SOFTWARE/HARDWARE CO-DESIGN AND CO-SPECIALISATION: NOVEL SIMULATION TECHNIQUES AND OPTIMISATIONS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2018

By
Andrey Rodchenko

School of Computer Science

Contents

Word Count: 35295

List of Tables

List of Listings

List of Figures

Abstract

SOFTWARE/HARDWARE CO-DESIGN AND CO-SPECIALISATION:
NOVEL SIMULATION TECHNIQUES AND OPTIMISATIONS
Andrey Rodchenko
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2018

Progress in microprocessors has moved towards increasing the number of cores, heterogeneity, and bitness of computing. Performance, programmability and energy efficiency of next generations of microprocessors will be highly dependent on efficient synchronisation, hardware virtualisation, memory subsystem utilisation and closer synergy between hardware and software. This multidisciplinary work addresses these challenges by advancing the state-of-the-art in the following three major fields of computer science: shared-memory synchronisation, computer architecture simulation, and high-level language computer architecture.

Firstly, this thesis presents a study of the state-of-the-art barrier synchronisation algorithms specialised for the Intel Xeon Phi architecture. The novel proposed hybrid barrier synchronisation algorithm exploits the topology, the memory hierarchy, and other capabilities of the Intel Xeon Phi 5110P coprocessor. The showcased algorithm achieves a 3.28× lower overhead than the Intel OpenMP barrier implementation (ICC 14.0.0), thus outperforming all other known implementations. The study investigates design issues of Intel Xeon Phi with respect to barrier synchronisation. Furthermore, the thesis introduces an extensible parameterised framework for empirical evaluation of barrier synchronisation algorithms on different systems, and it is released as free software.

Secondly, a novel open-source simulation platform named MaxSim is introduced.

MaxSim facilitates hardware/software co-design of managed runtime environments and architectures with tagged pointers support. It has an awareness of the managed runtime environment, supports fast tagged pointers simulation on the x86-64 architecture, allows to model new hardware extensions, to perform microarchitectural profiling and to model complex software changes via a novel address-space morphing technique. MaxSim is available as free software.

Finally, the work explores hardware/software co-design opportunities of managed runtime environments and architectures with tagged pointers support. It is shown how an array length can be stored in a tagged pointer and efficiently retrieved from it with the assistance of hardware extenstions in Java Virtual Machine implementations. The proposed technique resulted in up to 4% and 2% geometric mean dynamic energy reduction and up to 14% and 7% geometric mean L1 data cache loads reduction. The work also researches how tagged pointers can be used for storing type information in Java Virtual Machine implementations. In addition, novel hardware extensions to the address generation and load-store units are proposed to achieve low-overhead type information retrieval and tagged object pointers compression-decompression. The evaluation shows up to 26% and 10% geometric mean heap space savings, up to 50% and 12% geometric mean dynamic random-access memory dynamic energy reduction, and up to 49% and 3% geometric mean execution time reduction.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

Doing a PhD is a significant time and effort investment, so it is a serious decision one has to make. Before starting, it is necessary to have clear goals and motivation, knowledge and interest in the area, support of family and friends. While doing it, it is important to have guidance and assistance of experienced researchers and have a team of great colleagues. I am grateful to all people who helped me both to start and to do it which are very many.

First of all, I would like to thank my supervisor Professor Mikel Lujan for his guidance and advice. He directed me at every step of this thesis from elaborating the ideas to presenting them in the papers and conferences. I did not have to worry about anything rather than doing research: whatever hardware, proprietary software, attendance of the conferences or symposiums were necessary, everything was provided and organised fast and efficiently.

During different stages of my PhD, I was also actively supported by my co-supervisor Antoniu Pop and by APT research fellows Andy Nisbet and Christos Kotselidis. I would like to thank them for productive meetings, writing and presentation support. I would like to thank every member of the APT team for making it a great place to do research. Specifically, I would like to thank Paraskevas Yiapanis, Athanasios Stratikopoulos, Mireya Paredes, Yaman Cakmakci, James Clarkson, Sebastian Werner, Ioanna Alifieraki, Swapnil Gaikwad, Serhat Gesoglu, Merve Simsek, Seckin Simsek, Guillermo Callaghan, Amanieu D'Antras, Cosmin Gorgovan, Nikolaos Foutris, Geoffrey Ndu, Will Toms, and John Mawer for everyday support and fruitful discussions during this PhD.

