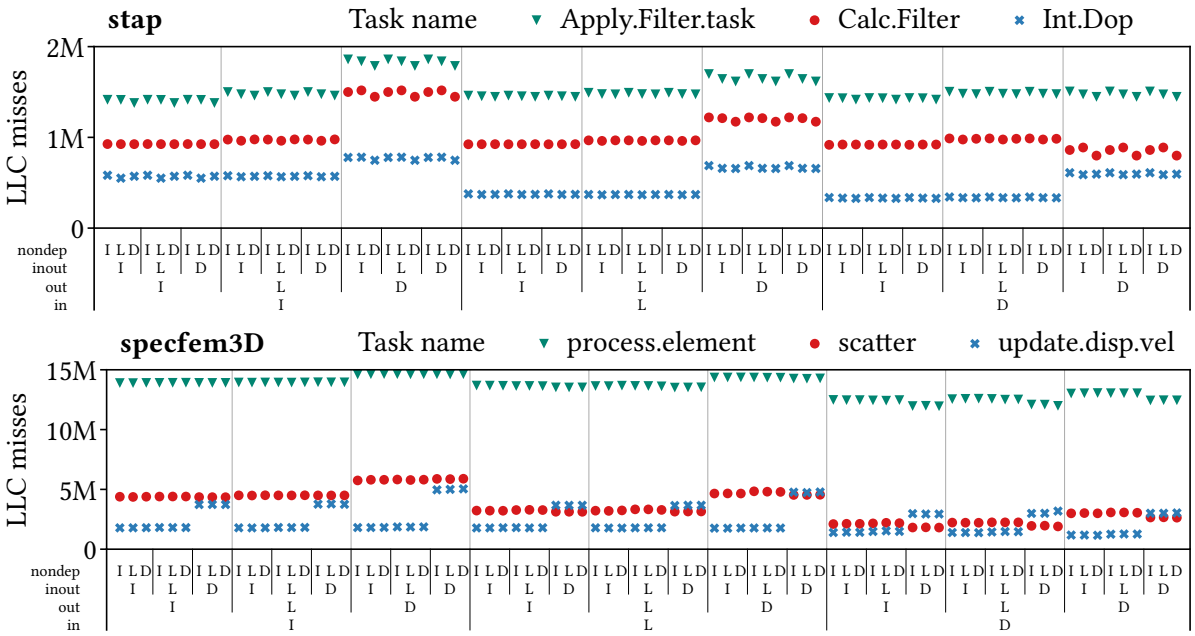


Figure 4.7: Cache performance per task type for different DTIP configurations.
Figure shows only selected tasks chosen as representatives of group of tasks observing the same behavior. Total number of task types for stap and specfem3D are 16 and 9, respectively. Tasks that have the same behavior as the shown tasks and tasks with small relative number of misses are omitted.

performance within 6.9% from the optimal performance for evaluated task types. Taking these findings into account, the evaluation of DTIP is performed using a single configuration. The exploration of per-task configurations might bring further performance improvements and is left for future work.

4.4 Evaluation

This section presents an evaluation of TTIP and DTIP. The experimental infrastructure is explained in Chapter 3. The benchmarks used for the evaluation and the input parameters are shown in Table 3.2. The LRU policy is selected as the baseline for the comparison of the relative performance. In addition, this section considers DRRIP as a state-of-the-art replacement policy, as well as the policies that DRRIP is combined of, SRRIP and BRRIP. BRRIP is configured to use the probability for assigning a *long* RRI of $\epsilon = 1/32$. DRRIP uses a SDM with 32 sets.

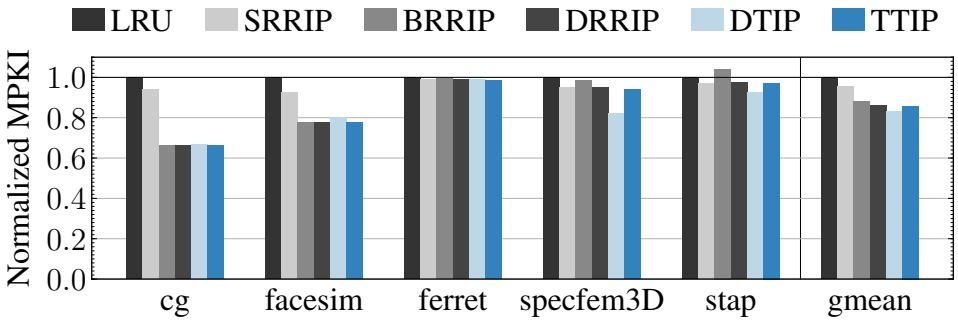


Figure 4.8: MPKI of TTIP and DTIP normalized to LRU

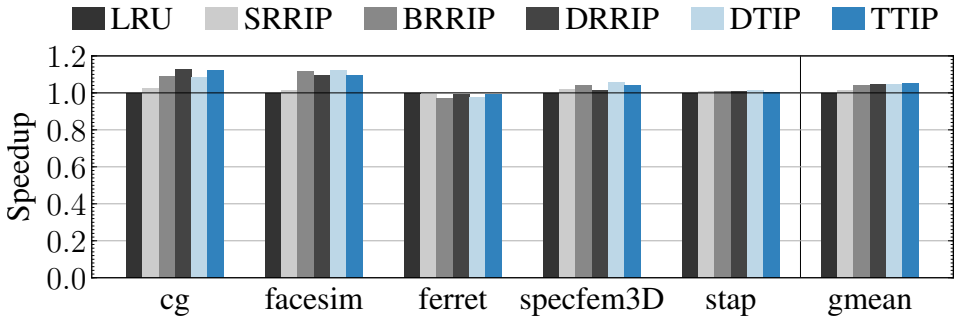


Figure 4.9: Speedup of TTIP and DTIP compared to LRU

4.4.1 Performance Results

Figure 4.9 compares TTIP and DTIP with LRU and state-of-the-art SRRIP, BRRIP and DRRIP in terms of MPKI and speedup normalized to LRU.

TTIP upgrades BRRIP by supporting multiple probability values and being able to optimize the probability per task type. It achieves up to 32.1% and on average 11.2% reduction in MPKI compared to LRU. The speedup over LRU is up to 12.3% and on average 5.1%. TTIP performs similarly as DRRIP, having 3.3% higher MPKI and being 0.8% faster than DRRIP. However, it does not need the hardware for Set Dueling, but instead uses a small mapping table described in Section 4.2.1.1 and whose cost is discussed in Section 4.4.2.

DTIP improves MPKI over LRU for up to 33.3% and on average 16.8%. The largest contribution of improvement in MPKI comes from specfem3D, where misses to *output* dependencies of the largest task are reduced by assigning *immediate* RRI to *outputs*. This decision does not significantly impact the number of misses to *inputs* and *non-dependencies*. DTIP is faster than LRU for up to 12.1% and on average 4.8%. Compared to SRRIP, which is another static RRIP policy, DTIP achieves up to 29.1% (12.8% on average) lower MPKI and performs up to 10.5% (3.7% on average) faster. The improvement over SRRIP comes from the fact that DTIP differentiates the cache lines by their data-dependency types. DTIP is able to

benefit from this information by assigning a more optimal RRI to cache lines so that different access patterns that collide in LLC have least possible negative effects on each other. DTIP reduces MPKI by 3.1% on average and is faster 0.3% than DRRIP.

Even though it shows higher MPKI than DTIP on average, TTIP achieves better execution time. The contributor to this effect is cg, where DTIP fails to achieve a speedup comparable to TTIP and DRRIP. The largest task type, which performs a matrix-vector multiplication, is the main source of MPKI improvement of DTIP over TTIP. However, three smaller, but still important tasks, show higher execution time with DTIP due to increased number of misses to *inputs* and *non-dependencies*. The improvement in execution time achieved in the largest task is not enough to compensate losses in three smaller tasks, because hits in the largest task are hidden by the unavoidable misses to the matrix.

4.4.2 Design Costs

To store the RRI information associated with each cache line, both TTIP and DTIP need $2n$ bits per cache set, the same as DRRIP, whereas LRU requires $n \log n$ bits per set, where n is the cache associativity. In the system evaluated in this work ($n = 16$), RRIP policies consume $2\times$ less space than LRU.

The mapping table required by TTIP has 4 entries, one for each core. Probabilities are stored with resolution of 6 bits, making the size of the structure $4 \times 6\,\text{bit} = 3\,\text{B}$. In addition, TTIP requires 4 hardware counter registers, each one being 32 bit long. The total additional hardware cost required by TTIP is $3\,\text{B} + 4 \times 32\,\text{bit} = 19\,\text{B}$. After each task instance, the runtime reads the corresponding hardware counter and potentially sets the new probability for the new task instance, which incurs overhead of few instructions. Calculating the best probability after the training period takes less than hundred instructions.

The mapping table for DTIP, RegDepT, contains 32 pairs of 48-bit physical addresses, thus providing each core with 8 entries, which is more than enough to cover the most demanding tasks in regards to number of data-dependencies. In the case of larger demand for mapping table entries, smaller, less important dependencies can be omitted or merged with another dependency of the same type without degrading the performance. Since the table contains physical addresses, it is possible that one variable requires more than one entry when mapping of virtual to physical addresses is not continuous. To overcome this issue, large memory pages can be used. The total size of the evaluated mapping table is $32 \times 2 \times 48\,\text{bit} = 348\,\text{B}$. When a new task instance is scheduled for execution, the mapping table is updated with data-dependencies of the task. Upon completion of a task, the runtime clears the entries from the mapping table

that belong only to that task. Both actions require several tens of instructions. The total runtime overhead in terms of number of instructions is negligible when compared with the total number of instructions of any benchmark used for the evaluation.

4.5 Summary

Improving LLC performance is of great importance in modern and future systems. In multi-core processors, threads generating various access patterns are competing for LLC resources. To achieve best performance, it is necessary to protect certain access patterns from being thrashed by accesses coming from another thread. In this thesis we exploit semantic information about applications written in data-flow task-based programming models to better manage the LLC. The runtime system provides the information about task types and task data-dependencies to the LLC in order to improve the insertion policy. We propose two techniques:

TTIP - Task Type-aware Insertion Policy tries to determine the best probability for inserting lines in the recency stack by using runtime-guided dynamic approach that evaluates the performance of several preset probabilities and chooses the best performing one.

DTIP - Dependency Type-aware Insertion Policy is a static policy that assigns to cache lines re-reference intervals based on the type of data-dependency they correspond to. Data that will be used by the next tasks is given more chance to stay in cache by assigning it a more immediate RRI, while read-only data is given less priority.

These two policies use the runtime system for providing the hardware with the necessary information for determining appropriate insertion configurations, which simplifies hardware design. The overheads of the runtime extensions are negligible. The performance benefits compared to LRU are significant for both policies. TTIP performs slightly worse than DRRIP, but uses simpler hardware. DTIP performs better than DRRIP on average, which proves the benefits of using runtime information about the application in designing LLC replacement policies. In comparison with DRRIP, our policies do not use set dueling monitors and do not require a decoder for determining *dedicated follower* sets.

Possible improvements for TTIP include discarding probabilities that perform badly from the training process. DTIP can be extended to distinguish between dependencies, since different dependencies of the same type may have slightly different access patterns that benefit from different insertion positions. Further benefits could be obtained by taking into account task types as well.

Chapter 5

Implementing Reduction in the Cache Hierarchy

This chapter proposes a runtime-assisted technique for performing reductions in the processor's cache hierarchy (RICH). The goal of RICH is to be a universally applicable solution regardless of the reduction variable type, size and access pattern. RICH introduces a hardware component equipped with functional units to perform reductions at any level of the cache hierarchy. Existing constructs in a shared-memory parallel programming model are extended to let the programmer specify at which location in the cache hierarchy a certain reduction should be computed. The runtime system couples the application with the operating system, with the goal to provide the underlying hardware with the information about the reduction variable. This interface is designed without modifying the processor's ISA. As a result, RICH supports the use of algorithms with reductions implemented in third-party libraries.

The main contributions of this chapter are:

- RICH enables the programmer to offload the reduction operation from the core to a desired level of the cache hierarchy. This functionality is facilitated by extending existing OpenMP-like annotations in the parallel code.
- RICH introduces a new hardware component, the Reduction Module, able to perform reductions at all levels of the cache hierarchy.
- Our design couples the parallel application and the underlying hardware with a runtime-assisted interface that does not modify the processors ISA. As a result, RICH is applicable to common scenarios where complex codes use reduction algorithms implemented in third-party pre-compiled libraries, which is not supported in the state-of-the-art hardware techniques for reductions, such as COUP [181] and PCLR [67].
- Experimental results for vector-reductions show that RICH achieves performance improvements of 1.8× on average, compared to the current approaches implemented in

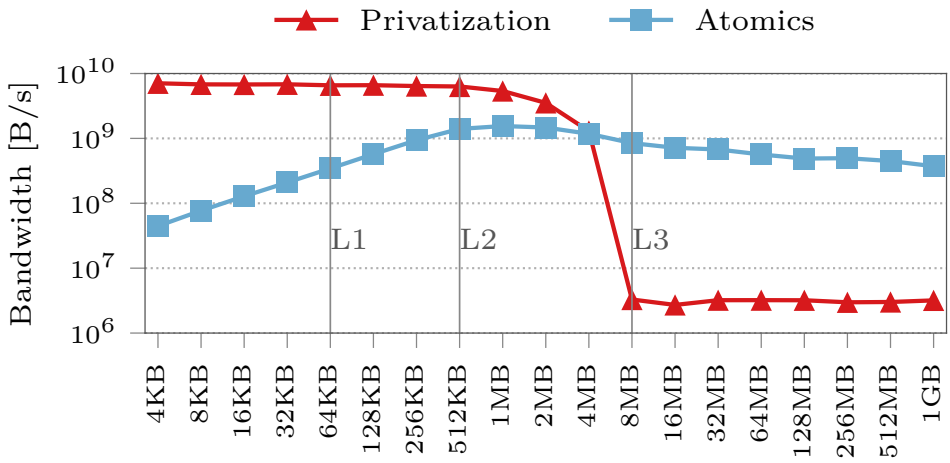


Figure 5.1: Achieved memory bandwidth for RandomAccess benchmark for different reduction array sizes on an IBM POWER8 processor with 192 threads. Vertical lines show the sizes of the data caches per core.

parallel programming models. With scalar-reductions, RICH outperforms software privatization 1.09× on average. RICH performs on average 1.11× faster than COUP.

5.1 Limitations of Current Reduction Techniques

Section 2.3.1 introduces the notion of reductions in the context of HPC applications. There are two intuitive software-based techniques to parallelize reductions, based on privatization and atomic instructions, which are explained in detail in Section 2.3.2. The main conclusion is that these two approaches are suitable for distinct scenarios. Privatization works well for reductions over scalars and small arrays. For larger reduction variables, however, privatized data increases cache pollution. Solutions that use atomics perform worse than privatization for small reduction variables due to frequent cache misses caused by the invalidations of cache lines in the private caches as well as the increased coherence traffic. Figure 5.1 shows an example of such behavior by comparing the achieved memory bandwidth of the two previously-mentioned techniques considering different reduction variable sizes. The analysis is performed using RandomAccess [4], a kernel that accesses the reduction variable following a uniform probability distribution function.

Results show that for small array sizes, privatization is the approach delivering higher performance as it avoids the shared updates. In contrast, atomics achieve significantly lower performance for small problem sizes due to the coherence effects. The performance of atomics improves when the array size increases as conflicts between different threads are less likely to

occur. As the array size approaches the size of the core's portion of the L3 cache, privatization suffers from a performance drop of 1000× due to the overhead of handling the private copies of the reduction variable. Although atomics also show a notable drop in performance, they perform significantly better than privatization. With the further increase of the reduction variable's size beyond 8 MB, the performance of privatization stagnates, while the performance of atomics slowly degrades.

This analysis clearly shows how different reduction methods deliver different performance depending on the size of the reduction variable. Specifically, the size of the reduction variable dictates which reduction technique should be used to achieve better performance. Solutions allowing a manual or automatic selection of the reduction technique are required in order to achieve the best possible performance across all possible scenarios without exposing the programmer to the complexity of implementing application-specific ad hoc reduction techniques. To further reduce the overheads of software techniques, hardware solutions are necessary.

5.1.1 Overcoming Limitations Using Hardware-Assisted Reductions

There are several state-of-the-art hardware techniques addressing issues related to coherence invalidations or data privatization costs. They either implement atomic remote memory accesses [6] or use private cache lines [67, 181]. Remote atomic updates implement atomicity by performing the final reduction, involving all the partial results, at a specific hardware component. These components can be the last-level cache or the memory controller equipped with additional functional units. The use of private cache lines is based on the same concepts as its software privatization counterpart. In this case, processor caches are used as temporal buffers to accumulate intermediate results. Private cache lines are initialized to the neutral element on the first access and are reduced at cache line eviction or at the end of a software routine by generating the final value in the last-level cache. Described designs avoid the high coherence traffic triggered by shared updates to the reduction variable.

However, previous proposals do not perform optimally in all scenarios. Solutions based on remote memory accesses are suitable only for infrequent reduction instructions due to the high overhead of offloading instructions to a shared resource far from the core. Applications with irregular memory accesses do not efficiently use cache memories. The usage of private cache lines in such codes results in a sequence of initialization, cache placement and eviction events. Moreover, in the case of large reduction arrays, privatizing the reduction variable significantly pollutes the content of cache memories. Consequently, further architectural innovations are needed to avoid these issues while keeping the benefits of low coherence traffic.

5.1.2 Ongoing Challenges

Reductions on scalar variables or small output arrays are well supported in current designs. However, reductions considering large arrays or displaying irregular access patterns require novel techniques to avoid performance degradation due to cache pollution and increased coherence traffic. Proposed hardware and software solutions are just suitable for a subset of scenarios, depending on the size of the reduction variable and its memory access pattern. To the best of our knowledge, a technique effective for all reduction scenarios has not been proposed. Moreover, previously proposed hardware techniques require ISA extensions to handle reduction operations, which makes them incompatible with applications that use pre-compiled libraries containing reduction operations.

The design described in the remaining of this chapter aims to achieve the following goals: (i) To achieve better performance than the state of the art considering a wide range of reduction variable sizes and different memory access patterns. (ii) To avoid modifications to the processor's ISA and thus maintain the compatibility with pre-compiled and dynamically linked libraries. (iii) To let the programmer expose application-specific knowledge to the hardware without the need for ad hoc implementations of reductions.

5.2 Implementing Reductions in the Cache Hierarchy

RICH is a runtime-assisted technique for performing reductions in the cache hierarchy. The programmer makes use of simple source code annotations to identify reduction variables and specify both the reduction operator and the hardware components where reductions should take place. Such annotations are expressed using the extended `reduction` pragma directive [137]. The runtime system is responsible for providing the hardware with the information specified by the programmer. Finally, the additional hardware components in the processor's caches are responsible for handling and executing the reduction operations. RICH relies on the following extensions:

- Programming model support to define the reduction technique and the runtime system extensions to set up the relevant hardware components.

- A novel hardware component, called Reduction Module (RM), located at the cache hierarchy. The RM performs the reduction instructions issued by the cores.

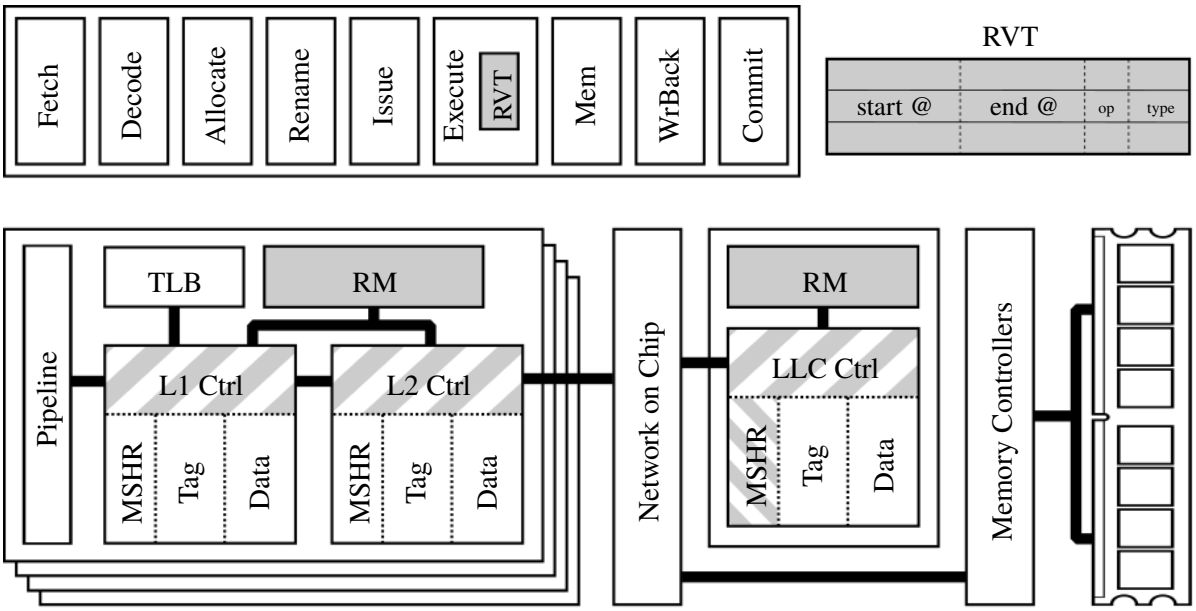


Figure 5.2: Top left: The pipeline schematic with the added RVT component; Top right: Microarchitecture of the RVT; Bottom: Microarchitecture of the memory hierarchy with the added Reduction Modules. New components are colored in solid gray and modified components in striped gray.

- Microarchitectural extensions in the processor and its memory hierarchy to handle reduction requests in the core and their propagation to the RM through the cache hierarchy. A part of these extensions is the Reduction Variable Table (RVT) used to recognize the instructions participating in a reduction based on the referenced memory address.

This section describes these extensions in detail. Finally, it discusses different design decisions and the implications of the proposed processor functionalities.

5.2.1 Microarchitectural Support for Reductions

The following paragraphs describe the hardware modifications proposed to execute reduction operations in the cache hierarchy. Figure 5.2 shows the relevant details of a multi-core processor microarchitecture with the added (solid gray) and modified (striped gray) components. The proposal is described in a context where each core is equipped with two levels of private caches while the Last-Level Cache (LLC) is shared among all cores. Our architectural innovations can be also deployed, with minor adaptations, in other contexts with different cache memory hierarchies. A Reduction Module (RM) is added to the private caches of each core as well as to the LLC. The private caches share a single RM. The cache controllers are modified

to communicate with the RM and to handle reduction store instructions. A small hardware component that holds the range of reduction variables for the current thread is placed in each core. All added and modified hardware structures are described in detail in the following paragraphs.

Recognizing reduction instructions is partially facilitated by a special hardware structure called Reduction Variable Table (RVT). For a given address, the RVT determines if the address belongs to a reduction variable in the current thread. For load and store instructions, the RVT is accessed in the execute stage, once the destination address of the memory operation is calculated. The RVT holds the ranges of virtual addresses corresponding to the reduction variables (*start* @ and *end* @), as well as the data type (*type*) and the operator (*op*) used for accumulating values into each reduction variable. The content of the RVT is managed by the runtime system, as explained in Section 5.2.2.

The reduction operation is composed of: a load from the reduction variable into a register, an arithmetic or logic operation that updates this register and a store of the modified register to the original memory location. Depending on the target architecture, the atomicity of the load-modify-store chain is achieved in different ways: (i) Load-Link and Store-Conditional instructions [14, 78, 168] and (ii) Compare-And-Swap construct [81]. RICH is implemented to support both synchronization mechanisms. Since RICH uses only the address accessed by the loads and stores to determine if they participate in reduction operation, it is not important which mechanism is used to ensure the atomicity of the reduction operation.

RICH supports reduction operations that update the reduction variable with a sequence of load-modify-store instructions. All reduction operators defined in the OpenMP standard 5.0 have this property. This covers arithmetic instructions ADD, SUB, MUL, logical operations AND and OR, bitwise operations AND, OR and XOR and MIN/MAX. In addition, RICH supports the DIV operation. Operations on both integer and floating point data are allowed.

When a core recognizes a reduction operation, the arithmetic or logic instructions involved in it plus the load instructions to the reduction variable are converted into NOP instructions in the core's pipeline. After effectively eliminating these instructions, the CPU converts the reduction store instruction into a special store instruction that holds information from these removed instructions: the reduction operator, the data to be reduced and the reduction variable's address. The special store instruction is propagated through the cache hierarchy until it arrives to the cache level configured to perform the reduction. To ensure the correctness of this design, an instruction consuming the reduction variable is not permitted to execute before the reduction

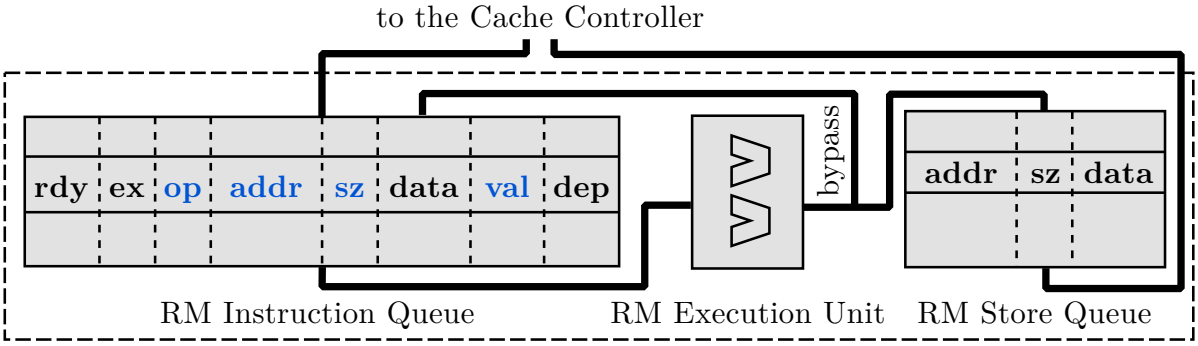


Figure 5.3: Microarchitecture of the Reduction Module.

operation has finished. This is enforced by using existing OpenMP synchronization primitives such as barriers or dependencies between different user functions [137].

Reduction Module. Figure 5.3 shows the microarchitecture of a Reduction Module (RM) which consists of the following three hardware structures:

The *RM Instruction Queue* (RMIQ) contains instructions that are to be executed or are being executed by the RM. The RMIQ is designed as a circular queue to maintain the order of the inserted instructions. Each entry in the RMIQ contains information specified by a reduction instruction, i.e the reduction operation to be performed (op), the address of the reduction variable (addr), its size (sz) and the value that is to be reduced into the reduction variable (val). The *data* field holds the current value of the accessed location within the reduction variable, which may not be available in cache at the time of inserting an instruction into the RMIQ. In that case, the entry is marked as "not ready" (field *rdy*). Only ready instructions can be executed. The *ex* field indicates whether the instruction is being executed. The *dep* field points to an entry in the RMIQ that depends on the result of this instruction.

The *RM Execution Unit* (RMEX) contains the logic that performs arithmetic and logic operations on all standard data types. It consists of an Arithmetic-Logic Unit (ALU) and a Floating-Point Unit (FPU).

The *RM Store Queue* (RMSQ) is a circular buffer for storing the results of the reductions until they can be written back to the cache's data storage. Entries of the RMSQ contain an address, the corresponding data and its size. Whenever a cache's write port is not in use, the controller writes the oldest entry from the RMSQ into the cache and removes it from the RMSQ.

RICH configurations. In our proposed architecture, the system can be configured to perform reductions at different levels of the cache hierarchy, i.e. at any of the private caches or the shared last-level cache. Although the inner behavior of the RM is the same, the handling of

the reduction instructions in the cache controller depends on the RICH configuration. Depending on the cache level where the reduction is performed, the following three configurations are defined: $RICH_{L1}$, $RICH_{L2}$ and $RICH_{LLC}$. Configurations $RICH_{L1}$ and $RICH_{L2}$ imply that partial reduction is performed in the corresponding private caches. After the reduction task is finished, the reduction lines from caches are written back and the final reduction is carried out at the LLC level. In the $RICH_{LLC}$ configuration, only the RM in the LLC is active. Inactive RMs do not consume energy as they are turned off by the power gating mechanism [143].

The responsibility for choosing the RICH configuration for each task lies on a programmer, who can use knowledge about application's cache performance accompanied with the processor's architectural specifications (e.g., cache size and organization). Cache performance of a parallel code can be obtained by profiling or estimated using expert knowledge of the code and its memory access patterns. Alternate designs consider automatic selection of the best configuration based on various run-time performance metrics such as IPC and cache performance. The experimental results in Section 5.4 show that $RICH_{L1}$ and $RICH_{L2}$ are best-performing configurations and that they both achieve similar performance. A real RICH-enabled processor could implement only one of these configurations. Alternatively, the processor could implement all proposed RICH configurations, with the default configuration being $RICH_{L1}$. Other configurations can be selected by a programmer to support future applications that would benefit from reductions in shared caches.

Processing a reduction instruction in caches. When a special store instruction involved in a reduction reaches the cache level where it will run, it is inserted in the RMIQ and marked as "not ready". Before the instruction can start executing, the current value of the reduction variable needs to be fetched into the RM. Depending on the state of the RM and the corresponding cache, different actions are performed:

- If the RMIQ contains an entry reducing to the same address as the new instruction, the new instruction needs to wait for the data from the preceding instruction, whose *dep* field is updated to point to the newly inserted instruction.

- Otherwise, the reduction data has to be read from the RMSQ or the data cache. If the RMSQ contains an entry matching the address of the new reduction instruction, data is read from the RMSQ into the *data* field in the RMIQ and the new instruction is marked as ready.

- If the reduction is performed in a private cache ($RICH_{L1}$ and $RICH_{L2}$), data is fetched from the cache in the case of a *cache hit*. In case of a *cache miss*, a cache line is allocated

and filled with neutral elements corresponding to the reduction operator. When a cache line holding the reduction variable is evicted from a private cache, it is reduced by the RM inside the LLC.

- If the reduction is performed in the shared cache ($RICH_{LLC}$), the data is fetched from the cache's data storage. In case of a *cache miss*, a standard request is sent to the memory controller and the pointer to the RMIQ entry requesting the data is inserted into the MSHR. Once the data arrives to the cache, it is written into the appropriate entry in the RMIQ, simultaneously marking the entry as ready.

When scheduling an instruction for execution, the controller takes the first ready instruction from the head of the RMIQ, sets its *ex* bit and forwards the entry to the RMEX for execution. Once the execution finishes, the result, together with the destination address, is stored in the RMSQ, while the corresponding entry in the RMIQ is freed. In case an instruction is waiting in the RMIQ for the output of the finished reduction operation, this output is written in the *data* field of the corresponding entry, marking it as ready. Entries from the RMSQ are written back to the cache's data store when the cache's write port is available and removed from the RMSQ.

When a request is sent to the RM, the corresponding cache line is locked, which prevents it from being evicted. The lock guarantees that the line is present for the write-back operation from the RM, which releases the lock upon completion.

Accessing a reduction variable outside of the reduction scope. Once the reduction finishes, the application often accesses the reduction variable for further processing. It is necessary to differentiate between accesses generated inside the reduction scope and those accesses that happen outside of reduction context. RICH uses the RVT to recognize reduction instructions. The runtime system populates the RVT before the reduction context begins and clears it after the reduction is finished. This mechanism is described in Section 5.2.2. If the variable is accessed outside of the reduction context, the request is processed as a normal memory instruction. Also, instructions accessing the reduction variable that do not belong to the reduction operation are not allowed to run before the whole reduction has finished. This is enforced by using OpenMP synchronization primitives [137].

Memory consistency. Memory consistency of non-reduction data is not affected. All loads and stores are issued by the core in a way that maintains Total Store Order (TSO) memory consistency model [158]. On the other hand, the loads and stores issued by the RM and non-dependent, non-reduction loads and stores issued by the core can be seen in different order by the memory subsystem. To guarantee that an access to a reduction variable never

returns a wrong value, (i) the programmer ensures that, within the reduction task, the reduction variable is only accessed with read-modify-write pattern, i.e. reduction operation, and (ii) the source-to-source compiler inserts a memory fence after a reduction task to guarantee that a successive consumer task accesses the correct data.

Cache coherence. RICH does not modify the cache coherence protocol. Depending on the RICH configuration, specific explicit synchronization actions are performed to guarantee coherence of reduction data in the caches. In all configurations, the reduction variable can either be present in the cache's data store or in the RMSQ of the same cache. The cache controller considers both locations when searching for a cache line of an in-flight reduction variable. $RICH_{L1}$ and $RICH_{L2}$ require a final reduction of the partially reduced data, which is performed at the end of the reduction task. A memory fence, inserted by the source-to-source compiler, guarantees that these data is not consumed before the final reduction takes place.

It is possible that partial copies of the reduction variable are present in caches above the cache that performs the reduction. To guarantee the correctness of execution, these copies are sent to the cache level where the reduction takes place. This action can be done in parallel with the reduction task and, thus, does not incurr additional overhead.

Support for precise exceptions and speculation. RICH implementation maintains support for precise exceptions by guaranteeing in-order retiring of instructions. Events caused by exceptions and misspeculations are bidirectionally communicated between the core and the RM. In case of an exception or misspeculation, the appropriate in-flight instructions in the RM are flushed, new values stored in the RMSQ are discarded and old values stored in the RMIQ are restored.

5.2.2 Programming Model and Compiler Support

The programming model support for the proposed hardware design relies on the existing implementations of the most popular shared memory parallel programming model, OpenMP [137]. OpenMP supports both loop-level and task-based parallelism. In task-based codes, the programming model offers explicit synchronization with *taskwait* constructs. In addition, if programmers define data dependencies between tasks, OpenMP automatically ensures correct execution in a data-flow manner by respecting the user-specified task data dependencies. Specifically, tasks that depend on data produced by other tasks are scheduled to execute only when these data dependencies are satisfied. When loop-based parallelism is employed, implicit barriers are added to enforce synchronization.

We design RICH to be agnostic to the applied parallelization technique. The proposed programming model extensions are built on top of the existing implementation of reductions in OpenMP.

This is beneficial as any extension to a programming model requires careful design for consistency with minimal implications on unrelated constructs, user understanding and compatibility with previous versions and existing codes.

We extend the reduction directive as follows, with the added parameter shown in bold.

$$reduction(reduction\text{-}ident. : \textbf{\textit{[reduction-technique]}} : list)$$

As defined in Section 2.19.5.4 of the OpenMP 5.0 standard [137], *reduction-identifier* specifies the reduction operator while *list* specifies the list of reduction variables. The added optional field *reduction-technique* specifies which reduction technique to use (CPU, $RICH_{L1}$, $RICH_{L2}$ or $RICH_{LLC}$). The default configuration, CPU, executes the reduction operations in the core and does not use the hardware acceleration in the RM. Using the information specified in this annotation simplifies our design as it does not require adding special reduction instructions to the processor's ISA, and, therefore, maintains the compatibility with pre-compiled libraries.

The information specified in the programming model directives is forwarded to the RVT by a function call implemented in the runtime library using instructions on memory-mapped registers. This call is inserted by a source-to-source compiler in the code location where the executing thread encounters the beginning of the parallel region or a task that participates in a reduction. This source code location is considered as the start of the reduction scope, which is terminated once all tasks or iterations from that parallel region finish.

Figure 5.4 illustrates how a programmer uses the programming model extensions. The starting point in this example is a parallelized vector reduction code that uses the OmpSs reduction construct for tasks. The example is applicable to both task-based and loop-based parallel codes in both OmpSs and OpenMP programming models. The programmer selects the cache level where the reduction will take place by taking into account properties of the application like the workload size, reduction variable size, and its memory access pattern. This example considers a case when reduction is executed in the L1 cache, which is specified in the reduction clause (1), as defined earlier in this section. The right-hand side of the figure shows the transformations done by the Mercurium compiler. Functions using prefix `nanos` encapsulate the task creation and synchronization. The source-to-source compiler inserts calls to functions implemented in the runtime system library used to populate the RVT with the

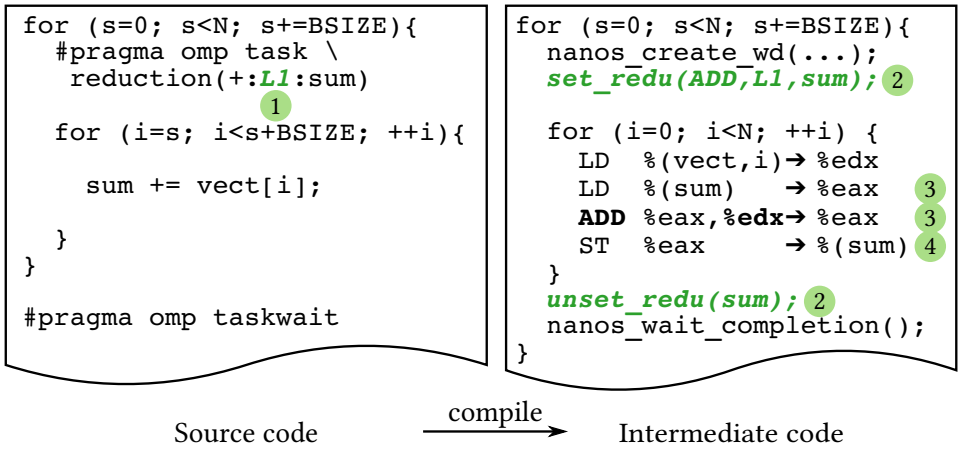


Figure 5.4: Left: The OmpSs-based source code of a reduction using RICH features. Right: Code transformations done by Mercurium compiler.

information about reduction variables and the chosen reduction location 2. During program's execution, the load and arithmetic operations belonging to reductions 3 are discarded. The store instruction 4 is enriched with the reduction operation type (`ADD`, from RVT) and the register holding the value to be reduced (`%edx`, from the preceding ALU instruction). The enriched store instruction is then forwarded to the core's RM. The further handling of reduction instructions by the hardware is explained in detail in Section 5.2.1.

5.2.3 Discussion

This section discusses three different techniques for recognizing reduction instructions in the processor core:

(i) Section 5.2.1 describes a design that utilizes a structure called Reduction Variable Table (RVT) to identify instructions involved in reductions.

(ii) An alternative implementation does not use the RVT, but requires changes in the memory management subsystem. This technique extends the OS page table and the translation look-ahead buffer (TLB) with the fields that specify whether a memory page corresponds to a reduction variable and which operator is used to accumulate on that reduction variable. Such an implementation requires that reduction variables are stored in separate memory pages from the rest of the application data. The operating system adds a functionality to mark reduction memory pages in the page table, which is exposed to the runtime system. At the beginning and the end of each reduction parallel region, the page table and the TLB need to be updated.

Contrary to the RVT-based design, the identification of reduction instructions is postponed until the instruction arrives to the data cache.

(iii) Extending the ISA with reduction store instructions with the format *STORE (reduction_variable, operation, value)* maintains most of the hardware unmodified. On the other hand, it requires back-end compiler support and recompiling third-party libraries. Since RICH modifies the programming model's directives, changes are required to the source-to-source compiler that translates the directives into the appropriate runtime library. Additionally, extending the ISA with reduction instructions requires adding support to the backend compilers. This request is more difficult to satisfy due to the existence of numerous vendors that implement source-to-machine code compilers.

Each described solution has a set of strengths and weaknesses and choosing the appropriate approach depends on the goals of the designer. In the context of this work, the aim is to preserve compatibility with pre-compiled libraries without increasing the complexity of the system. Both RVT-based and TLB-based solutions satisfy the compatibility requirements. These solutions use functions implemented in the runtime system library to set-up the hardware components for tracking reduction variables. Running code compiled for RICH-enabled CPU on a CPU without the support for RICH is achieved by disabling the set-up functions in the runtime library. This compatibility is not possible in a solution that modifies the ISA. The main drawbacks of TLB-based approach are high memory fragmentation for small reduction variables, complex modifications to the memory management system as well as necessity for pipeline stalling due to late detection of the reduction instructions. Taking these factors into account, RVT-based design is selected as a way of recognizing reductions.

5.3 RICH Design Decisions

The experiments used for the exploration in this section are performed on the simulation infrastructure described in Section 3.1 using the benchmarks presented in Section 3.2. The input sizes for each benchmark are shown in Table 3.4.

5.3.1 Design Space Exploration

There are three design parameters that influence the performance of the proposed Reduction Module: (i) number of RMIQ entries; (ii) number of the functional units in the RMEX; and (iii) design of the functional units in the RMEX. This section is dedicated to the evaluation

Table 5.1: RICH design space exploration.
The values in bold are the ones selected for the final design as the result of the design space exploration.

RM	RMEX:	**1**, 2, 4 FUs
	FU design:	**pipelined**, non-pipelined
	RMIQ entries:	1, 2, 4, 8, **16**, 32, 64
RVT	1, 2, 4, 8, 16, **32**, 64, 128, 256, 512, 1024, 2048 entries	

of the impact of the aforementioned parameters on the processor's performance. In addition, different latencies of arithmetic operations are explored in order to evaluate the performance of the Reduction Module with all supported arithmetic and logic operations on both fixed and floating-point numbers. Operations are modeled to have the same latency as the corresponding instructions in current Intel processors. The list of parameters and explored values is presented in Table 5.1. The final purpose of this analysis is to determine the optimal parameters of the RM in the context of the simulated processor and evaluated benchmarks.

Figure 5.5 shows the speedup of RICH versus the ideal implementation of reductions, where each reduction instruction takes 1 cycle and does not interact with the cache subsystem. The speedup of 1 represents the upper theoretical limit for the achievable performance. X-axis corresponds to different configurations for some of the RM's components. The exploration analyzes the effects of different counts of functional units and their design (non-pipelined vs. pipelined). The number of RMIQ entries per functional unit is denoted with *RMIQ*.

Each data point shows the mean speedup among all benchmarks for a specific group of reduction operations and the cache level at which the reduction is performed. Operations of similar latencies and behaviors are grouped together. Each group is designated with one of three symbols shown in the left-hand side of the legend. Data corresponding to configurations $RICH_{L1}$ and $RICH_{L2}$ are plotted together as these configurations exhibit similar trends and sensitivity to the RM's parameters. Points belonging to $RICH_{L1}$ and $RICH_{L2}$ are painted in light blue color, while points associated with $RICH_{LLC}$ are presented in dark blue, as shown in the right-hand side of the legend.