In order to start this PhD, many people helped me to gain knowledge and interest in the area. I would like to thank my teachers in the Secondary School No. 3, Zhukovsky. Specifically, I would like to thank Sima Mikhailovna Bilich, Svetlana Fyodorovna Remennikova, Elvira Ivanovna Kapustina, and Nina Georgievna Blokhina. I would like to thank my lecturers at Moscow Institution of Physics and Technology,

and my colleagues at Intel and Optimizing Technologies. Specifically, I would like to thank Fatima Gyoeva, Andrey Bokhanko, Sergey Novikov, Alexander Drozdov, Pavel Matveyev, Dmitry Maslennikov, Andrey Dobrov, Valery Perekatov, Arnold Plotkin, and Boris Babayan.

I would like to thank Microsoft Research and the School of Computer Science at the University of Manchester for providing financial support via a PhD Scholarship. I would like to thank Anton Lokhmotov for notifying me about this fully funded PhD opportunity.

I would like to thank my mother Taisiya Rodchenko, my sister Larisa, her husband Mikhail, their children Maria and Matvey, all other members of my big family, and friends for their love and support. I would like to thank my wife Anna Bystranova for her love, support and being with me in the great city of Manchester.

This thesis is dedicated to my father Victor Rodchenko, who was a reasearcher in the area of flight dynamics and control of aircrafts. He is and will always be an inspiration for my work.

Chapter 1

Introduction

The steady growth of computing performance is a significant driving force for human development. Advances in processors technologies follow the path of increasing the number of cores, their heterogeneity and bitness of computing. From the software side, one of the trends is a steady development of high-level language virtual machine technologies. As it will be claimed in this chapter, further performance gains in computing will be significantly dependent on hardware/software co-specialisation and co-design.

In detail, this chapter discusses:

1. **Trends in integrated circuits and processors technologies**.

2. **The evolution of high-level language virtual machine technologies**.

3. **A hardware/software co-design and co-specialisation approach** for hardware/software systems development.

4. **Aims** of this work.

5. **Contributions** to the areas of hardware/software co-design and co-specialisation.

6. **Published papers** stemming from this work.

7. **The structure** of the thesis.

1.1 Trends in Integrated Circuits and Processor Technologies

In 1965, Gordon Moore made an observation and prediction that **the number of transistors on integrated circuits would double roughly every two years**. This trend in computing is known as **Moore's Law** [Moo65]. Indeed, when this work was started in 2013, processors using 22nm technology nodes were shipped to consumers. In 2017, when this work was completed, 10nm devices were expected to appear in the market. However, this exponential growth will end at some point in the future due to physical limitations. The exact time is difficult to predict as it depends on future technological advancements. The ultimate theoretical quantum limit of miniaturisation of electronic devices sets the year of 2036 as the upper boundary for Moore's Law [Pow08]. The inevitable end of Moore's Law will require specialisation of integrated circuits to get further performance improvements.

Prior to the early 2000s, utilisation of the growing transistor budget was mainly focused on the single-core *Central Processing Unit* (CPU) domain. Processor performance improvements were primarily achieved by the technological improvements and development in microarchitecture leveraging frequency growth and higher *Instruction Per Clock* (IPC) rates. **The single-core micro-architecture performance growth** followed the empirical **Pollack's Rule** [Bor07], stating that it **is roughly proportional to the square root of the increase in logic complexity**, which is commensurate with the number of logic transistors. However, in the mid-2000s the microprocessor frequencies reached a plateau in the 3-4 GHz range hitting *Thermal Design Power* (TDP) and microarchitectural constraints. Therefore, the increase of IPC stopped as increasing *Instruction-Level Parallelism* (ILP) became harder [KS04] combined with relatively small progress in reducing memory access latency.

Consequently, **transistor utilisation is now typically exploited by increasing the number of processor cores**. Multiprocessors allow achieving linearly proportional performance to the number of cores that can be used simultaneously for independent non-interfering workloads, in contrast to the square root growth by Pollack's Rule in case of the single powerful monolithic core [Bor07]. This fact explains the appearance of commodity off-the-shelf many-core system such as the 60-Core Intel Xeon Phi 5110P coprocessor. However, moving in a many-core direction introduces numerous challenges. Firstly, future multi-core systems are moving towards heterogeneous and asymmetric architectures [EBSA+11], [Bor07], which are challenging to program.

Secondly, performance scaling of a single application will require extraction of parallelism sufficient to exploit increasing computational resources.

Another trend in computing is **the increase of bitness of CPUs**. Bitness defines the typical size of data the CPU can operate with (a word), however sizes of registers, address buses and other components can be different. For instance, the first commercially available microporcessor Intel 4004 had 4-bit data width and 12-bit address width, and the most recent to date Intel x86-64 microprocessors feature 64-bit data width and 48-bit address width. 64-bit computing became the norm in server and desktop segments since the turn of the century, and it is now the norm in the mobile devices segment since the introduction of the ARMv8 architecture in 2011.