The results show that reductions using addition or multiplication exhibit little sensitivity to the RM's configurations due to relatively low operation latency. Division, however, benefits from having more functional units and a larger RMIQ. Using two FUs in the RMEX improves performance of $RICH_{L1}$ and $RICH_{L2}$ by 2.7% on average compared to the single FU-design, but comes with 94.2% higher area overhead. Pipelined functional units benefit from more RMIQ entries as they are able to execute multiple independent operations simultaneously,

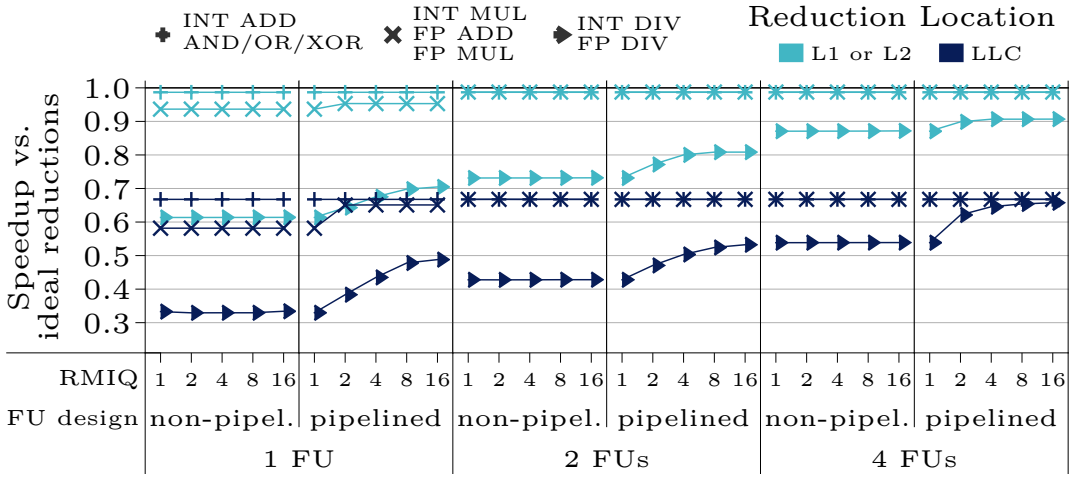


Figure 5.5: RICH speedup vs. ideal reductions for different configurations of functional units and RMIQ in the RM, depending on operation type and reduction location.

contrary to the non-pipelined designs. Considering these results, it is decided to use a single pipelined ALU and a single pipelined FPU. RMIQ with 16 entries is chosen to get best possible division performance. This configuration will be used for the further evaluation of RICH presented in the remaining of this document.

Reduction Variable Table (RVT) is used by the core to recognize the instructions involved in the reduction and is described in detail in Section 5.2.1. The RVT contains multiple entries to support tasks with more than one reduction variable. To evaluate the impact of the RVT size on the processor's performance, the RVT is modeled using CACTI 7 based on the chip frequency of 2.4 GHz. The model shows that small RVT designs with up to 256 entries can be accessed within 1 cycle, while medium (up to 1024 entries) and larger designs require 2 and 3 cycles respectively. RVT configurations with latencies of 2 and 3 cycles degrade the overall performance by negligible 0.8% and 1.8% on average, respectively, compared to the RVT with 1-cycle latency. Applications having more in-flight reduction variables that cannot fit in the RVT are still executed correctly. In that case, reduction operations on variables that do not fit into the RVT are not accelerated by the RM. To optimally run all benchmarks from Table 3.3, a 4-entry RVT is needed. A design with a 32-entry RVT is selected as it covers potentially more complex codes while keeping 1-cycle access latency.

5.3.2 Hardware Cost of Implementing RICH

This section presents the discussion on the area and storage required to implement RICH. As explained in Section 5.2.1, the Reduction Module (RM) consists of two queues, RMIQ and

Table 5.2: Hardware cost of implementing RICH in 22nm.

	RVT	RMIQ	$RMEX_{ALU}$	$RMEX_{FPU}$	RMSQ
Area [mm^2]	0.0002	0.013	0.038	0.223	0.003
Storage [KB]	0.055	3.22	-	-	2.09

Baseline processor's area	192.48	mm^2
Reduction Module area (IQ + EX + SQ)	0.277	mm^2
Total area overhead (17 RMs + 16 RVTs)	4.71	mm^2
Total area overhead relative to the baseline	2.45	%

RMSQ, and an Execution Unit (RMEX) that contains two functional units, i.e. one ALU and one FPU. Private caches L1 and L2 share a single Reduction Module and the LLC has its own RM. Additionally, the design has a single Reduction Variable Table (RVT) per core. Thus, the simulated 16-core processor contains 17 Reduction Modules and 16 RVTs.

Table 5.2 shows the sizes of particular RM components as well as overall area of a processor occupied by the added hardware. According to the McPAT model, the FPU used in the RM is 60 % smaller than the FPU in the core due to the removal of support for SIMD instructions. The modeled ALU supports 32bit and 64bit arithmetic and logic operations on integers.

The total area consumed by the RMs is 4.71 mm^2 or 2.4 % of the whole die area of the baseline processor. Alternatively, a processor design that uses a larger LLC can be considered. The simulations show that a processor with a 40 MB, 16-way set-associative LLC obtains on average 1.5% better performance than the reference processor. RICH performs 1.51× better than the baseline processor while requiring 30% less additional chip area than a 40 MB LLC. Thus, it is concluded that RICH utilizes additional hardware more efficiently than just extending already existing hardware components like the LLC.

To further reduce the hardware overhead of RICH, a design where RMEX uses the core's functional units (FUs) is also considered. A design where all RMs use the functional units from the cores has the lowest area overhead of 0.28 mm^2 or negligible 0.14% of the processor area. However, such implementation requires long data paths from the LLC to the core, which can significantly impact the latency of reduction operations in the LLC. In an improved design, the RM in the LLC has its own functional units, while the RMs in private caches share the FUs with their cores. Since private caches are close to cores, timing requirements are easier to satisfy. However, the core's components such as reservation stations have to be modified to support sharing of the FUs. The explored configurations offer a clear trade-off between the performance, the design complexity and the area overheads.

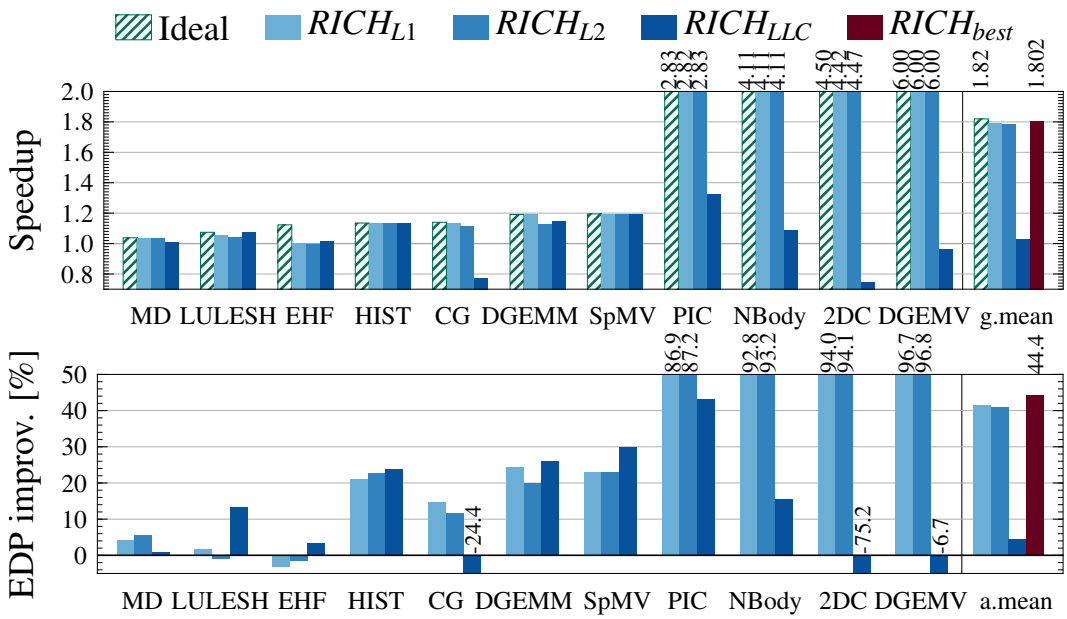


Figure 5.6: Speedup and Energy-Delay Product improvement of RICH over the baseline with atomics for benchmarks with reductions on arrays.

5.4 Evaluation

This section presents the evaluation of RICH, comparing it against widely-accepted software-based reduction techniques based on privatization and atomic operations, and a state-of-the art hardware solution. The evaluation is performed using the benchmarks described in Section 3.2 with input parameters shown in Table 3.4.

5.4.1 Evaluating RICH with Vector-Reductions

The analysis in Section 5.1 shows that, in case of larger arrays, reductions implemented with atomics achieve better performance than software privatization. Taking this into consideration, atomics are chosen as the baseline for evaluating RICH on benchmarks that perform reductions on vector variables. All reported results in the following sections correspond to the overall performance including both reduction and non-reduction tasks.

The upper part of Figure 5.6 shows performance speedups of the three RICH configurations normalized to the reduction approach based on atomic operations. The figure also includes performance of ideal reductions, where each reduction operation takes 1 cycle and does not issue requests to the cache hierarchy. The ideal configuration indicates the maximal achievable speedup per benchmark.

On average, $RICH_{L1}$, $RICH_{L2}$ and $RICH_{LLC}$ perform 79.4%, 76.6% and 2.9% faster than the baseline, respectively. In general, RICH outperforms the implementation with atomics due to a combination of several factors. First RICH performs the load-modify-store sequence as one instruction, and therefore reduces the number of requests to the cache hierarchy and does not use ALUs and FPUs in the core. Second, RICH does not suffer from coherence effects caused by conflicts between different threads, contrary to the reductions with atomics. Coherence effects manifest themselves in increased miss ratio to reduction variable due to invalidations by other threads and retrying the update operation or waiting for a lock release, depending on the implementation of atomics in the target architecture. Finally, when the reduction variable is updated at lower-level caches, it is not present in the higher-level caches, reducing the pollution of these caches. This effect results in better cache performance for input data. Additionally, due to larger sizes of lower-level caches, accesses to the reduction variable result in less misses, which is explained in the following section. These three factors contribute to, on average, lower execution time of RICH compared to atomics.

The highest performance gains are observed in PIC, NBody 2DC and DGEMV, where $RICH_{L1}$ performs from 2.8× to 6.0× faster than atomics-based approach. The main contributor for faster execution in case of PIC, NBody and DGEMV is the reduced number of misses, as shown in Figure 5.7. On the other hand, in the case of 2DC, only a small reduction in cache misses is observed. Even though collisions still occur, RICH reduces amount of cycles spent on waiting due to lock contention.

The lower part of Figure 5.6 shows the improvements in energy-delay product (EDP) of RICH compared to the baseline with atomics. On average, the best RICH configuration per benchmark improves EDP by 44.4% compared to the baseline. The highest EDP improvements are observed for the benchmarks where RICH achieves highest speedups, i.e., PIC, NBody, 2DC and DGEMV with 87.2% to 96.8% improvement. EDP is mainly improved due to lower execution time and reduced energy consumption by the caches due to reduced amount of misses in RICH configurations, which is demonstrated in Section 5.4.2.

$RICH_{LLC}$ consumes less power than $RICH_{L1}$ and $RICH_{L2}$ since it uses just one RM in the LLC. This effect is clearly observed for EHF, Hist and SpMV. Even though these benchmarks achieve similar performance across all RICH configurations, there are notable differences in the EDP among three RICH configurations. The additional reason for such behavior is the reduced number of misses in $RICH_{L2}$ and $RICH_{LLC}$ configurations, as described in Section 5.4.2.

$\boldsymbol{RICH_{best}}$ is defined as the optimal RICH configuration per benchmark. In benchmarks reducing on vector variables, $RICH_{best}$ achieves performance speedup of 1.8× and 44.4% better

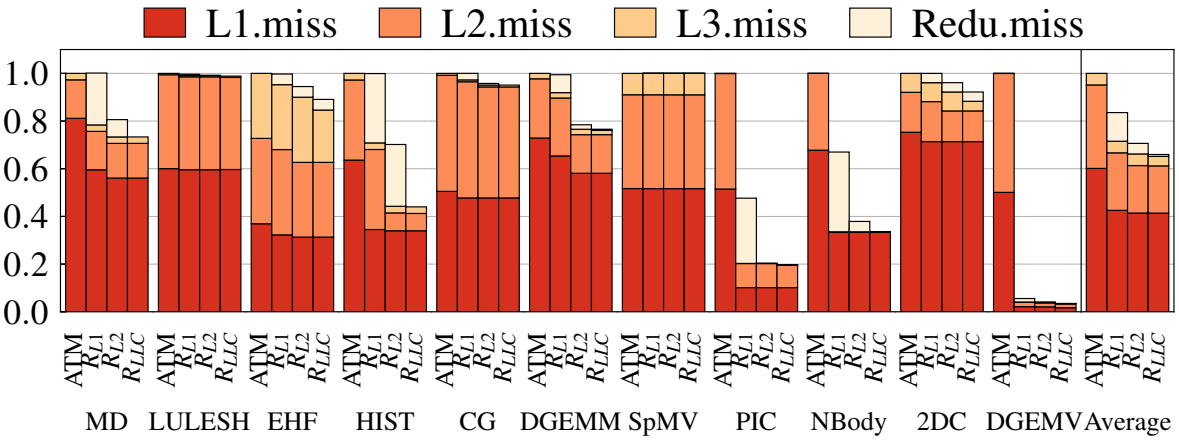


Figure 5.7: Breakdown of misses across all cache levels. `Redu.miss` denotes misses generated by the Reduction Module in the cache level where the reduction is performed. Configurations *Atomics*, $RICH_{L1}$, $RICH_{L2}$ and $RICH_{LLC}$ are denoted as ATM, R_{L1}, R_{L2} and R_{LLC}, respectively.

EDP than atomics. RICH allows the programmer to specify the optimal reduction location via pragma constructs supported by the programming model, as described in Section 5.2. In this context, $RICH_{best}$ represents the performance improvement that can be obtained by choosing the best location to carry out the reductions.

5.4.2 Impact of RICH on Cache Performance for Vector-Reductions

Figure 5.7 shows the breakdown of misses for all three cache levels regarding benchmarks that perform reductions on vectors. Misses are normalized to the total misses occurring when reductions rely on atomic operations (configuration *ATM*). Label `Redu.miss` corresponds to the misses triggered by accesses to the reduction variable generated by the Reduction Module. These misses occur in the cache level where the reduction is performed.

In eight benchmarks there is a negligible difference in total misses between *ATM* and $RICH_{L1}$. In these cases, the reduction variable is accessed in a more structured manner which does not cause data invalidations invoked by the coherence protocol. Nonetheless, $RICH_{L1}$ still achieves speedup over atomics due to time penalties when using atomic instructions, in addition to the fact that RICH internally compacts the load-modify-store instructions into one instruction. For other benchmarks, one can observe the effects of coherence invalidations that manifest themselves as increased total number of misses in *ATM* compared to $RICH_{L1}$. This is most notable in benchmarks where RICH achieves highest performance speedups, i.e., DGEMV, NBody and PIC.

Another interesting effect to analyze is the significant average reduction of total number of misses when performing reductions in lower-level caches, i.e., the L2 and the LLC. The cause for this behavior is the reduced pollution of the L1 cache by the reduction variable and higher hit ratio to the reduction variable in L2/LLC due to larger size of those caches. This effect is observable in almost all benchmarks and is most prominent in MD, DGEMM, Hist, NBody and PIC. The reduction in misses is not translated into performance improvements of $RICH_{L2}$ over $RICH_{L1}$ because the added miss penalties are hidden by an out-of-order core. However, as having less misses results in reduced cache traffic, the energy consumed by the memory hierarchy is reduced, which is demonstrated through EDP improvements in Section 5.4.1.

5.4.3 Evaluating RICH with Scalar-Reductions

As shown in Section 5.1, privatization is the best performing technique for handling reductions in applications with reductions on scalar variables. Therefore, software privatization is selected as the baseline for parallel codes that perform reductions on scalars.

The top part of Figure 5.8 shows the performance speedup of three RICH configurations normalized to software privatization. On average, $RICH_{L1}$ and $RICH_{L2}$ perform 9.5% faster than the baseline. RICH achieves the highest performance benefits for applications that have highest ratio of reduction instructions with respect to the overall number of instructions, such as DotP, NB, PS and VctR. The performance benefits come from the reduced number of instructions executed in the core. Specifically, since reduction operations are offloaded to the Reduction Module (RM), the core can execute instructions in advance while the RM computes the reduction in the cache. NQ and KT exhibit marginal improvements since the amount of instructions involved in reduction operations of these benchmarks represents a small percentage of the whole execution.

The figure also shows the performance of an idealistic implementation of reductions, where each reduction operation is performed instantaneously and does not issue requests to the cache hierarchy, as described in Section 5.4.1. Results show that RICH achieves close-to-ideal performance in all benchmarks on scalars except VctR, a benchmark that calculates a sum of double-precision floating point values on a scalar variable of the same type. Since this operation is modeled to take 3 cycles, the serialization of reductions in the RM combined with the high frequency of reduction instructions in this benchmark limits the performance achieved by RICH.

In $RICH_{L2}$ configuration, fetching data from the L2 cache to the RM takes more cycles than the equivalent operation in $RICH_{L1}$ configuration due to the higher access latency of the L2

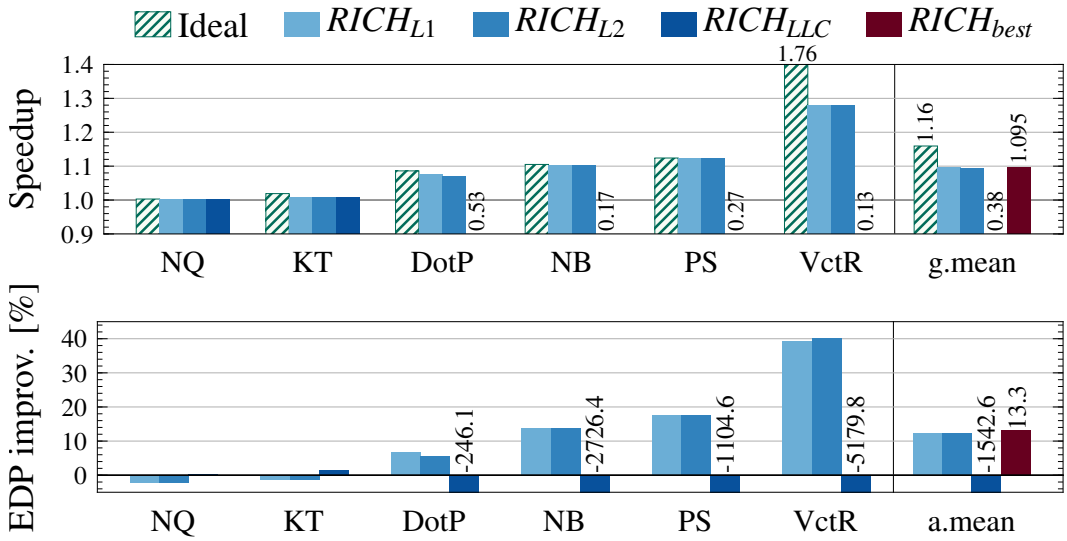


Figure 5.8: Speedup and Energy-Delay Product of RICH compared to the baseline with software privatization for benchmarks that perform reductions on scalars.

cache. Nonetheless, it can be observed that these configurations achieve the same performance. The explanation for this behavior is the fact that the execution of reduction instructions in $RICH_{L1}$ and $RICH_{L2}$ configurations is overlapped with other instructions executed at the CPU level in a way that the reduction latencies are hidden.

$RICH_{LLC}$ suffers from performance slowdowns compared to the baseline. With reductions on scalar variables at the LLC, all reduction instructions are serialized as they depend on each other. This explains the performance slowdown suffered by DotP, NB, PS and VctR. Contrarily, this effect is not visible in NQ and KT. Due to the low ratio of reduction instructions in these benchmarks, serialized instructions from one iteration in the LLC's RM have time to finish before the arrival of instructions from the next iteration. Moreover, the benefits of offloading instructions to the RM outweigh the small performance degradation due to serialization in the LLC.

The bottom part of Figure 5.8 shows the improvements in energy-delay product (EDP) of RICH compared to the baseline with software privatization. On average, $RICH_{best}$ improves EDP by 13.3% compared to the baseline. The main factor contributing to EDP improvements is faster execution time, particularly for the four benchmarks where RICH achieves notable speedups. In the case of NQ and KT, $RICH_{LLC}$ achieves the best EDP due to less power overhead of having just one RM in the LLC compared to having one RM per core in $RICH_{L1}$ and $RICH_{L2}$ configurations.

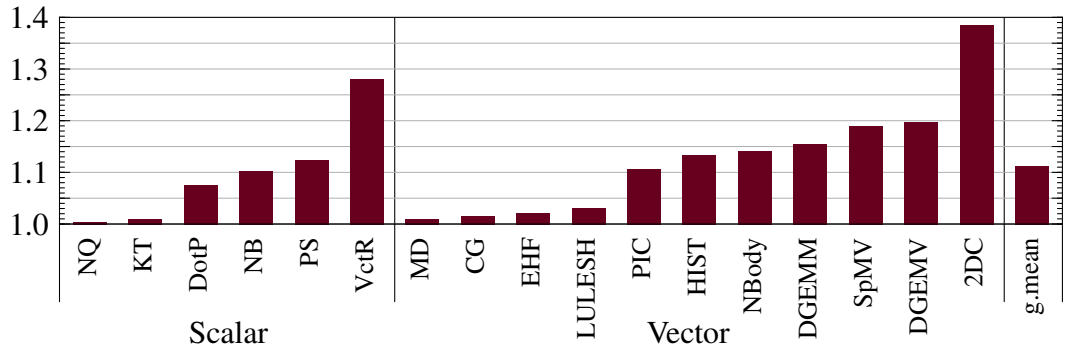


Figure 5.9: Speedup of $RICH_{best}$ compared to COUP [181].