Power dissipation is a major limitation for future generations of many-core systems [Bor07, EBSA+11, HRSS11]. Specialisation of an integrated circuit for a certain application is an effective way to improve energy efficiency. If the types of applications to be executed on a designed many-core system are known in advance, some cores can be specialised for the dominant types of applications. Thus, instead of having identical general-purpose cores, many-core systems can contain different specialised cores. The recent trend is **the increase of the degree of heterogeneity of multi-core systems**. Three major categories of specialised integrated circuits are described below:

- *Application Specific Integrated Circuits* (ASIC) are special-purpose non-reconfigurable circuits designed to run particular applications. It is shown that H.264 encoding is 500× less energy efficient on the 4-core *Chip Multi-Processor* (CMP) than on the ASIC [HQW+10].

- *Field Programmable Gate Arrays* (FPGA) are special-purpose reconfigurable circuits. FPGAs can be reconfigured fully or partially at runtime in contrast to design-time specialised ASICs. FPGAs provide higher energy efficiency than general purpose cores but lower than ASICs for certain workloads [Und04].

- *Graphics Processing Units* (GPU) are another evolving branch of specialised integrated circuits. GPUs were originally oriented towards massively parallel graphics workloads and were later adapted to other *Single-Instruction Multiple-Thread* (SIMT) tasks. In this domain, they showed performance superiority over general purpose cores on a number of applications, for instance, a sparse matrix conjugate gradient solver and a multigrid solver [BFGS03].

All these categories of specialised circuits can be placed on a single multi-core chip with general-purpose CPUs. Among the recent solutions featuring an FPGA and

a CPU on a single chip are Xilinx Zynq-7000 [MCS14b] and Altera Arriva V FPGA [MCS14a]. Such chips feature lower delays and higher bandwidth of data movement between a CPU and an FPGA and reduce the cost of offloading computation from the former to the latter. Thus, integrated CPU and FPGA chips allow broader classes of tasks to benefit from offloading and acceleration on an FPGA in comparison with separate chips. The heterogeneous multi-core chips can also contain different types of general-purpose CPU cores with different characteristics. A notable example is the ARM big.LITTLE architecture, featuring "big" high-performance and "LITTLE" low-power cores implementing the same *Instruction Set Architecture* (ISA).

1.2 The Evolution of High-Level Language Virtual Machine Technologies

As *Hardware* (HW) becomes more diverse, *Software* (SW) portability and adaptation for performance become increasingly important. To tackle this problem, applications may target not an underlying HW directly but an abstract *Virtual Machine* (VM). The concept of the *High-Level Language* (HLL) VM lies in reflecting HLL features in the *Virtual-Instruction Set Architecture* (V-ISA) [SN05]. So the HLL VM programs are initially converted to V-ISA. The process of conversion from HLL code to ISA or V-ISA code is known as compilation, and it is performed by a SW program called a compiler. On the execution stage a *Managed Runtime Environment* (MRE) can interpret code in the V-ISA format instruction by instruction modelling a VM behavior. Alternatively, an MRE can compile multiple V-ISA instructions to ISA instructions and execute them on a host. Compilation during runtime is called *Just-In-Time* (JIT) compilation.

One of the earliest examples of HLL VMs was the BCPL typeless language and the OCODE abstract stack machine specifically designed for this language [Ric69]. At the first stage, programs written in BCPL are compiled to the OCODE abstract machine format, and at the second stage, they can be either compiled to or interpreted on a target machine. This design choice is the key to portability of the BCPL compiler. Though the BCPL language is not in wide use nowadays, the language itself influenced the widely used C language and its implementation approach influenced the implementation of the Pascal language. Similiarly to BCPL, at the first stage, programs in the typed imperative Pascal language are translated to the P-code stack-based virtual machine [NNJA74], while at the second stage, they are either interpreted on or translated