5.4.4 Comparison with Other Proposals

In this section, $RICH_{best}$ is compared with the state-of-the-art technique for reductions in hardware, COUP [181]. $RICH_{best}$ is defined as the best RICH configuration per benchmark in terms of performance. COUP implements privatization of the reduction variable in the private caches by modifying the cache coherence protocol. This design allows multiple cores to acquire a line with update-only permission. The partial results are accumulated in private caches using in-core functional units, while the final result is calculated on demand in the LLC or memory controller, which are equipped with dedicated functional units. To simulate COUP, its functionality is implemented in the context of the simulation infrastructure used for this evaluation. The model assumes that coherence operations performed by COUP have zero cost. The handling of *update-only* lines is implemented in detail.

Figure 5.9 shows the speedup of $RICH_{best}$ compared to COUP. $RICH_{best}$ achieves 1.11× better performance on average and up to 1.38× improvement in case of 2DC. Significant improvements are also obtained for Hist, NBody, DGEMM, SpMV and DGEMV. RICH outperforms COUP due to reducing the traffic between the core and the L1 cache as the reduction variable is updated directly in the cache. Moreover, the ability to execute reductions at lower-level caches benefits benchmarks like LULESH, 2DC and VctR.

According to the McPAT and CACTI models, RICH requires 2.45% more area than the baseline processor and introduces 3.8 % of power overhead. However, the performance improvement of 1.11× over COUP compensates for the increased power consumption of the RICH design. Consequently, RICH achieves better energy consumption than COUP. Another important improvement of RICH over COUP is the support for external pre-compiled libraries. Many scientific applications use mathematical libraries that implement algorithms with reductions, e.g. matrix multiplications. COUP requires modifying the ISA to mark the loads and stores belonging to the reduction operation, which requires access to the complete

source code to be compiled. RICH uses information about the reduction variable provided by the runtime system and does not require ISA modifications, thus supporting linking against pre-compiled algorithmic libraries.

5.5 Summary

This chapter introduces RICH, a proposal to accelerate the execution of reductions on modern processors. RICH improves the performance of vector-reductions while keeping well-performing support for reductions on scalars. RICH enables the programmer to select the optimal cache level where reductions take place. It relies on hardware, runtime system and OpenMP-compatible programming model extensions.

Extensive evaluation with reductions on vector variables show that RICH outperforms the atomics-based software technique in terms of execution speed on average by 1.8× and up to 6.0×. The energy-delay product is improved up to 96.8% (44.4% on average). Moreover, the total number of misses in the cache hierarchy is reduced by up to 96.6% (34.0% on average). RICH implementation requires only 2.4% additional silicon area and introduces a 3.8% power overhead.

RICH outperforms COUP, a state-of-the art hardware-based technique for reductions, by up to 1.38× (1.11× on average). Furthermore, thanks to its runtime-hardware interaction, RICH does not modify the ISA. Thus, it is compatible with applications that use routines with reductions present in pre-compiled mathematical and algorithmic libraries.

Chapter 6

Criticality-Driven Prioritization inside the Memory Hierarchy

This chapter describes PrioRAT, a memory request prioritization scheme that exploits runtime-level criticality information. It follows a holistic approach where the runtime system knowledge is used in hardware to drive the prioritization algorithm in shared on-chip memory hierarchy. Specifically, it exploits the notion of task criticality, considering that the faster execution of critical tasks leads to a better overall performance of a parallel code. The programmer uses the simple annotations in the programming model to specify the critical tasks. A source-to-source compiler translates these annotations into the function calls defined in the runtime system library. During the execution, the runtime system provides the underlying hardware with the information about the criticality of the currently running tasks. Then, on-chip hardware resources make use of this knowledge to prioritize the memory requests coming from the critical tasks. As a result, the critical tasks have their memory requests served faster, which reduces their duration and therefore, improves the performance of the whole parallel application by reducing the length of the critical path.

This proposal makes the following contributions:

- PrioRAT offers the programmer a simple mechanism to annotate criticality of tasks in task-based parallel codes. This feature is provided by extending the annotations in existing parallel programming models, such as OpenMP and OmpSs.

- It extends the shared on-chip components to take into account the criticality of the memory requests when making scheduling decisions. The task criticality knowledge is forwarded to the hardware by the runtime system library using the well-supported memory mapped registers, which preserves the processor's ISA.

- It extensively evaluates the performance of PrioRAT on a cycle-accurate architectural simulator using a set of characteristic workloads from the High Performance Computing (HPC) domain. PrioRAT outperforms the baseline system without prioritization by up to 30.3% in terms of execution speed (4.2% on average). The evaluation further analyzes in detail the impact of prioritization on the performance of tasks and the whole application.

6.1 Challenges in Prioritization Techniques

In order to overcome the issues caused by the increasing gap between processor and memory speeds, it is necessary to employ advanced techniques that fully exploit available memory resources. A higher memory throughput can be achieved by reordering DRAM commands [140, 149, 169]. Simple solutions, however, do not consider the fact that memory requests can be issued from different processor cores, which can lead to fairness issues. Solutions designed for multi-threaded processors aim to ensure a fair share of memory resources among all the threads [128, 129, 132]. Nevertheless, codes with an unbalanced workload distribution among threads often do not benefit from fair scheduling schemes. In such scenarios, a better performance can be achieved with schedulers that take into account request criticality [73].

However, such designs do not have the high-level notion of the critical path at the level of the whole application. Specifically, in certain parallel codes, it is possible to sacrifice the performance of non-critical tasks by giving priority to critical tasks in order to reduce the critical path and, thus, the overall execution time. Previous works on accelerating critical tasks have specific hardware requirements, such as heterogeneous cores [43] and support for dynamic voltage and frequency scaling [37]. Therefore, it is necessary to design a solution that is applicable to a wide range of modern processor architectures. This section explores the impact of memory request prioritization on accelerating the critical path in a parallel code.

6.1.1 Accelerating Critical Path by Memory Request Prioritization: A Proof of Concept

To illustrate the effects of prioritization of critical tasks, we develop a synthetic application that performs a strided access to an array. The stride is a configurable parameter that is used to indirectly tune the pressure on the caches and main memory by modifying the reuse of the accessed cache lines. The application is split into tasks and each task is designed to access its portion of the input array with the same stride. Tasks are split in two groups: (i) The first group

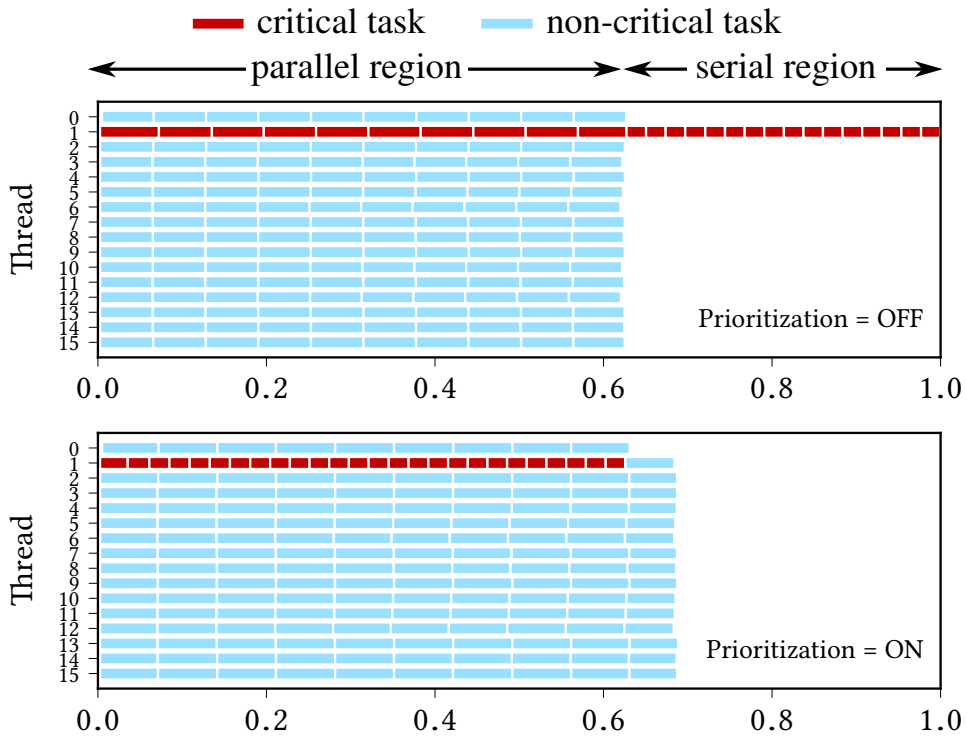


Figure 6.1: The effects of the prioritization on the execution of a parallel code.
Execution traces of the synthetic benchmark in the baseline configuration without prioritization (above) and the configuration with prioritization (below).

of tasks is artificially serialized in order to simulate a critical path. (ii) Tasks belonging to the second group can run in parallel with all tasks from the first and the second group.

The benchmark is configured to run with 26 critical and 150 non-critical tasks, each accessing an array of 256 KB with a stride of 16. This is equivalent to one access per cache line and achieves the highest memory contention for this benchmark in our simulated environment. Two executions of such configuration are simulated on a cycle-accurate simulator configured to simulate a 16-core CMP with a three-level cache hierarchy and main memory (Chapter 3 describes the experimental setup in detail). The first run is performed on a baseline system without prioritization, while the second configuration prioritizes the requests issued by the critical tasks. Figure 6.1 shows the parallel traces of these two executions. Each trace displays the tasks and their duration during the execution. For each task we show the time interval when it executes on the x-axis and the thread where it executes on the y-axis. Time is normalized to the execution on the baseline configuration and both traces have the same time scale. Critical and non-critical tasks are highlighted with different colors.

Without prioritization (see Figure 6.1, top), the serialization of critical tasks and its negative impact on the total execution time are clearly observed. During the first 62% of the execution,

the critical tasks execute slower compared to the critical tasks in the last third of execution. This is a result of the high memory contention caused by the concurrent execution of many tasks in the parallel region.

The bottom part of Figure 6.1 shows the trace when memory requests issued by critical tasks are given a higher priority in the shared on-chip resources. The effects of the prioritization are clearly manifested through the reduced duration of the critical tasks. The non-critical tasks execute slower, but the whole application finishes faster as the serialized tasks are no longer in the critical path. This example clearly demonstrates the importance of having a high-level notion of the application within the hardware as it enables better decisions at the hardware level. Taking these conclusions into consideration, we propose a solution that exploits the runtime system information about a task-based parallel code to guide prioritization of memory requests inside the on-chip memory hierarchy.

6.2 PrioRAT: Criticality-Driven Prioritization inside Memory Hierarchy

PrioRAT is a holistic approach for prioritization of memory requests in the on-chip memory hierarchy driven by the application-specific information available in the runtime system library. The programmer uses simple pragma directives added to a parallel task-based programming model to annotate task types belonging to the critical path of an application. The runtime system provides the underlying hardware with the information about task criticality specified by the programmer. Then, each core in the processor makes use of this knowledge about the task criticality to define the priority level of issued memory requests. Finally, minimal micro-architectural extensions in the shared components of the on-chip memory hierarchy are required to prioritize memory requests depending on the priority level of each request.

Thus, PrioRAT relies on the following extensions:

- The programming model extensions to enable a programmer to annotate critical tasks,
- The runtime system support to forward the user-specified information to the hardware,
- Micro-architectural extensions to prioritize memory requests inside the processor, specifically in the on-chip interconnection network, the last-level cache and the memory controller.

The remaining of this section provides a detailed description of the mentioned extensions.

6.2.1 Programming Model and Runtime System Support

PrioRAT is built on top of the existing parallel task-based programming model, OpenMP [137]. OpenMP is a directive-based programming model, where a programmer defines units of parallelism, such as tasks and loop iterations, and annotates all the properties that are necessary for the correct synchronization. Then, the runtime system library handles the scheduling of the defined tasks and loops on a multi-threaded machine by respecting the dependencies between tasks and both implicit and explicit synchronization primitives.

Internally, the runtime system manages tasks using a task dependency graph. This data structure is constructed by following the dependencies between tasks based on the directives annotated by the programmer. In the graph, each task has a set of predecessors, i.e., tasks and synchronization events, that need to complete before the task can be scheduled for execution. The runtime system can analyze this graph to identify potential critical paths in the execution of a parallel code.

To express task criticality, the existing OpenMP parallel task pragma is extended as follows:

pragma omp task [**critical**]

The OpenMP standard already includes *priority* annotation in the *parallel task* clause which is used as a hint to the task scheduler. The extensions proposed in PrioRAT, *critical*, affects the prioritization inside on-chip resources once the task is scheduled for the execution. In a real system, these two annotations can be merged. In this thesis, the domains of task scheduling by the runtime system and the prioritization of requests in hardware are separated.

In an existing task-based parallel code, the programmer needs to add criticality annotations to identify critical task types. Then, a source-to-source compiler translates these directives to function calls to the runtime system library that implements the routines defined in the programming model. During the execution of a parallel application, the runtime system library forwards the criticality information to the underlying hardware. This action is performed via memory-mapped registers before the user code of each task starts executing. The hardware mechanisms that exploit the notion of task criticality to prioritize memory requests are explained in detail in the following sections.

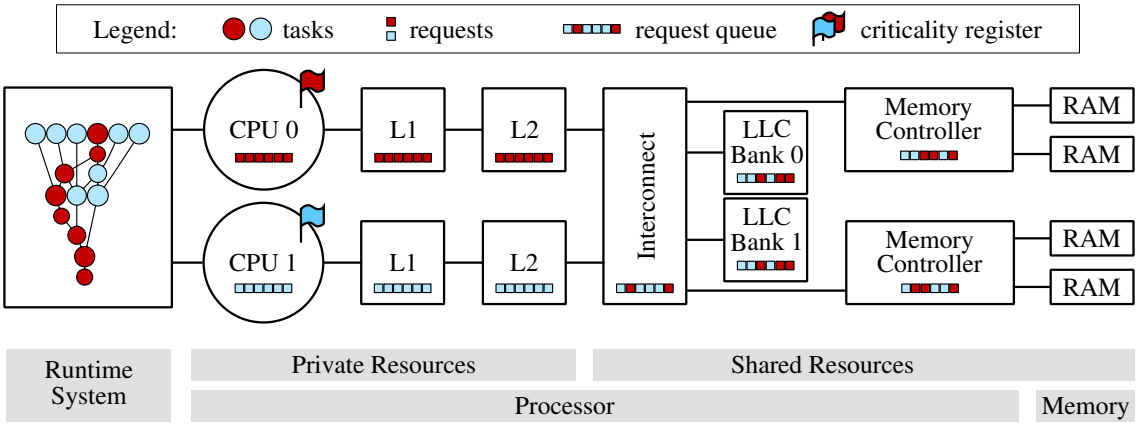


Figure 6.2: Overview of a dual-core system implementing PrioRAT.
Core 0 executes a critical task, while core 1 executes a non-critical task. The dark red color corresponds to the critical tasks and the requests they issue. Non-critical tasks and their memory requests are colored in light blue.

6.2.2 Hardware Extensions for Request Prioritization

This section is dedicated to the description of the micro-architectural extensions necessary to implement PrioRAT. Figure 6.2 shows a runtime system and a dual-core processor that implements PrioRAT, connected to the main memory. Within the processor, we identify the private resources, i.e., the core, the L1 and L2 private caches, and shared resources, i.e., the on-chip interconnection network connecting private caches to the shared last-level cache (LLC), the LLC itself, and the memory controllers. For the sake of simplicity, it is assumed that each core executes a single task at any given moment. Section 6.2.3.1 discusses the implications of designs that implement simultaneous multi-threading (SMT), which are able to have mixed criticality workloads on the single physical core.

6.2.2.1 Awareness of Task Criticality in the Core

As explained in Section 6.2.1, the runtime system library provides the hardware with the criticality of each task before its execution. Each core exposes a memory-mapped register to the software, which is referred to as *criticality register*. When issuing a request to the L1 cache, a core flags the request with criticality information stored in the *criticality register*. This information is carried with the request along its way through the memory hierarchy. Since the private on-chip components process requests only from a single task instance, they do not need to consider the request's priority. However, once the request arrives to shared components, the first one being the interconnect, its priority is taken into account when the components decide the ordering of the requests for processing.

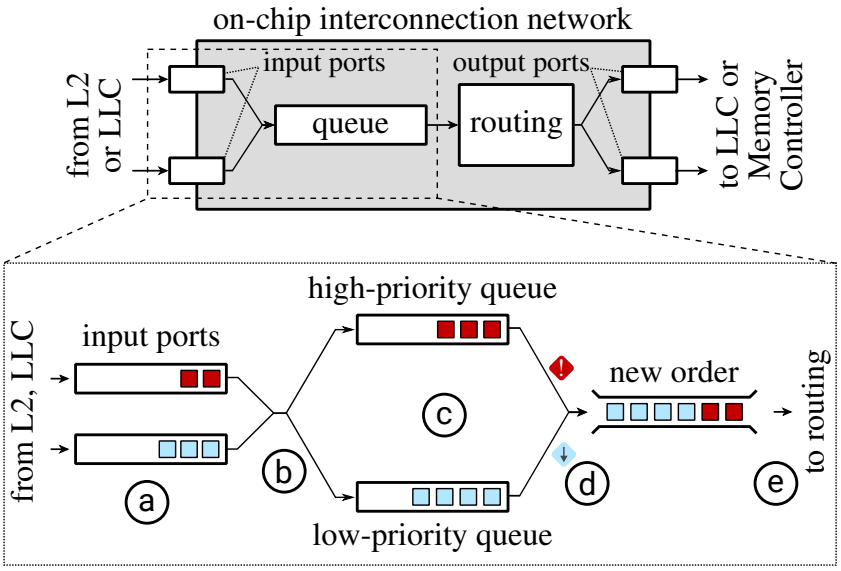


Figure 6.3: Request prioritization inside the on-chip interconnection network.
The top image shows the micro-architecture of a generic on-chip interconnect. The section annotated with the dashed rectangle is presented in detail in the bottom image.

To implement request prioritization inside the shared on-chip resources, the queues that hold incoming requests before they get processed are extended. PrioRAT implements a double-queue design, where each queue holds either high or low priority requests, as shown in Figures 6.3 and 6.4. When selecting the next request for processing, the requests in the high-priority queue are preferred over the requests from the low-priority queue. The following sections provide the specific design considerations for each relevant shared on-chip component.

6.2.2.2 Prioritization inside the Interconnect Bus

Figure 6.3 shows the micro-architecture of the interconnect network on chip whose purpose is to route the requests among the L2 and the L3 caches and the memory controllers. Memory requests come into the input ports ⓐ and get sorted ⓑ into the appropriate priority queue depending on their criticality ⓒ. The ordering of the requests with the same criticality follows *first in-first out* algorithm. When sending requests to the destination component ⓔ, the logic gives priority to critical requests ⓓ.

6.2.2.3 Prioritization inside the Shared Last-Level Cache

The LLC cache stores the requests coming from the L2 caches in a queue before they get to access the tag and data arrays. To implement PrioRAT, the queues are extended as shown in Figure 6.4. This figure shows a group of requests with mixed criticality as they travel through

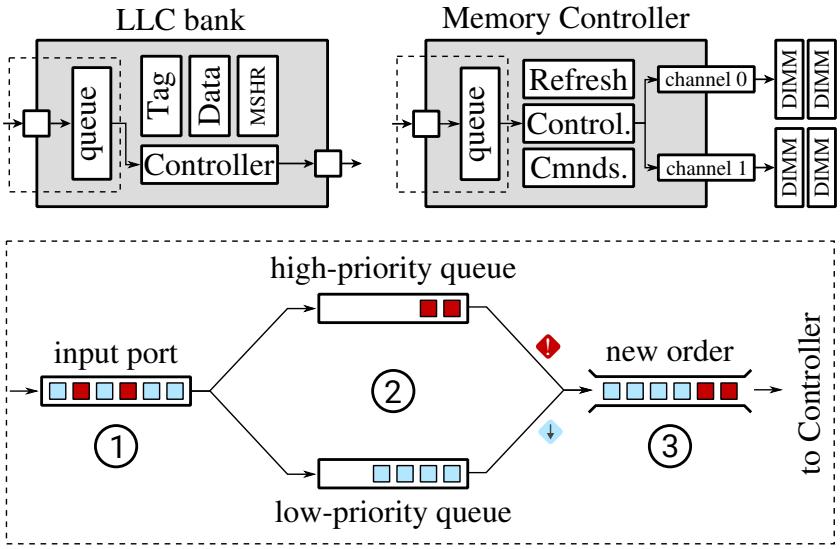


Figure 6.4: Priority queue inside the last-level cache and the memory controller.
The top images show the micro-architecture of the last-level cache bank and the memory controller. The sections annotated with the dashed rectangle are presented in detail in the bottom image. Both the LLC and the memory controller have the same queue implementation.

the queues. The requests are processed in order of arrival at the input port of the cache ① and they get sorted into the two queues depending on their criticality ②. Requests from the high-priority queue are served before other requests ③.

Multiple modern processors implement sliced or banked LLC caches, where each slice contains a subset of all cache lines. Such designs offer a higher throughput as accesses to different banks can be parallelized. Moreover, they exhibit a better scalability in many-core chips. PrioRAT is agnostic to the internal design of a cache. In processor designs where each cache slice handles requests independently, PrioRAT introduces the prioritized double-queue for each slice.

6.2.2.4 Prioritization inside the Memory Controller

In order to take into account the requests' criticality in the memory controller, the existing request scheduling policy needs to be extended. PrioRAT is independent of the baseline scheduling policy as it only adds another sorting criterion. However, in order to achieve a better performance, an architect needs to also take into consideration the design details of DRAM chips, such as bank-level parallelism, row buffer locality, etc.

To illustrate this fact, one can analyze an existing memory controller scheduling policy such as *First-Ready, First-Come, First-Served* (FR-FCFS). This policy schedules first the request that hit in the already open rows in the DRAM chip. If no such request exists, the ordering is

done in a *First-Come, First-Served* (FCFS) manner. PrioRAT augments this policy with the task criticality information. To preserve the performance benefits of exploiting row hits, the criticality is considered only after there is no request satisfying the row hit condition. For the requests of the same criticality, FCFS ordering is used.

In order to offer higher memory bandwidth to the cores, many modern processors are equipped with multiple memory controllers. The PrioRAT design does not require any synchronization between memory controllers as the prioritization is done independently in each controller. Thus, PrioRAT can be directly applied to such designs without any modification.

6.2.3 Discussion

This section discusses the interaction of the proposed design with various features in the modern computers and the associated parallel programming models.

6.2.3.1 Support for Simultaneous Multi-Threading (SMT)

Modern processors implement support for simultaneous multi-threading in order to better utilize the resources of a superscalar out-of-order core. In such systems, in general, one physical core can execute two different threads. In the context of this work this means that two tasks of different criticality can share the same core. To ensure the correct prioritization of critical requests, the input queue of the L1 cache is extended in the same way as described for the LLC in Section 6.2.2.3. The core is also equipped with a *criticality register* per hardware thread.

6.2.3.2 On-Chip Interconnect Design Implications

Processors employ different designs of on-chip networks depending on design factors such as number of cores, desired bandwidth, hardware and power overheads, etc. Section 6.2.2, describes the micro-architectural implementation of prioritization on an abstract model of the interconnect that consists of an input queue and routing logic. This section further explores the design implications of various specific network designs.

A point-to-point interconnect is the simplest network in terms of routing complexity, as each node is directly connected to every other node. Since such networks do not have active components (i.e., routers), prioritization is done at the receiving component (i.e., the LLC or the memory controller). Networks with ring topology [49] can implement prioritization either before the output port of the transmitting component or at the receiving component. More

complex ring topologies, such as H-Ring [15, 183], perform the prioritization at the nodes connected to multiple rings.

More complex topologies, such as two-dimensional (2D) mesh, 2D torus and their higher-dimensional counterparts [49] use on-chip routers to direct a message to the correct node. In these network implementations, the prioritization of requests is performed inside the routers, as explained in Section 6.2.2.

6.2.3.3 Hybrid Memory Systems

In recent years, computing systems with hybrid memory designs have appeared on the market. They generally combine DRAM chips with the high-speed alternatives, such as HBM [94], or non-volatile memories, such as PCM [175]. PrioRAT does not depend on memory technologies used in the system as it extends only the queuing logic inside the memory controller. In order to implement prioritization at the memory controller level, the only requirement is to extend the interface to the processor by a criticality bit that carries the request's criticality.

6.2.3.4 Multi-Socket and Multi-Node Support

Section 6.2.2 explains the design of PrioRAT on a single-socket architecture. PrioRAT can also be supported on multi-socked systems with a shared address space. To enable prioritization of requests coming from another socket, these requests need to carry the criticality information. Therefore, the interconnection interface between two sockets needs to be extended by a single bit that carries the criticality of requests.

Modern high-performance machines are, in general, designed as clusters that consist of multiple nodes which communicate with each other using a message-passing interface. In such systems, PrioRAT can be deployed to the individual nodes without modifications. In addition, PrioRAT can be extended to support prioritization of the messages between nodes by (i) adding support to annotate the criticality of a message to the calls to the message-passing library, and (ii) extending the switching hardware to take into account the criticality information.

6.2.3.5 Programming Model Support

PrioRAT is built on top of OpenMP, the most widely used shared-memory parallel programming model. Other task-based programming models, such as Chapel [38], Charm++ [97] and Cilk [82], can be extended to support PrioRAT. Specifically, task-spawning constructs in these languages can be extended to allow the annotation of critical tasks in a similar way as presented

in Section 6.2.1. The corresponding compilers translate the annotations to the instructions that forward the criticality data to the processor. This interface is implemented via memory-mapped registers and can be accessed either directly or through a system call in the operating systems that implement such a wrapper for the bare interface.

6.2.4 Combining priority and criticality annotations

OpenMP and OmpSs allow a programmer to specify the priority for a task using *priority (priority-value)* annotation, where priority-value is a non-negative integer. The *priority-value* is used to drive the task scheduling policy of the associated runtime system. The following paragraphs analyze the utility of the priority annotations in prioritization of memory requests in the on-chip memory hierarchy.

PrioRAT uses the criticality of the tasks to prioritize their memory requests. The criticality values are binary, i.e., critical or non-critical. This is reflected in the design of the hardware queues, which consist of low- and high-priority queues. PrioRAT can be adapted to consider the information provided in the *priority* annotations. The existing design of the interface between the runtime system and the hardware is applicable to this case as well. That is, the priority value can be passed to the hardware in the same way as the task criticality value. On the hardware side, such design would require a different priority queue implementation, since the *priority-value* can take any non-negative integer value. A possible implementation consists of a single queue with a modified insertion algorithm that follows the ordering defined by the requests' priority. A simpler variation to this design groups the priority values into several bins. For example, a three-level grouping would result in low-, medium- and high-priority requests. In this case, the cut-off value may be dynamic and defined by the programmer or the runtime system.

6.3 Evaluation

For the evaluation of the performance of PrioRAT, a system without request prioritization is selected as the baseline architecture. The evaluation is performed on the simulating infrastructure described in Section 3.1 using the benchmarks presented in Section 3.2. The input sizes for every benchmark is shown in Table 3.6. The evaluation starts with a high-level analysis of performance gains achieved by PrioRAT compared to the baseline configuration. In addition, the effects of the prioritization on the duration of tasks are explored. The analysis of the service time of memory requests provides further insights into the performance of PrioRAT. Section 6.3.2

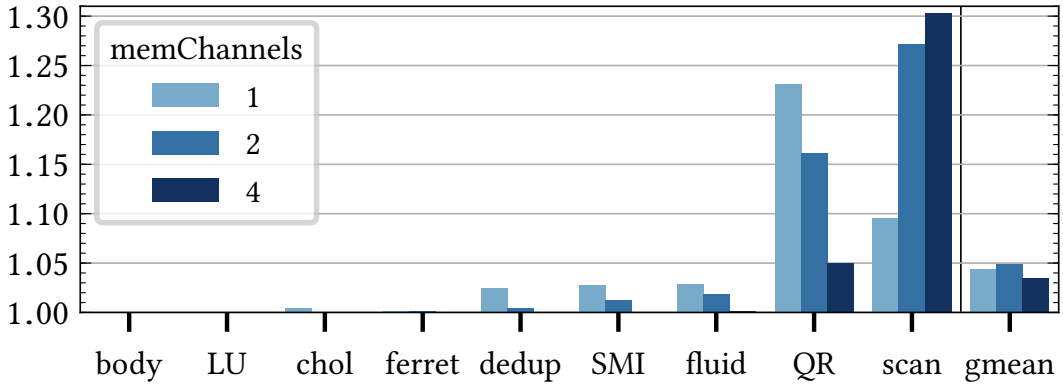


Figure 6.5: Speedup of PrioRAT compared to the baseline, for different number of memory channels.

examines the impact of the memory traffic intensity on the performance. Finally, Section 6.3.3 shows how parameters such as the LLC size and memory latency impact performance of PrioRAT.

6.3.1 Performance Evaluation

Figure 6.5 shows the speedups achieved by PrioRAT compared to the baseline configuration without prioritization. The performance benefits are shown for all evaluated benchmarks and for different number of memory channels. The codes achieving highest speedups are scan and QR, which perform up to 1.3× and 1.23× faster, respectively. An interesting observation is that these two codes obtain the best performance for different memory configurations, i.e., scan performs the best with more memory channels. On the contrary, QR achieves best performance benefits for lower number of memory channels. The performance depends on the traffic rate at the memory level coming both from critical and non-critical tasks. These effects are evaluated in detail in Section 6.3.2.

To evaluate the impact of request prioritization on the internal execution of each application, we record the duration of each task instance when running on the baseline system, as well as on the system that implements PrioRAT. The tasks are grouped in two groups depending on their criticality. The average duration of the tasks from each of the groups is compared between the baseline and PrioRAT configurations. These data are displayed in Figure 6.6, for all evaluated benchmarks and systems with 1, 2 and 4 memory channels. In all the benchmarks, PrioRAT reduces the duration of the critical tasks. This reduction in duration comes from reduced latency of the memory requests, as they are given the priority over the requests coming from

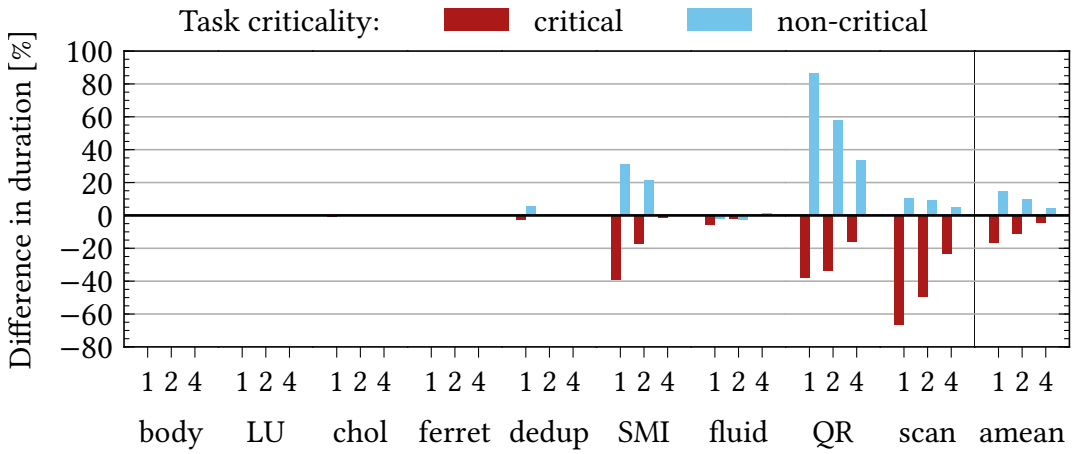


Figure 6.6: Change in the average duration of tasks per task criticality compared to the baseline configuration. Data is shown for configurations with 1, 2 and 4 memory channels.

the non-critical tasks. On the other hand, this causes the increase of duration of non-critical tasks compared to the baseline configuration.

The highest impact of prioritization on the task duration is observed for scan and QR. The average duration of critical tasks in scan is reduced up to 66.7%. On the other hand, the performance of the non-critical tasks is decreased up to 10.7%. Prioritization does not have a significant negative effect on the duration of non-critical tasks because only 13.8% of all memory traffic comes from critical tasks.

In case of QR, the performance loss suffered by the non-critical tasks is considerably greater than the speedup of critical tasks. This is due to the fact that critical requests represent the majority of all requests. Therefore, effectively increasing the latency of non-critical requests by their de-prioritization has a significant effect on the duration of non-critical tasks. However, the critical path is still accelerated which results in performance improvements of PrioRAT compared to the baseline, as seen in Figure 6.5.

In order to further analyze the internal behavior of the tasks, we focus on the memory requests issued by processor cores. To evaluate the impact on the prioritization on the latency of these requests, the request round-trip times is analyzed. For each memory reference, we measure the time that passes between issuing the request to the L1 cache until the acknowledgment is received by the core. Finally, we compare the measurements obtained with PrioRAT configuration with the baseline execution. Figure 6.7 shows the results of these measurements, only for requests that get serviced by the main memory, i.e., the requests that miss in the last-level cache. The requests hitting in one of the caches are omitted from this analysis in order to remove the influence of different access patterns that the tasks may have.

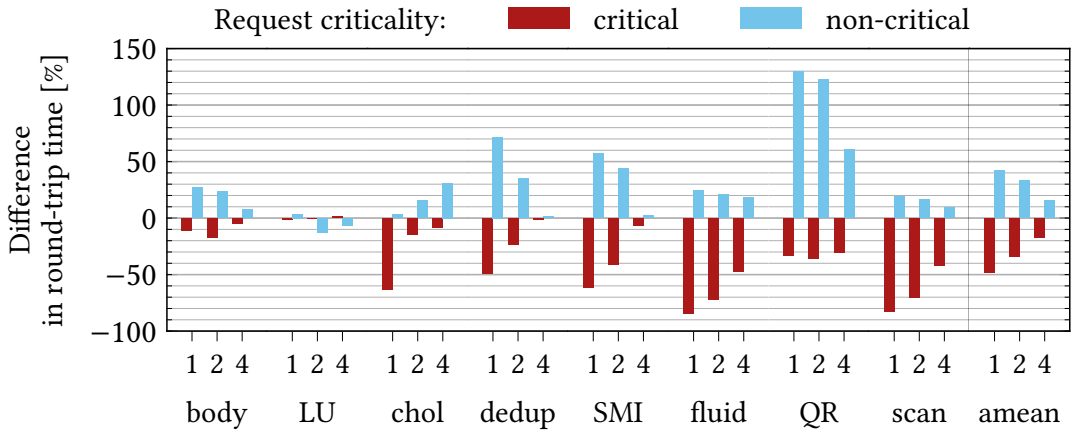


Figure 6.7: Difference in the memory request round-trip time in PrioRAT compared to the baseline. For each benchmark, x-axis shows configurations with different number of memory channels.

The results clearly show the effects of prioritization on the request round-trip time. The critical requests take on average 48.4%, 34.5% and 17.4% less cycles to get served for configurations with 1, 2 and 4 memory channels, respectively, compared to the baseline configuration. We also observe that with an increasing the number of memory channels, the impact of prioritization on the request service time is reduced. This is further elaborated in Section 6.3.2. The largest round-trip time improvements are observed for fluidanimate and scan. The service time of non-critical requests in QR suffers the most among all applications, which manifests itself through significant performance degradation of non-critical tasks as explained earlier in this section and shown in Figure 6.6.

Most applications exhibit a notable sensitivity of the critical requests' latency improvement to available memory bandwidth. These codes also observe a significant contention in shared resources in the memory hierarchy. Therefore, increasing the memory bandwidth reduces saturation, which reduces the baseline round-trip time. This effect is further explained in Section 6.3.2. For example, in the case of SMI, one of the critical tasks performs a matrix-matrix multiplication, which is the most memory intensive kernel in that application. On the other hand, body, LU and QR have the opposite behavior. In these codes, the memory load caused just by critical tasks is not big enough to saturate the memory links. Similarly to the critical tasks, the latency of the non-critical tasks is also sensitive to the number of channels in all the cases.

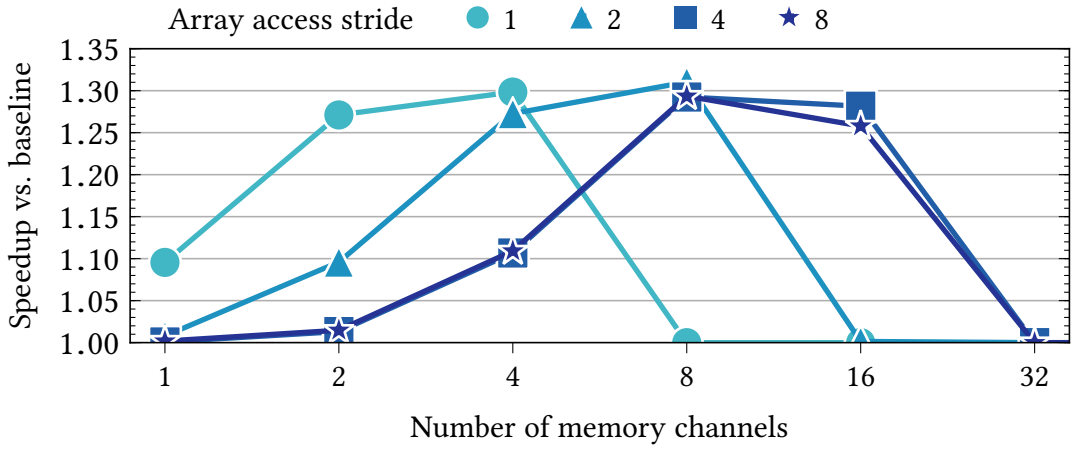


Figure 6.8: Impact of access strides on the performance of PrioRAT running scan benchmark for different number of memory channels.

6.3.2 Performance Impact of Memory Traffic Intensity

This section evaluates the effect of memory traffic intensity on the speedup achieved by PrioRAT. Scan is selected for the analysis due to its simplicity and the support for a precise control of the memory traffic intensity. To model different traffic intensities at the memory level, we adjust the stride of the array accesses. A stride of 1 means that every element in the input array is accessed. Since each element of the array has the size of 4B, a configuration with a stride of 16 produces one access per 64B cache line. Therefore, using larger strides results in less reuse of the cache lines and increased memory traffic. We perform this study for different number of memory channels ranging from 1 to 32, which is equivalent to the effective bandwidths of 4.3 GB/s to 137.6 GB/s.

Figure 6.8 shows the results of this analysis for strides ranging from 1 to 8. We do not show the results for a stride of 16 elements as this configuration has the same behavior as the executions with stride values 4 and 8. The results demonstrate that the curves corresponding to different stride values have a similar bell-like shape. For each stride setting, there is an optimal memory bandwidth where PrioRAT achieves the highest speedup. For example, for a stride values of 1 and 2, the highest performance improvements are achieved when using 4 and 8 memory channels, respectively. Once the bandwidth values move farther from the optimal, the speedup gradually drops to 1. In order to explain the reason behind the shape of the curve and its horizontal offset depending on the stride, we analyze two factors that limit the speedup that PrioRAT can achieve: (i) the theoretical speedup limit and (ii) impact of the memory request prioritization on the speedup.

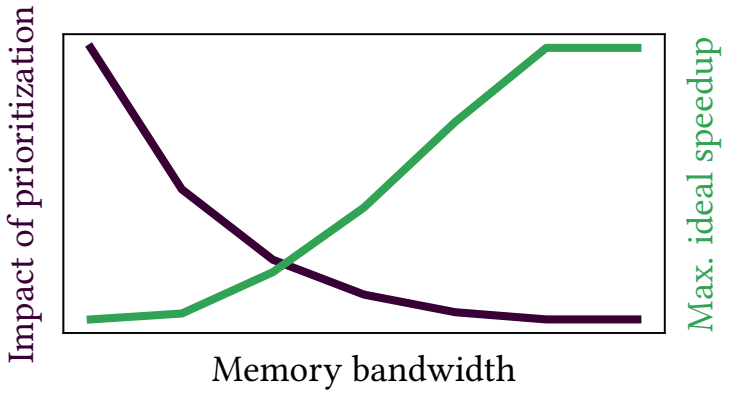


Figure 6.9: Impact of memory bandwidth on factors that control the overall achieved speedup: speedup achievable by request prioritization impact (left y-axis) and speedup achievable by critical path acceleration (right y-axis), for scan benchmark using stride of 8 elements.
The impact of prioritization is measured as a ratio of the durations of non-critical and critical requests. The maximal ideal speedup is calculated as a speedup achieved by setting the duration of critical tasks to zero. These metrics are displayed in arbitrary units. Memory bandwidth represents the total available bandwidth and linearly depends on the number of memory channels.

The relation between these two factors and the memory bandwidth is shown in Figure 6.9. On the right y-axis we show the maximal achievable speedup for the benchmark depending on the memory bandwidth, which is shown on the x-axis. To aid the explanation of this figure, we refer to the top trace in Figure 6.1, which represents the execution of scan on a baseline processor. In particular, we note that the execution consists of a parallel region, where both non-critical and critical tasks execute simultaneously, and a serial region, where only the remaining critical tasks execute.

The maximal ideal speedup can be calculated as follows:

$$Spd_{max} = \frac{t(whole.app)}{t(par)} = \frac{t(par)+t(seq)}{t(par)} = 1 + \frac{t(seq)}{t(par)},$$

where $t(x)$ represents a duration of the region x, and $whole.app$, par and seq correspond to the whole application, its parallel region and serial region, respectively. A larger speedup is achieved when the ratio of the durations of sequential and parallel regions is larger.

For a low memory bandwidth, the memory contention in the parallel region is high, which slows down the tasks in this region. As a consequence, the duration of the parallel region is significantly larger compared to the duration of the serial region, which results in low speedup values. On the other hand, when the bandwidth increases, the memory contention decreases, which reduces the length of the parallel region compared to the duration of the whole application.

When bandwidth increases over a certain threshold, the achieved speedup stagnates as the contention is reduced to the zero.

Another factor that affects the achievable performance improvements is the impact of the memory request prioritization on the performance of a task. When memory contention is high, scheduling critical requests before the others can improve the performance of a critical task by reducing the service time of the critical requests. This effect diminishes as the memory contention is reduced because the prioritization cannot reduce the request latency in non-saturated systems.

The conclusion of this analysis, in the case of scan, is that when the impact of the prioritization is the highest, the effects of reduced execution time of critical tasks has low impact on overall speedup. Similarly, when accelerating critical tasks can bring the most overall performance benefits, this cannot be done by prioritization of memory requests. Between those extreme scenarios, there is an optimal point where PrioRAT achieves the highest speedups, ranging from 25.8% to 31.0%, as shown in Figure 6.8.

From the results presented in Figure 6.8, it can observed that the optimal memory bandwidth increases as the stride values change from 1 to 4. This is an expected behavior due to the fact that speedup depends on the memory contention, which is proportional to the stride. However, configurations with strides 4 and 8 behave similarly as they exhibit equal contention. The CPU resources become a bottleneck when request issue rate increases over some threshold.

6.3.3 Performance Impact of the LLC Size and Memory Latency

In this section we analyze how different parameters of the simulated system impact the performance of PrioRAT.

6.3.3.1 The Last-level Cache Size

Figure 6.10 shows the speedup of PrioRAT compared to the baseline for three selected benchmarks: cholesky, SMI and scan. For cholesky and SMI, PrioRAT achieves higher speedups with smaller LLC sizes. In addition, the achieved performance benefits are, in general, higher for configurations with smaller number of memory channels. Both effects are caused by the increased memory contention due to larger miss rate of smaller caches and lower available memory bandwidth. In such scenarios, the impact of request prioritization is higher.

In contrast, the performance of scan is not sensitive to the cache size because this benchmark does not reuse the arrays and therefore does not benefit from caching. Similarly to two other

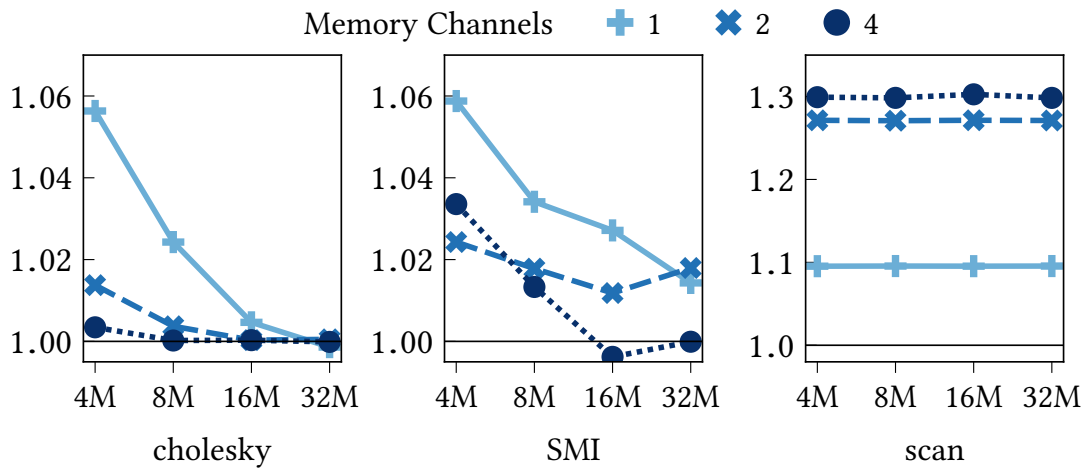


Figure 6.10: Impact of the LLC size on the speedup achieved by PrioRAT compared to the baseline. For each benchmark, x-axis shows the size of the LLC and the y-axis denotes achieved speedup.

benchmarks, scan also exhibits sensitivity to the available memory bandwidth. However, in case of this code, PrioRAT achieves higher speedups for configurations with more memory channels. This behavior is explained in Section 6.3.2. The data points for the LLC size of 16 MB in Figure 6.10 correspond to the points for the stride value of 1 in Figure 6.8.

6.3.3.2 Memory Latency

Figure 6.11 shows the speedup PrioRAT achieves against the baseline configuration for different memory latencies and number of memory channels. The results for cholesky and SMI show that, in general, the performance improvements are higher for larger memory latencies. This is the case because the configurations with higher memory latencies take more time to process each request, which increases memory contention and, therefore, the impact of prioritization on the performance of critical tasks. When using one and two memory channels, scan exhibits different behavior than cholesky and SMI, i.e., the speedup is higher for lower memory latencies. Scan benchmark has a higher memory contention than other evaluated benchmarks. The region for the optimal speedups is achieved when the memory contention decreases, as explained in Section 6.3.2. Both the memory latency and the number of memory channels directly impact memory contention, and therefore affect the achieved performance improvements.

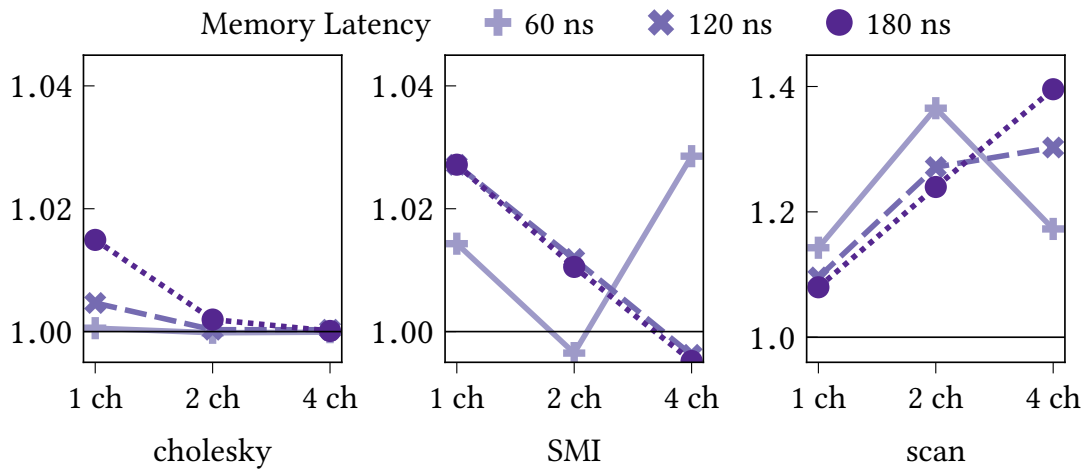


Figure 6.11: Impact of the memory latency on the speedup of PrioRAT versus the baseline. X-axis shows the number of memory channels.

6.4 Summary

This chapter presents PrioRAT, a runtime-assisted approach for prioritization of memory requests in the processor. PrioRAT provides a programmer with a simple interface to define the criticality of tasks in a task-based parallel code. The runtime system library is responsible for forwarding this information to the underlying hardware. The processor uses the knowledge of the task criticality to guide the prioritization of the memory requests inside the shared on-chip memory hierarchy.

The evaluation of PrioRAT is performed on a set of representative codes from the domain of high-performance computing. The extensive evaluation shows that PrioRAT outperforms the baseline system by up to 30.3% in terms of the execution time. Further analysis demonstrates high impact of the prioritization on the request service time and duration of the tasks. Finally, we revisit the main factors that impact the performance of an application in the context of this work, such as memory contention.

PrioRAT exploits the high-level information about the application that is not available in the state-of-the-art proposals. It is demonstrated that the availability of such knowledge inside the on-chip components can lead to notable performance improvements.

Chapter 7

Conclusions

The work presented in this thesis has demonstrated the utility of the runtime system in the design of on-chip memory hierarchy. This chapter outlines the main goals and the contributions of this thesis. The list of publications is shown at the end of the chapter.

7.1 Thesis Goals and Contributions

The current trends in the design of computer systems are followed by many challenges. This thesis focuses specifically on two well-established issues in the field of high-performance computing, the Memory Wall and the Programmability Wall. The Memory Wall is a result of the ever-increasing gap between the processor and memory speeds. What makes this fact important is the negative performance impact of slow memory accesses. In order to solve this and many other issues, and enabled by the increasing number of transistors on chip, the design of recent processors is getting more complex with each generation. This inherently makes programming such systems more difficult, which is a phenomenon known as the Programmability Wall.

The work presented in this thesis consists of three independent contributions that tackle the described challenges in the context of on-chip memory hierarchies and task-based parallel programming models. Each contribution introduces an optimization implemented in the memory hierarchy. Common to all proposals is the interaction with the runtime system library. This holistic design brings two benefits: (i) hardware design can exploit the high-level knowledge of an application available at the runtime system; (ii) it provides complete solutions covering both hardware and software stack, which makes such systems easier to program and more attractive to adopt. The following sections provide the brief conclusions of each contribution presented in this thesis.

7.1.1 Runtime-Aware Shared Last-Level Cache Insertion Policies

Processor caches represent an important component of the microprocessor and their careful design is of paramount importance for achieving a good overall performance. Shared caches introduce additional challenges. The memory access patterns at the shared cache level are complex as they are a combination of patterns coming from different processor cores. It is necessary to apply sophisticated techniques in order to improve the performance of shared caches. The first proposal of this thesis targets a shared last-level cache and introduces cache insertion policies aware of access patterns on both task and memory region level. The proposal consists of two insertion policies. The first, TTIP, uses a parameter per task type that controls a probabilistic insertion, by which it effectively achieves a fine allocation of cache space to the co-running tasks. TTIP is a dynamic policy that automatically selects the best performing parameter per task during the execution and is, therefore, able to adapt to the changes of application's behavior. The second policy, DTIP, takes into account the task-data dependency type when performing the insertion of a cache line. Its design is based on the observation that different dependency types observe distinct long-term access patterns. These policies outperform both the industry standard, LRU policy, and the state-of-the-art replacement policy, DRRIP. The conclusion of the work on the first proposal is that the runtime-system-level information about a parallel code can be of a great utility in the design of a replacement policy for a shared last-level cache.

7.1.2 Reductions in the Cache Hierarchy

Reductions constitute a frequent algorithmic pattern in high-performance and scientific computing. Sophisticated techniques are needed to ensure their correct and scalable concurrent execution on modern processors. Reductions on large arrays represent the most demanding case where traditional approaches are not always applicable due to low performance scalability. The second proposal of this thesis addresses these challenges by proposing a runtime-assisted solution for reductions in the cache hierarchy, RICH, that relies on architectural and parallel programming model extensions. RICH updates the reduction variable directly in the cache hierarchy with the help of added in-cache functional units. It relies on programming model extensions that allow a programmer to easily annotate the reduction variables in an existing parallel code. The runtime system library serves as an interface between the parallel application and the hardware. Experiments show that RICH achieves the performance improvements

of 1.11× on average, compared to the state-of-the-art hardware-based approaches, while it introduces 2.4% area and 3.8% power overhead.

7.1.3 Criticality-Driven Prioritization inside the Memory Hierarchy

Producing a perfectly-balanced parallel code is not achievable in practice. There are many factors that cause suboptimal load balancing. Some limitations are imposed by an algorithm itself, where the work cannot be equally distributed among all parallel threads. Even with perfect work separation, various factors during the execution can introduce issues with load balancing. In order to efficiently use the computing resources of the parallel systems, it is necessary to minimize these issues. Many solutions aim to improve load balancing, ranging from compiler and runtime system optimizations to hardware-based approaches. The third contribution of this thesis accelerates the critical parts of an application by prioritization of the corresponding memory requests inside the on-chip memory hierarchy, which can improve the load-balancing. The proposal relies on the user annotations of task criticality, which is enabled by simple extensions of the parallel programming model. The runtime system is responsible to pass the criticality information to the underlying hardware. The extensive evaluation has shown that the prioritization of memory requests can notably reduce the duration of critical tasks and, therefore, reduce the critical path of a parallel code. This technique can achieve up to 30.3% faster execution time compared to the baseline system without any acceleration of critical tasks. The conclusion of this work is that utilization of the criticality information at the application level can notably improve the performance with minimal hardware modifications.

7.2 Future Work

The work presented in this thesis opens new opportunities for the further research on the explored topics. This section provides a brief overview of potential future directions for investigation.

Dynamic Task type aware insertion policy. TTIP allows for several further improvements. The policy trainer can be optimized to blacklist poorly performing configurations from the future training phases to reduce the performance penalty of using such configurations. Furthermore, in order to maintain the adaptability to the changes in the behavior of application, the blacklisting can be reset after a certain number of training cycles. Moreover, short tasks or tasks that do not run many times during the execution can be excluded from the training process.

Dynamic dependency-type aware insertion policy. DTIP is a static policy that always uses the same configuration for a given dependency type. The policy can be improved by introducing a dynamic mechanism to adapt to the behavior of each application. Moreover, it is possible to exploit already present information about task-data dependencies to implement a precise access pattern estimator per memory region. Such knowledge about access patterns can provide better replacement policies.

Computation offloading to the memory hierarchy. The second proposal explores the benefits of executing reduction-like operations in the cache hierarchy. Offloading more complex operations increases pressure on the buses in the on-chip memory hierarchy. A possible solution for this problem is an implementation of more complex cores in the caches, memory controller and the main memory, which is already explored in previous works. However, the already present mechanisms for device-specific implementation of tasks inside the OmpSs programming model can be applied for offloading the eligible code segments to execute near memory.

Applying dynamic criticality knowledge on the prioritization of memory requests. The third proposal of this thesis introduces a mechanism to exploit user-provided criticality annotations to prioritize the critical requests in the memory hierarchy. The next improvement to this scheme is to exploit the already existing mechanisms for critical path detection during the execution. This approach would overcome an issue of the change of critical path when the original critical path is accelerated. In addition, it may be beneficial to discriminate the memory requests according to the issuing task type and the corresponding dependency type to which the requested address belongs. As shown in Chapter 4, this discrimination can be successfully applied to cache design. Further exploration of the opportunities given by the runtime system knowledge might lead to additional performance improvements.

Exploiting task criticality in the cache replacement policies. The first proposal exploits the knowledge about parallel tasks to optimize the design of the cache replacement policy. To extend this idea, we consider a cache replacement policy driven by task criticality. Cache lines corresponding to critical tasks may be given more priority in cache compared to other lines. This can be achieved by assigning higher re-reference interval values to such cache lines. A potential result of such design is a better cache performance of critical tasks and consequently, improved performance of critical tasks and the whole application.

7.3 Publications

This section lists the publications resulting from the work presented in this thesis, the related posters and the publicly available code.

Publications of the Thesis:

- **Dimić**, **V.**, Moretó, M., Casas, M., and Valero, M. "Runtime-Assisted Shared Cache Insertion Policies Based on Re-reference Intervals". In: *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*. Ed. by F. F. Rivera, T. F. Pena, and J. C. Cabaleiro. Vol. 10417. Lecture Notes in Computer Science. Springer, 2017, pp. 247–259. DOI: 10.1007/978-3-319-64203-1_18

- **Dimić**, **V.**, Moretó, M., Casas, M., Ciesko, J., and Valero, M. "RICH: Implementing Reductions in the Cache Hierarchy". In: *2020 International Conference on Supercomputing*. ICS '20. New York, NY, USA: ACM, 2020, 13 pages. DOI: 10.1145/3392717.3392736

- **Dimić**, **V.**, Moretó, M., Casas, M., and Valero, M. *PrioRAT: Criticality-Driven Prioritization Inside the On-Chip Memory Hierarchy*. (under submission)

Other Publications:

- **Dimić**, **V.** and Dziegielewska, O. "Metaheuristics Based Approach for Parallelizing Applications in On-Chip Multiprocessor". In: *ACACES'14: Advanced Computer Architecture and Compilation for Embedded Systems 2014 Poster Abstracts*. Poster abstract. 2014

- **Dimić**, **V.**, Moretó, M., Casas, M., and Valero, M. "Runtime-Assisted Shared Cache Insertion Policies Based on Re-reference Intervals". In: *RoMoL Final Workshop*. Poster without proceedings. Mar. 2018

Publicly Available Code:

- **Dimić**, **V.** *Reduction Benchmarks*. https://github.com/vdimic/reduction-benchmarks. 2020

- **Dimić**, **V.** *Array Scan Benchmark*. https://github.com/vdimic/scan-benchmark. 2020

Bibliography

[1] Advanced Micro Devices (AMD), Inc. *AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions*. Tech. rep. 24594. Advanced Micro Devices, May 2018.

[2] Advanced Micro Devices (AMD), Inc. *AMD EPYC ™ 7002 Series Processors: A New Standard for the Modern DataCenter*. Data Sheet. Apr. 2020.

[3] Aga, S., Jeloka, S., Subramaniyan, A., Narayanasamy, S., Blaauw, D., and Das, R. "Compute Caches". In: *2017 IEEE International Symposium on High Performance Computer Architecture*. HPCA '17. IEEE, 2017, pp. 481–492. DOI: 10.1109/HPCA.2017.21.

[4] Aggarwal, V., Sabharwal, Y., Garg, R., and Heidelberger, P. "HPCC RandomAccess benchmark for next generation supercomputers". In: *IEEE International Symposium on Parallel Distributed Processing*. IPDPS '09. May 2009, pp. 1–11. DOI: 10.1109/IPDPS.2009.5161019.

[5] Ahn, J., Hong, S., Yoo, S., Mutlu, O., and Choi, K. "A scalable processing-in-memory accelerator for parallel graph processing". In: *ACM/IEEE 42nd Annual International Symposium on Computer Architecture*. ISCA '15. June 2015, pp. 105–117. DOI: 10.1145/2749469.2750386.

[6] Ahn, J. H., Erez, M., and Dally, W. J. "Scatter-Add in Data Parallel Architectures". In: *Proceedings of the 11th Annual Symposium on High Performance Computer Architecture*. HPCA '05. 2005, pp. 132–142.

[7] Alameldeen, A. R. and Wood, D. A. "Adaptive cache compression for high-performance processors". In: *Proceedings of the 31st Annual International Symposium on Computer Architecture*. ISCA '04. June 2004, pp. 212–223. DOI: 10.1109/ISCA.2004.1310776.

[8] Alvarez, L., Casas, M., Labarta, J., Ayguadé, E., Valero, M., and Moreto, M. "Runtime-Guided Management of Stacked DRAM Memories in Task Parallel Programs". In: *Proceedings of the 32nd International Conference on Supercomputing*. ICS '18. ACM, 2018, pp. 218–228. ISBN: 978-1-4503-5783-8. DOI: 10.1145/3205289.3205312.

[9] Alvarez, L., Moretó, M., Casas, M., Castillo, E., Martorell, X., Labarta, J., Ayguadé, E., and Valero, M. "Runtime-Guided Management of Scratchpad Memories in Multicore Architectures". In: *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*. PACT '15. 2015, pp. 379–391. ISBN: 978-1-4673-9524-3. DOI: 10.1109/PACT.2015.26.

[10] Alvarez, L., Vilanova, L., Moreto, M., Casas, M., Gonzàlez, M., Martorell, X., Navarro, N., Ayguadé, E., and Valero, M. "Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. ACM, 2015, pp. 720–732. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750411.

[11] Arelakis, A., Dahlgren, F., and Stenstrom, P. "HyComp: A Hybrid Cache Compression Method for Selection of Data-type-specific Compression Methods". In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 38–49. ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830823.

[12] ARM. *big.LITTLE Technology: The Future of Mobile*. White Paper. 2013.

[13] *ARM® Cortex®-A75 Core. Technical Reference Manual*. 100403_0200_00_en. ARM. 2016.

[14] *ARM®Architecture Reference Manual Supplement. ARMv8, for the ARMv8-R AArch32 architecture profile*. ID033117. ARM. 2016.

[15] Ausavarungnirun, R., Fallin, C., Yu, X., Chang, K. K., Nazario, G., Das, R., Loh, G. H., and Mutlu, O. "Design and Evaluation of Hierarchical Rings with Deflection Routing". In: *Proceedings of the 26th International Symposium on Computer Architecture and High Performance Computing*. SBAC-PAD'14. IEEE, 2014, pp. 230–237. DOI: 10.1109/SBAC-PAD.2014.31.

[16] Babbage, C. "On the Mathematical Powers of the Calculating Engine". In: *The Origins of Digital Computers: Selected Papers*. Ed. by B. Randell. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 19–54. ISBN: 978-3-642-61812-3. DOI: 10.1007/978-3-642-61812-3_2.

[17] Balasubramonian, R., Kahng, A. B., Muralimanohar, N., Shafiee, A., and Srinivas, V. "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 14.2 (June 2017), 14:1–14:25. ISSN: 1544-3566. DOI: 10.1145/3085572.

[18] Barcelona Supercomputing Center. *Mercurium C/C++ source-to-source compiler*. May 2014.

[19] Barcelona Supercomputing Center. *Nanos++ Runtime Library*. May 2014.

[20] Barcelona Supercomputing Center. *OmpSs Specification*. Apr. 2014.

[21] Barcelona Supercomputing Center. *BSC Application Repository*. 2020. URL: https://pm.bsc.es/projects/bar (visited on 05/11/2020).

[22] Belady, L. A. "A study of replacement algorithms for a virtual-storage computer". In: *IBM Systems journal* 5.2 (1966), pp. 78–101.

[23] Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R. S., and Yelick, K. "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems". In: *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Techinal Representative* 15 (Jan. 2008).

[24] Beveridge, J. and Wiener, B. *Multithreading Applications in Win32: The Complete Guide to Threads*. Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 978-0-201-44234-2.

[25] Bienia, C. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, Jan. 2011.

[26] Blume, B., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D., Petersen, P., Pottenger, B., Rauchwerger, L., Tu, P., and Weatherford, S. "Polaris: The Next Generation in Parallelizing Compilers". In: *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*. LCPC'94. Berlin/Heidelberg: Springer-Verlag, 1994, pp. 141–154.

[27] Blumofe, R., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. "Cilk: An Efficient Multithreaded Runtime System". In: *Journal of Parallel and Distributed Computing*. Vol. 37. Aug. 1996, pp. 55–69. DOI: 10.1006/jpdc.1996.0107.

[28] Bruening, D. L. and Amarasinghe, S. “Efficient, Transparent, and Comprehensive Runtime Code Manipulation”. AAI0807735. PhD thesis. USA: Massachusetts Institute of Technology, 2004.

[29] Bruening, D., Garnett, T., and Amarasinghe, S. “An Infrastructure for Adaptive Dynamic Optimization”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO ’03. San Francisco, California, USA: IEEE Computer Society, 2003, pp. 265–275. ISBN: 076951913X. DOI: 10.1109/CGO.2003.1191551.

[30] Brumar, I., Casas, M., Moretó, M., Valero, M., and Sohi, G. S. “ATM: Approximate Task Memoization in the Runtime System”. In: *Proceedings of the 31st International Parallel and Distributed Processing Symposium*. IPDPS ’17. IEEE, 2017, pp. 1140–1150. DOI: 10.1109/IPDPS.2017.49.

[31] Caheny, P., Alvarez, L., Derradji, S., Valero, M., Moretó, M., and Casas, M. “Reducing Cache Coherence Traffic with a NUMA-Aware Runtime Approach”. In: *Transactions on Parallel and Distributed Systems (TPDS)* 29.5 (2018), pp. 1174–1187. ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2787123.

[32] Caheny, P., Alvarez, L., Valero, M., Moretó, M., and Casas, M. “Runtime-assisted Cache Coherence Deactivation in Task Parallel Programs”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC ’18. IEEE Press, 2018, 35:1–35:12.

[33] Caheny, P., Casas, M., Moretó, M., Gloaguen, H., Saintes, M., Ayguadé, E., Labarta, J., and Valero, M. “Reducing cache coherence traffic with hierarchical directory cache and NUMA-aware runtime scheduling”. In: *Proceedings of the 25th International Conference on Parallel Architecture and Compilation Techniques*. PACT ’16. ACM, 2016, pp. 275–286. DOI: 10.1145/2967938.2967962.

[34] Caminal, H., Caballero, D., Cebrian, J. M., Ferrer, R., Casas, M., Moretó, M., Martorell, X., and Valero, M. “Performance and energy effects on task-based parallelized applications - User-directed versus manual vectorization”. In: *The Journal of Supercomputing* 74.6 (2018), pp. 2627–2637. DOI: 10.1007/s11227-018-2294-9.

[35] Casas, M., Moretó, M., Alvarez, L., Castillo, E., Chasapis, D., Hayes, T., Jaulmes, L., Palomar, O., Unsal, O. S., Cristal, A., Ayguadé, E., Labarta, J., and Valero, M. “Runtime-Aware Architectures”. In: *Proceedings of the 21st International Conference on Parallel and Distributed Computing*. Euro-Par ’15. Springer, 2015, pp. 16–27.

[36] Castillo, E., Alvarez, L., Moretó, M., Casas, M., Vallejo, E., Bosque, J. L., Beivide, R., and Valero, M. "Architectural Support for Task Dependence Management with Flexible Software Scheduling". In: *Proceedings of the 24th International Symposium on High Performance Computer Architecture*. HPCA '18. IEEE, 2018, pp. 283–295. DOI: 10.1109/HPCA.2018.00033.

[37] Castillo, E., Moreto, M., Casas, M., Alvarez, L., Vallejo, E., Chronaki, K., Badia, R., Bosque, J. L., Beivide, R., Ayguadé, E., Labarta, J., and Valero, M. "CATA: Criticality Aware Task Acceleration for Multicore Processors". In: *Proceedings of the 30th International Parallel and Distributed Processing Symposium*. IPDPS '16. IEEE, 2016, pp. 413–422. DOI: 10.1109/IPDPS.2016.49.

[38] Chamberlain, B., Callahan, D., and Zima, H. "Parallel Programmability and the Chapel Language". In: *The International Journal of High Performance Computing Applications* 21.3 (Aug. 2007), pp. 291–312. ISSN: 1094-3420, 1741-2846. DOI: 10.1177/1094342007078442.

[39] Chasapis, D., Casas, M., Moretó, M., Schulz, M., Ayguadé, E., Labarta, J., and Valero, M. "Runtime-Guided Mitigation of Manufacturing Variability in Power-Constrained Multi-Socket NUMA Nodes". In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16. Istanbul, Turkey: ACM, 2016. ISBN: 9781450343619. DOI: 10.1145/2925426.2926279.

[40] Chasapis, D., Casas, M., Moretó, M., Vidal, R., Ayguadé, E., Labarta, J., and Valero, M. "PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite". In: *Transactions on Architecture and Code Optimization* 12.4 (Dec. 2015), 41:1–41:22.

[41] Chasapis, D., Moretó, M., Schulz, M., Rountree, B., Valero, M., and Casas, M. "Power Efficient Job Scheduling by Predicting the Impact of Processor Manufacturing Variability". In: *Proceedings of the International Conference on Supercomputing*. ICS '19. Phoenix, Arizona: ACM, 2019, pp. 296–307. ISBN: 9781450360791. DOI: 10.1145/3330345.3330372.

[42] Chen, W., Liu, P., and Stelzer, K. *Implementation of a pseudo-LRU algorithm in a partitioned cache*. US Patent 7,069,390. June 2006.

[43] Chronaki, K., Rico, A., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. "Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures". In:

Proceedings of the 29th International Conference on Supercomputing. ICS '15. ACM, 2015, pp. 329–338. ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751235.

[44] Chronaki, K., Rico, A., Casas, M., Moretó, M., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. "Task Scheduling Techniques for Asymmetric Multi-Core Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 28.7 (July 2017), pp. 2074–2087. ISSN: 1045-9219. DOI: 10.1109/TPDS.2016.2633347.

[45] Ciesko, J., Mateo, S., Teruel, X., Beltran, V., Martorell, X., and Labarta, J. "Boosting irregular array Reductions through In-lined Block-ordering on fast processors". In: *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2015, pp. 1–6. DOI: 10.1109/HPEC.2015.7322443.

[46] Ciesko, J., Mateo, S., Teruel, X., Martorell, X., Ayguadé, E., and Labarta, J. "Supporting Adaptive Privatization Techniques for Irregular Array Reductions in Task-Parallel Programming Models". In: *OpenMP: Memory, Devices, and Tasks: 12th International Workshop on OpenMP, IWOMP 2016, Nara, Japan, October 5-7, 2016, Proceedings*. 2016, pp. 336–349. DOI: 10.1007/978-3-319-45550-1_24.

[47] Dally, B. "The future of GPU computing". In: *Proceedings of the 22nd Annual Supercomputing Conference*. SC'09. Nov. 2009.

[48] Dally, W. J., Labonte, F., Das, A., Hanrahan, P., Ahn, J.-H., Gummaraju, J., Erez, M., Jayasena, N., Buck, I., Knight, T. J., and Kapasi, U. J. "Merrimac: Supercomputing with Streams". In: *Supercomputing, 2003 ACM/IEEE Conference*. Nov. 2003, pp. 35–35. DOI: 10.1145/1048935.1050187.

[49] Dally, W. J. and Towles, B. P. *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN: 9780080497808.

[50] Davis, T. and Hu, Y. "The University of Florida Sparse Matrix Collection". In: *ACM Transactions on Mathematical Software* 38.1 (2011), pp. 1–25. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

[51] De Gonzalo, S. G., Huang, S., Gómez-Luna, J., Hammond, S., Mutlu, O., and Hwu, W.-m. "Automatic Generation of Warp-level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs". In: *Proceedings of the 2019 International Symposium on Code Generation and Optimization*. CGO '19. Washington, DC, USA, 2019, pp. 73–84. ISBN: 978-1-7281-1436-1.

[52] Dennard, R. H., Gaensslen, F. H., Yu, H.-n., Rideout, V. L., Bassous, E., Andre, and Leblanc, R. "Design of Ion-implanted MOSFETs with Very Small Physical Dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[53] Dimić, V. *Array Scan Benchmark*. https://github.com/vdimic/scan-benchmark. 2020.

[54] Dimić, V. *Reduction Benchmarks*. https://github.com/vdimic/reduction-benchmarks. 2020.

[55] Dimić, V. and Dziegielewska, O. "Metaheuristics Based Approach for Parallelizing Applications in On-Chip Multiprocessor". In: *ACACES'14: Advanced Computer Architecture and Compilation for Embedded Systems 2014 Poster Abstracts*. Poster abstract. 2014.

[56] Dimić, V., Moretó, M., Casas, M., Ciesko, J., and Valero, M. "RICH: Implementing Reductions in the Cache Hierarchy". In: *2020 International Conference on Supercomputing*. ICS '20. New York, NY, USA: ACM, 2020, 13 pages. DOI: 10.1145/3392717.3392736.

[57] Dimić, V., Moretó, M., Casas, M., and Valero, M. *PrioRAT: Criticality-Driven Prioritization Inside the On-Chip Memory Hierarchy*. (under submission).

[58] Dimić, V., Moretó, M., Casas, M., and Valero, M. "Runtime-Assisted Shared Cache Insertion Policies Based on Re-reference Intervals". In: *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*. Ed. by F. F. Rivera, T. F. Pena, and J. C. Cabaleiro. Vol. 10417. Lecture Notes in Computer Science. Springer, 2017, pp. 247–259. DOI: 10.1007/978-3-319-64203-1_18.

[59] Dimić, V., Moretó, M., Casas, M., and Valero, M. "Runtime-Assisted Shared Cache Insertion Policies Based on Re-reference Intervals". In: *RoMoL Final Workshop*. Poster without proceedings. Mar. 2018.

[60] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. "OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures." In: *Parallel Processing Letters* 21.2 (2011), pp. 173–193. ISSN: 0129-6264. DOI: 10.1142/S0129626411000151.

[61] Egielski, I. J., Huang, J., and Zhang, E. Z. "Massive Atomics for Massive Parallelism on GPUs". In: *Proceedings of the 2014 International Symposium on Memory Management*. ISMM '14. Edinburgh, United Kingdom, 2014, pp. 93–103. ISBN: 978-1-4503-2921-7. DOI: 10.1145/2602988.2602993.

[62] Etsion, Y., Cabarcas, F., Rico, A., Ramirez, A., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. "Task Superscalar: An Out-of-Order Task Pipeline". In: *Proceedings of the 43rd Annual International Symposium on Microarchitecture*. MICRO 43. 2010, pp. 89–100. DOI: 10.1109/MICRO.2010.13.

[63] Fang, Z., Zhang, L., Carter, J. B., McKee, S. A., Ibrahim, A., Parker, M. A., and Jiang, X. "Active memory controller". In: *The Journal of Supercomputing* 62.1 (Oct. 2012), pp. 510–549. ISSN: 1573-0484. DOI: 10.1007/s11227-011-0735-9.

[64] Forum, M. P. I. *MPI: A Message-Passing Interface Standard Version 3.1*. Standard. June 2015.

[65] Gao, F., Tziantzioulis, G., and Wentzlaff, D. "ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs". In: *Proceedings of the 52Nd Annual International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: ACM, 2019, pp. 100–113. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358260.

[66] Garcia, V., Rico, A., Villavieja, C., Carpenter, P., Navarro, N., and Ramirez, A. "Adaptive Runtime-Assisted Block Prefetching on Chip-Multiprocessors". In: *International Journal of Parallel Programming* (2016), pp. 1–21. ISSN: 1573-7640. DOI: 10.1007/s10766-016-0431-8.

[67] Garzarán, M. J., Prvulovic, M., Zhang, Y., Torrellas, J., Jula, A., Yu, H., and Rauchwerger, L. "Architectural Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors". In: *PACT '01*. 2001, pp. 243–254. DOI: 10.1109/PACT.2001.953304.

[68] Gonzalez-Mesa, M. A., Quislant, R., Gutierrez, E., and Plata, O. "Exploring Irregular Reduction Support in Transactional Memory". In: *Algorithms and Architectures for Parallel Processing*. Ed. by J. Kołodziej, B. Di Martino, D. Talia, and K. Xiong. 2013, pp. 257–266. DOI: 10.1007/978-3-319-03859-9_22.

[69] Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., and Snir, M. "The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer". In: *IEEE Transactions on Computers* C-32.2 (Feb. 1983), pp. 175–189. DOI: 10.1109/TC.1983.1676201.

[70] Gutierrez, E., Plata, O., and Zapata, E. L. "Improving Parallel Irregular Reductions Using Partial Array Expansion". In: *Supercomputing, ACM/IEEE 2001 Conference*. Nov. 2001, pp. 56–56. DOI: 10.1145/582034.582072.

[71] Han, H. and Tseng, C.-W. "Improving Compiler and Run-Time Support for Irregular Reductions Using Local Writes". In: *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*. LCPC '98. 1999, pp. 181–196. ISBN: 3-540-66426-2. DOI: 10.1007/3-540-48319-5_12.

[72] Han, H. and Tseng, C.-W. "A comparison of parallelization techniques for irregular reductions". In: *Proceedings of the 15th International Parallel and Distributed Processing Symposium*. IPDPS '01. 2001, p. 27. DOI: 10.1109/IPDPS.2001.924963.

[73] *Accelerating Dependent Cache Misses with an Enhanced Memory Controller*. ISCA '16. Seoul, Republic of Korea: IEEE Press, 2016, pp. 444–455. ISBN: 9781467389471. DOI: 10.1109/ISCA.2016.46.

[74] Hennessy, J. L. and Patterson, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X.

[75] Herlihy, M. and Moss, J. E. B. "Transactional Memory: Architectural Support for Lock-free Data Structures". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ISCA '93. San Diego, CA, USA, 1993, pp. 289–300. DOI: 10.1145/165123.165164.

[76] Hutton, G. "A tutorial on the universality and expressiveness of fold". In: *Journal of Functional Programming* 9.4 (1999), pp. 355–372.

[77] IBM Corporation. *IBM System/360 Model 85 Functional Characteristics*. 1968.

[78] IBM Corporation. *Power ISA Version 3.0 B*. Mar. 2017.

[79] IBM Corporation. *Power9 Processor User's Manual*. version 2.0. Apr. 2018.

[80] IEEE Standards Association. *Standard for Information Technology–Portable Operating System Interface (POSIX®) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. Standard. 1003.1c-1995. June 1995.

[81] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation. 2016.

[82] Intel Corporation. *Intel® Cilk™ Plus Language Extension Specification*. Sept. 2013.

[83] Intel Corporation. *Intel ® Xeon ® Scalable Processors*. Product Overview. 2020.

[84] International Organization for Standardization (ISO), Technical Committee SO/IEC JTC 1/SC 22. *Standard for Programming Language C++*. International Standard ISO/IEC 14882:2011. 2011.

[85] Ipek, E., Mutlu, O., Martínez, J. F., and Caruana, R. "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach". en. In: *Proceedings of the 35th International Symposium on Computer Architecture*. ISCA '08. Beijing, China: IEEE, June 2008, pp. 39–50. ISBN: 978-0-7695-3174-8. DOI: 10.1109/ISCA.2008.21.

[86] ITRS Working Group. *The International Technology Roadmap For Semiconductors: Interconnect*. Tech. rep. 2009.

[87] Jain, A. and Lin, C. "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement". In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA '16. Seoul, Republic of Korea: IEEE Press, 2016, pp. 78–89. ISBN: 9781467389471. DOI: 10.1109/ISCA.2016.17.

[88] Jaleel, A., Theobald, K. B., Steely Jr., S. C., and Emer, J. "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)". In: *SIGARCH Computer Architecture News* 38.3 (June 2010), pp. 60–71.

[89] Jaulmes, L. "Exploiting Task-Based Programming Models for Resilience". PhD thesis. Universitat Politècnica de Catalunya, 2019.

[90] Jaulmes, L., Casas, M., Moretó, M., Ayguadé, E., Labarta, J., and Valero, M. "Exploiting Asynchrony from Exact Forward Recovery for DUE in Iterative Solvers". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: ACM, 2015. ISBN: 9781450337236. DOI: 10.1145/2807591.2807599.

[91] Jaulmes, L., Moretó, M., Ayguadé, E., Labarta, J., Valero, M., and Casas, M. "Asynchronous and Exact Forward Recovery for Detected Errors in Iterative Solvers". In: *IEEE Transactions on Parallel and Distributed Systems* 29.9 (2018), pp. 1961–1974. DOI: 10.1109/TPDS.2018.2817524.

[92] Jaulmes, L., Moreto, M., Valero, M., and Casas, M. "A Vulnerability Factor for ECC-protected Memory". In: *Proceedings of the 25th International Symposium on On-Line Testing and Robust System Design*. IOLTS '19. IEEE, July 2019, pp. 176–181. DOI: 10.1109/IOLTS.2019.8854397.

[93] JEDEC. *DDR4 SDRAM Standard*. Specification. JESD79-4C. Jan. 2020.

[94] JEDEC. *High Bandwidth Memory (HBM) DRAM*. Specification. JESD235C. Jan. 2020.

[95] Jeloka, S., Akesh, N., Sylvester, D., and Blaauw, D. "A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory". In: *IEEE Journal of Solid-State Circuits* 51 (Apr. 2016), pp. 1–1. DOI: 10.1109/JSSC.2016.2515510.

[96] Jiménez, D. A. "Insertion and Promotion for Tree-based PseudoLRU Last-level Caches". In: *Proceedings of the 46th Annual International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 284–296.

[97] Kale, L. V. and Krishnan, S. "CHARM++: A Portable Concurrent Object Oriented System Based on C++". In: *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '93. Washington, D.C., USA: ACM, 1993, pp. 91–108. ISBN: 0897915879. DOI: 10.1145/165854.165874.

[98] Kang, M., Kim, E., Keel, M.-S., and Shanbhag, N. "Energy-efficient and high throughput sparse distributed memory architecture". In: ISCAS '15 2015 (July 2015), pp. 2505–2508. DOI: 10.1109/ISCAS.2015.7169194.

[99] Karlin, I., Keasler, J., and Neely, R. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. Livermore, CA, Aug. 2013, pp. 1–9.

[100] Kautz, W. H. "Cellular Logic-in-Memory Arrays". In: *IEEE Transactions on Computers* C-18.8 (Aug. 1969), pp. 719–727. ISSN: 0018-9340. DOI: 10.1109/T-C.1969.222754.

[101] Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., and Glasco, D. "GPUs and the Future of Parallel Computing". In: *IEEE Micro* 31.5 (Sept. 2011), pp. 7–17. ISSN: 0272-1732. DOI: 10.1109/MM.2011.89.

[102] Kessler, R. E. and Schwarzmeier, J. L. "Cray T3D: a new dimension for Cray Research". In: *Digest of Papers. Compcon Spring*. IEEE, Feb. 1993, pp. 176–182. DOI: 10.1109/CMPCON.1993.289660.

[103] Khan, S., Tian, Y., and Jimenez, D. "Sampling Dead Block Prediction for Last-Level Caches". In: *Proceedings of the 43rd Annual International Symposium on Microarchitecture*. MICRO-43. IEEE, Dec. 2010, pp. 175–186. DOI: 10.1109/MICRO.2010.24.

[104] Kharbutli, M. and Solihin, D. "Counter-Based Cache Replacement and Bypassing Algorithms". In: *IEEE Transactions on Computers* 57.4 (Apr. 2008), pp. 433–447. DOI: 10.1109/TC.2007.70816.

[105] Kim, H., Sim, J., Choi, Y., and Kim, L. "NAND-Net: Minimizing Computational Complexity of In-Memory Processing for Binary Neural Networks". In: *Proceedings of the 25th International Symposium on High Performance Computer Architecture*. HPCA '19. Feb. 2019, pp. 661–673. DOI: 10.1109/HPCA.2019.00017.

[106] Kim, Y., Han, D., Mutlu, O., and Harchol-Balter, M. "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers". In: *Proceedings of The 16th International Symposium on High-Performance Computer Architecture*. HPCA '10. Jan. 2010, pp. 1–12. DOI: 10.1109/HPCA.2010.5416658.

[107] Kim, Y., Seshadri, V., Lee, D., Liu, J., and Mutlu, O. "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM". In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA '12. Portland, Oregon: IEEE Computer Society, 2012, pp. 368–379. ISBN: 9781450316422.

[108] Kim, Y.-B. and Chen, T. W. "Assessing merged DRAM/Logic technology". In: *Integration* 27.2 (1999), pp. 179–194. ISSN: 0167-9260. DOI: 10.1016/S0167-9260(99)00006-1.

[109] Komatitsch, D. and Tromp, J. "Introduction to the spectral-element method for 3-D seismic wave propagation". In: *Geophysical Journal International* 139.3 (1999), pp. 806–822.

[110] Kozyrakis, C. E., Perissakis, S., Patterson, D., Anderson, T., Asanovic, K., Cardwell, N., Fromm, R., Golbus, J., Gribstad, B., Keeton, K., Thomas, R., Treuhaft, N., and Yelick, K. "Scalable Processors in the Billion-Transistor Era: IRAM". In: *Computer* 30.9 (Sept. 1997), pp. 75–78. ISSN: 0018-9162. DOI: 10.1109/2.612252.

[111] Kumar, S., Hughes, C. J., and Nguyen, A. "Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors". In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA '07. ACM, 2007, pp. 162–173. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250683.

[112] Laudon, J. and Lenoski, D. "The SGI Origin: A ccNUMA Highly Scalable Server". In: *SIGARCH Computer Architecture News* 25.2 (May 1997), pp. 241–251. ISSN: 0163-5964. DOI: 10.1145/384286.264206.

[113] Le Chevalier, F., Montecot, M., Doisy, Y., Letestu, F., and Chevalier, P. "STAP developments in Thales". In: *2009 European Radar Conference*. EuRAD '09. IEEE, Jan. 2009, pp. 53–56. ISBN: 978-1-4244-4747-3.

[114] Lee, D. U., Kim, K. W., Kim, K. W., Kim, H., Kim, J. Y., Park, Y. J., Kim, J. H., Kim, D. S., Park, H. B., Shin, J. W., Cho, J. H., Kwon, K. H., Kim, M. J., Lee, J., Park, K. W., Chung, B., and Hong, S. "25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV". In: *Digest of Technical Papers of International Solid-State Circuits Conference*. ISSCC '14. 2014, pp. 432–433. DOI: 10.1109/ISSCC.2014.6757501.

[115] Li, S., Ahn, J. H., Strong, R., Brockman, J., Tullsen, D., and Jouppi, N. "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures". In: *Proceedings of the 42nd Annual International Symposium on Microarchitecture*. MICRO-42. New York, New York, 2009, pp. 469–480. ISBN: 978-1-60558-798-1. DOI: 10.1145/1669112.1669172.

[116] Li, S., Niu, D., Malladi, K. T., Zheng, H., Brennan, B., and Xie, Y. "DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator". In: *Proceedings of the 50th Annual International Symposium on Microarchitecture*. MICRO-50. Cambridge, Massachusetts: ACM, 2017, pp. 288–301. ISBN: 9781450349529. DOI: 10.1145/3123939.3123977.

[117] Liu, H., Chen, Y., Liao, X., Jin, H., He, B., Zheng, L., and Guo, R. "Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures". In: *Proceedings of the International Conference on Supercomputing*. ICS '17. 2017, 26:1–26:10. DOI: 10.1145/3079079.3079089.

[118] Manivannan, M., Papaefstathiou, V., Pericas, M., and Stenstrom, P. "RADAR: Runtime-assisted dead region management for last-level caches". In: *Proceedings of the 22nd International Symposium on High Performance Computer Architecture*. HPCA '16. Mar. 2016, pp. 644–656. DOI: 10.1109/HPCA.2016.7446101.

[119] Manivannan, M., Pericás, M., Papaefstathiou, V., and Stenström, P. "Global Dead-Block Management for Task-Parallel Programs". In: *ACM Transactions on Architecture and Code Optimization* 15.3 (Sept. 2018), pp. 1–25. ISSN: 15443566. DOI: 10.1145/3234337.

[120] McMahon, F. H. *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range*. Tech. rep. UCRL-53745. Lawrence Livermore National Laboratory, Dec. 1986.

[121] Micron. *MT40A 8Gb: x4, x8, x16 DDR4 SDRAM Features*. Specification. CCMTD-1725822587-9875. Apr. 2020.

[122] *MIPS IV Instruction Set, Revision 3.2*. MIPS Technologies, Inc. 1995.

[123] Moore, G. E. "Cramming More Components onto Integrated Circuits". In: *Electronics* 38.8 (Apr. 1965), pp. 114–117.

[124] Moore, G. E. "Progress in digital integrated electronics". In: *Electron Devices Meeting, 1975 International*. Vol. 21. 1975, pp. 11–13.

[125] Mukkara, A., Beckmann, N., and Sanchez, D. "PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates". In: *Proceedings of the 52nd Annual International Symposium on Microarchitecture*. MICRO-52. Columbus, OH, USA, 2019, pp. 1009–1022. ISBN: 9781450369381. DOI: 10.1145/3352460.3358254.

[126] Muralimanohar, N., Balasubramonian, R., and Jouppi, N. "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0". In: *Proceedings of the 40th Annual International Symposium on Microarchitecture*. MICRO-40. IEEE Computer Society, 2007, pp. 3–14. ISBN: 0-7695-3047-8. DOI: 10.1109/MICRO.2007.30.

[127] Muralimanohar, N., Balasubramonian, R., and Jouppi, N. *CACTI 6.0: A Tool to Understand Large Caches*. Tech. rep. 2009.

[128] Mutlu, O. and Moscibroda, T. "Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers". In: *IEEE Micro* 29.1 (Jan. 2009), pp. 22–32. ISSN: 0272-1732. DOI: 10.1109/MM.2009.12.

[129] Mutlu, O. and Moscibroda, T. "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors". In: *Proceedings of the 40th Annual International Symposium on Microarchitecture*. MICRO 40. USA: IEEE Computer Society, 2007, pp. 146–160. ISBN: 0769530478. DOI: 10.1109/MICRO.2007.40.

[130] Nai, L., Hadidi, R., Sim, J., Kim, H., Kumar, P., and Kim, H. "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks". In: *Proceedings of the 23rd International Symposium on High Performance Computer Architecture*. HPCA '17. Feb. 2017, pp. 457–468. DOI: 10.1109/HPCA.2017.54.

[131] Nair, R., Antao, S. F., Bertolli, C., Bose, P., Brunheroto, J. R., Chen, T., Cher, C., Costa, C. H. A., Doi, J., Evangelinos, C., Fleischer, B. M., Fox, T. W., Gallo, D. S., Grinberg, L., Gunnels, J. A., Jacob, A. C., Jacob, P., Jacobson, H. M., Karkhanis, T., Kim, C., Moreno, J. H., O?Brien, J. K., Ohmacht, M., Park, Y., Prener, D. A., Rosenburg, B. S., Ryu, K. D., Sallenave, O., Serrano, M. J., Siegl, P. D. M., Sugavanam, K., and Sura, Z. "Active Memory Cube: A Processing-In-Memory Architecture for Exascale Systems". In: *IBM Journal of Research and Development*. Mar. 2015.

[132] Nesbit, K. J., Aggarwal, N., Laudon, J., and Smith, J. E. "Fair Queuing Memory Systems". In: *Proceedings of the 39th Annual International Symposium on Microarchitecture*. MICRO-39. 2006, pp. 208–222. DOI: 10.1109/MICRO.2006.24.

[133] Nguyen, T. M. and Wentzlaff, D. "MORC: A Manycore-oriented Compressed Cache". In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 76–88. ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830828.

[134] Oaks, S. and Wong, H. *Java Threads*. 3rd. 2009. ISBN: 978-0-596-00782-9.

[135] OpenMP Architecture Review Board. *OpenMP Application Program Interface, v3.0*. 2008.

[136] OpenMP Architecture Review Board. *OpenMP Application Program Interface, v4.0*. 2013.

[137] OpenMP Architecture Review Board. *OpenMP Technical Report 4 Version 5.0 Preview 1*. Nov. 2016.

[138] Papaefstathiou, V., Katevenis, M. G., Nikolopoulos, D. S., and Pnevmatikatos, D. "Prefetching and Cache Management Using Task Lifetimes". In: *Proceedings of the 27th International Conference on International Conference on Supercomputing*. ICS '13. Eugene, Oregon, USA: ACM, 2013, pp. 325–334. ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465443.

[139] Pawlowski, J. T. "Hybrid Memory Cube (HMC)". In: *HOT CHIPS 23* (Aug. 2011).

[140] Peiron, M., Valero, M., Ayguadé, E., and Lang, T. "Vector Multiprocessors with Arbitrated Memory Access". In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ISCA '95. S. Margherita Ligure, Italy: ACM, 1995, pp. 243–252. ISBN: 0897916980. DOI: 10.1145/223982.224435.

[141] Plattner, H. and Zeier, A. *In-Memory Data Management: Technology and Applications*. Springer Berlin Heidelberg, 2012. ISBN: 9783642295744.

[142] Pottenger, W. M. "The Role of Associativity and Commutativity in the Detection and Transformation of Loop-level Parallelism". In: *Proceedings of the 12th International Conference on Supercomputing*. ICS '98. Melbourne, Australia, 1998, pp. 188–195. ISBN: 0-89791-998-X. DOI: 10.1145/277830.277870.

[143] Powell, M., Yang, S.-H., Falsafi, B., Roy, K., and Vijaykumar, T. N. "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories". In: *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*. ISLPED '00. Rapallo, Italy, 2000, pp. 90–95. ISBN: 1-58113-190-9. DOI: 10.1145/344166.344526.

[144] Qureshi, M., Jaleel, A., Patt, Y., Steely, S., and Emer, J. "Set-Dueling-Controlled Adaptive Insertion for High-Performance Caching". In: *IEEE Micro* 28.1 (Jan. 2008), pp. 91–98.

[145] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. "Adaptive Insertion Policies for High Performance Caching". In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA '07. 2007, pp. 381–391.

[146] Qureshi, M. K., Lynch, D. N., Mutlu, O., and Patt, Y. N. "A Case for MLP-Aware Cache Replacement". In: *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. ISCA '06. 2006, pp. 167–178.

[147] Rico, A., Cabarcas, F., Villavieja, C., Pavlovic, M., Vega, A., Etsion, Y., Ramirez, A., and Valero, M. "On the Simulation of Large-scale Architectures Using Multiple Application Abstraction Levels". In: *ACM Transactions on Architecture and Code Optimization* 8.4 (Jan. 2012), 36:1–36:20. DOI: 10.1145/2086696.2086715.

[148] Rico, A., Duran, A., Cabarcas, F., Etsion, Y., Ramirez, A., and Valero, M. "Trace-driven simulation of multithreaded applications". In: *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. ISPASS '11. Apr. 2011, pp. 87–96. DOI: 10.1109/ISPASS.2011.5762718.

[149] Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., and Owens, J. D. "Memory Access Scheduling". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ISCA '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 128–138. ISBN: 1581132328. DOI: 10.1145/339647.339668.

[150] Rotenberg, E., Bennett, S., and Smith, J. E. "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching". In: *Proceedings of the 29th Annual International Symposium on Microarchitecture*. MICRO-29. Paris, France: IEEE Computer Society, 1996, pp. 24–35. ISBN: 0818676418.

[151] Rupp, K. *Microprocessor Trend Data*. https://github.com/karlrupp/microprocessor-trend-data. 2018.

[152] Sánchez Barrera, I., Casas, M., Moretó, M., Ayguadé, E., Labarta, J., and Valero, M. "Graph Partitioning Applied to DAG Scheduling to Reduce NUMA Effects". In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '18. ACM, 2018, pp. 419–420. ISBN: 978-1-4503-4982-6. DOI: 10.1145/3178487.3178535.

[153] Sánchez Barrera, I., Moretó, M., Ayguadé, E., Labarta, J., Valero, M., and Casas, M. "Reducing Data Movement on Large Shared Memory Systems by Exploiting Computation Dependencies". In: *Proceedings of the 2018 International Conference on Supercomputing*. ICS '18. ACM, 2018, pp. 207–217. ISBN: 978-1-4503-5783-8. DOI: 10.1145/3205289.3205310.

[154] Sartor, J., Venkiteswaran, S., McKinley, K., and Wang, Z. "Cooperative caching with keep-me and evict-me". In: *Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures*. INTERACT-9. Feb. 2005, pp. 46–57.

[155] Scott, S. L. "Synchronization and Communication in the T3E Multiprocessor". In: *SIGPLAN Notices* 31.9 (Sept. 1996), pp. 26–36. ISSN: 0362-1340. DOI: 10.1145/248209.237144.

[156] Seshadri, V., Kim, Y., Fallin, C., Lee, D., Ausavarungnirun, R., Pekhimenko, G., Luo, Y., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C. "RowClone: Fast and Energy-Efficient in-DRAM Bulk Data Copy and Initialization". In: *Proceedings of the 46th Annual International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 185–197. ISBN: 9781450326384. DOI: 10.1145/2540708.2540725.

[157] Seshadri, V., Lee, D., Mullins, T., Hassan, H., Boroumand, A., Kim, J., Kozuch, M. A., Mutlu, O., Gibbons, P. B., and Mowry, T. C. "Ambit: In-memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology". In: *Proceedings of the 50th Annual International Symposium on Microarchitecture*. MICRO-50 '17. Cambridge, Massachusetts, 2017, pp. 273–287. DOI: 10.1145/3123939.3124544.

[158] Sewell, P., Sarkar, S., Owens, S., Nardelli, F. Z., and Myreen, M. O. "X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors". In: *Communications of the ACM* 53.7 (July 2010), pp. 89–97. ISSN: 0001-0782. DOI: 10.1145/1785414.1785443.

[159] Shewchuk, J. R. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. rep. 1994.

[160] Singh, G., Chelini, L., Corda, S., Awan, A. J., Stuijk, S., Jordans, R., Corporaal, H., and Boonstra, A. "Near-Memory Computing: Past, Present, and Future". In: *Microprocessors and Microsystems* 71 (Aug. 2019), p. 102868. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2019.102868.

[161] Smith, J. E. "A Study of Branch Prediction Strategies". In: *Proceedings of the 8th Annual Symposium on Computer Architecture*. ISCA '81. Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981, pp. 135–148.

[162] Stone, H. S. "A Logic-in-Memory Computer". In: *IEEE Transactions on Computers* 19.1 (Jan. 1970), pp. 73–78. ISSN: 0018-9340. DOI: 10.1109/TC.1970.5008902.

[163] Subramanian, L., Lee, D., Seshadri, V., Rastogi, H., and Mutlu, O. "The Blacklisting Memory Scheduler: Achieving high performance and fairness at low cost". In: *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. Oct. 2014, pp. 8–15. DOI: 10.1109/ICCD.2014.6974655.

[164] Sun Microsystems. "UltraSPARC T2 supplement to the UltraSPARC architecture 2007". In: (2007). Draft D1.4.3.

[165] Tan, X., Bosch, J., Jiménez-González, D., Álvarez-Martínez, C., Ayguadé, E., and Valero, M. "Performance analysis of a hardware accelerator of dependence management for task-based dataflow programming models". In: *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. ISPASS '16. 2016, pp. 225–234. DOI: 10.1109/ISPASS.2016.7482097.

[166] Tan, X., Bosch, J., Vidal, M., Álvarez, C., Jiménez-González, D., Ayguadé, E., and Valero, M. "General Purpose Task-Dependence Management Hardware for Task-Based Dataflow Programming Models". In: *Proceedings of the 31st International Parallel and Distributed Processing Symposium*. IPDPS '17. 2017, pp. 244–253. DOI: 10.1109/IPDPS.2017.48.

[167] Teruel, X. *OmpSs Quick Overview, A Practical Approach*. 2013.

[168] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. Editors Andrew Waterman and Krste Asanović. RISC-V Foundation. May 2017.

[169] Valero, M., Lang, T., Llaberıéa, J. M., Peiron, M., Ayguadé, E., and Navarra, J. J. "Increasing the Number of Strides for Conflict-Free Vector Access". In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ISCA '92. Queensland, Australia: ACM, 1992, pp. 372–381. ISBN: 0897915097. DOI: 10.1145/139669.140400.

[170] Valero, M., Moretó, M., Casas, M., Ayguade, E., and Labarta, J. "Runtime-Aware Architectures: A First Approach". In: *Supercomputing frontiers and innovations* 1.1 (2014).

[171] Venkat, A. and Tullsen, D. M. "Harnessing ISA Diversity: Design of a heterogeneous-ISA Chip Multiprocessor". In: *SIGARCH Computer Architecture News* 42.3 (June 2014), pp. 121–132.

[172] Vries, H. de. *AMD's Hammer micro architecture preview*. 2001. URL: http://www.chip-architect.com/news/2001_10_02_Hammer_microarchitecture.html (visited on 05/20/2020).

[173] Wang, Z., McKinley, K. S., Rosenberg, A. L., and Weems, C. C. "Using the Compiler to Improve Cache Replacement Decisions". In: *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*. PACT '02. USA: IEEE Computer Society, 2002, p. 199. ISBN: 0769516203.

[174] Wilson, R. P., French, R. S., Wilson, C. S., Amarasinghe, S. P., Anderson, J. M., Tjiang, S. W. K., Liao, S.-W., Tseng, C.-W., Hall, M. W., Lam, M. S., and Hennessy, J. L. "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers". In: *SIGPLAN Notices* 29.12 (Dec. 1994), pp. 31–37. ISSN: 0362-1340. DOI: 10.1145/193209.193217.

[175] Wong, H.-S. P., Raoux, S., Kim, S., Liang, J., Reifenberg, J. P., Rajendran, B., Asheghi, M., and Goodson, K. E. "Phase Change Memory". In: *Proceedings of the IEEE* 98.12 (2010), pp. 2201–2227. DOI: 10.1109/JPROC.2010.2070050.

[176] Wu, C.-J., Jaleel, A., Hasenplaugh, W., Martonosi, M., Steely Jr., S. C., and Emer, J. "SHiP: Signature-based Hit Predictor for High Performance Caching". In: *Proceedings of the 44th Annual International Symposium on Microarchitecture*. MICRO-44. Porto

Alegre, Brazil, 2011, pp. 430–441. ISBN: 978-1-4503-1053-6. DOI: 10.1145/2155620.2155671.

[177] Wulf, W. A. and McKee, S. A. "Hitting the memory wall: implications of the obvious". In: *SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588.

[178] Xi, S., Jacobson, H., Bose, P., Wei, G.-Y., and Brooks, D. "Quantifying sources of error in McPAT and potential impacts on architectural studies". In: *Proceedings of the 21st International Symposium on High Performance Computer Architecture*. HPCA '15. 2015, pp. 577–589. DOI: 10.1109/HPCA.2015.7056064.

[179] Yu, H. and Rauchwerger, L. "Adaptive reduction parallelization techniques". In: *Proceedings of the 14th international conference on Supercomputing*. 2000, pp. 66–77. DOI: 10.1145/2591635.2667180.

[180] Zhang, G., Chiu, V., and Sanchez, D. "Exploiting semantic commutativity in hardware speculation". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. IEEE, Oct. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783737.

[181] Zhang, G., Horn, W., and Sanchez, D. "Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-coherent Systems". In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii, 2015, pp. 13–25. ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830774.

[182] Zhang, L., Fang, Z., and Carter, J. B. "Highly Efficient Synchronization based on Active Memory Operations". In: *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. IPDPS '04. USA: IEEE, 2004, pp. 58–67. DOI: 10.1109/IPDPS.2004.1302981.

[183] Zhang, X. and Yan, Y. "Comparative Modeling and Evaluation of CC-NUMA and COMA on Hierarchical Ring Architectures". In: *IEEE Transactions on Parallel and Distributed Systems* 6.12 (Dec. 1995), pp. 1316–1331. ISSN: 1045-9219. DOI: 10.1109/71.476171.

List of Figures

List of Tables

Glossary

ALU ..... Arithmetic-Logic Unit
AMC ..... Active Memory Cube
BIP ..... Bimodal Insertion Policy [145]
BRRIP ..... Bimodal RRIP [88]
BSC ..... Barcelona Supercomputing Center
CAM ..... Content-Addressable Memory
CMP ..... Chip Multi-Processor
DDR ..... Double Data Rate
DepRRI ..... Dependency type to RRI mapping
DIMM ..... Dual In-line Memory Module
DIMM ..... Dual Inline Memory Module
DIP ..... Dynamic Insertion Policy [145]
DRAM ..... Dynamic Random-Access Memory
DTIP ..... Data-Type-aware Insertion Policy
EDP ..... Energy-Delay Product
FCFS ..... First-Come, First-Served
FIFO ..... First-In First-Out
FPU ..... Floating-Point Unit
FR-FCFS ..... First-Ready FCFS
FU ..... Functional Unit
GPU ..... Graphics Processing Unit
HBM ..... High Bandwidth Memory
HMC ..... Hybrid Memory Cube
HPC ..... High-Performance Computing
IPV ..... Insertion/Promotion Vector [96]
ISA ..... Instruction Set Architecture
LIP ..... LRU Insertion Policy

LLC ..... Last-Level Cache
LRU ..... Least-Recently Used
MPI ..... Message-Passing Interface
MPKI ..... Misses Per Kilo Instruction
MRU ..... Most-Recently Used
MSHR ..... Miss Status Handling Register
NRU ..... Not-Recently Used
NUMA ..... Non-Uniform Memory Access
NVM ..... Non-Volatile Memory
OS ..... Operating System
PCM ..... Phase-Change Memory
PIM ..... Processing In Memory
QoS ..... Quality of Service
RegDepT ..... Region to Dependency Type mapping
RICH ..... Reductions In Cache Hierarchy
RM ..... Reduction Module
RMEX ..... Reduction Module Execution unit
RMIQ ..... Reduction Module Instruction Queue
RMS ..... Recognition, Mining, and Synthesis
RMSQ ..... Reduction Module Store Queue
RRI ..... Re-Reference Interval
RRIP ..... Re-Reference Interval Prediction [88]
RVT ..... Reduction Variable Table
SDRAM ..... Synchronous Dynamic Random-Access Memory
SHiP ..... Signature-based Hit Predictor [176]
SIMD ..... Single Instruction, Multiple Data
SMT ..... Simultaneous Multi-Threading
SRAM ..... Static Random-Access Memory
SRRIP ..... Static RRIP [88]
TDG ..... Task Dependency Graph
TLB ..... Translation Look-ahead Buffer
TM ..... Transactional Memory
TSO ..... Total Store Ordering
TSV ..... Through-Silicon Vias
TTIP ..... Task-Type-aware Insertion Policy