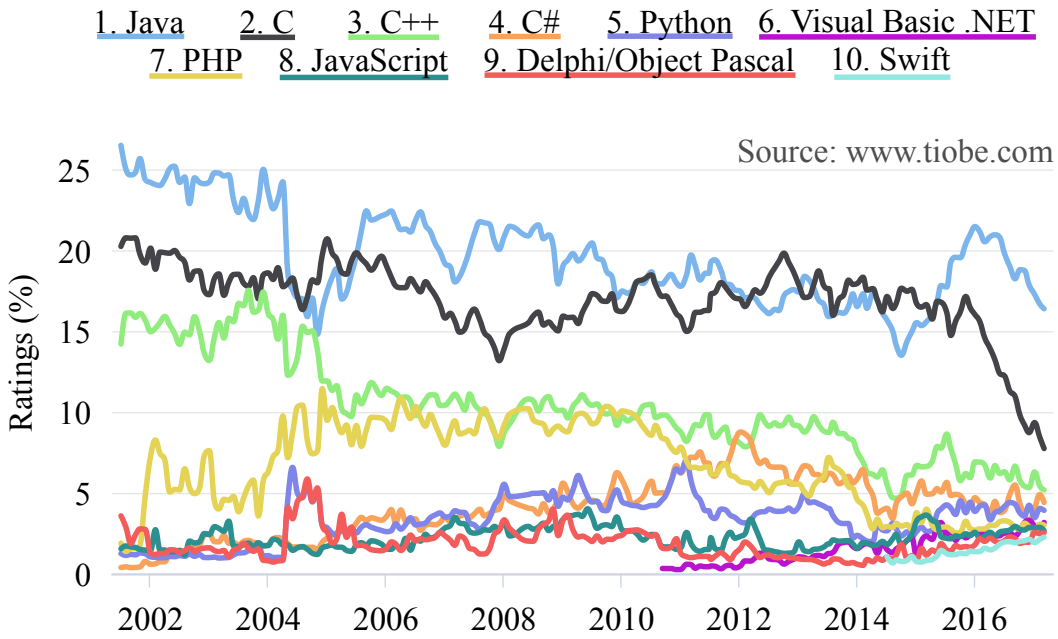


Figure 1.1: The TIOBE programming community index [TIO] indicating programming language popularity over the last 15 years of the 10 programming languages with the highest index in March 2017.

to a target machine. This design choice and the features of the language made Pascal fairly popular.

As it is indicated by the TIOBE programming community index [TIO] shown in Figure 1.1, the Java language is one of the most popular HLLs over the last 15 years. Java is a multi-threaded class-based object-oriented language targeting the *Java Virtual Machine* (JVM) [LYBB14]. It was one of the first languages to define a memory model; a memory model describes atomicity, visibility and order of memory accesses made by a thread with respect to other threads. The code is initially compiled to the V-ISA of the JVM called Java bytecode. Thus, Java applications can be distributed in the bytecode form, hiding the source code and providing portability. Java bytecode not only provides portability but also advanced security guarantees via bytecode validation and runtime checks. The Java runtime system targeted for a specific HW is capable of executing the bytecode on the target machine. There are a number of runtime environments for application virtualisation based on the similar principles such as the *Common Language Runtime* (CLR) [MWG00] and Android Runtime [Goo15]. As for the JVM and the CLR, they are not only used to run programs written in Java and C# ranked 1 and 4 respectively in Figure 1.1, but also programs written in other languages that can

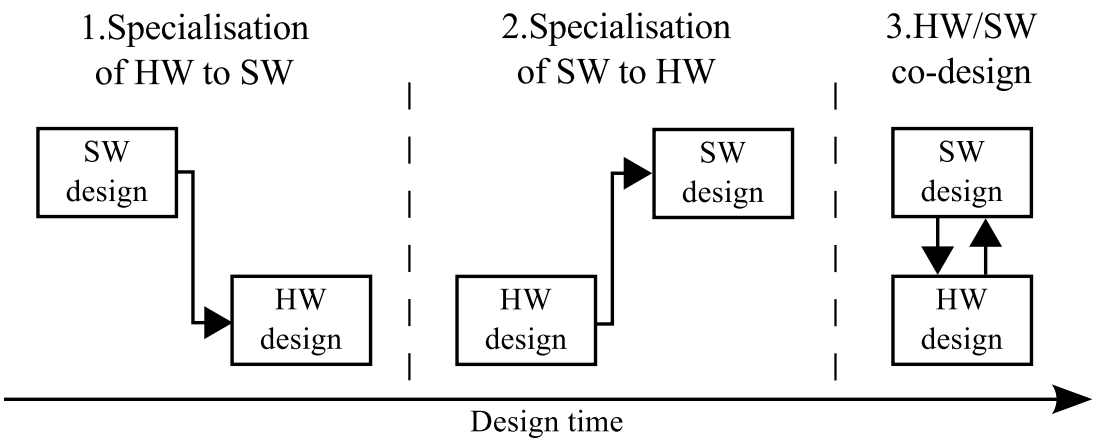


Figure 1.2: Patterns of dependent HW/SW evolution.

target these virtual machines. Visual Basic .NET applications are executed on the CLR, and Visual Basic .NET is ranked 6 in Figure 1.1. Python and JavaScript applications can be executed on the JVM via the Graal compiler [Gra16], and the Truffle [WW12] self-optimising *Abstract Syntax Tree* (AST) interpreter. Python and JavaScript are also among the most popular languages ranked 5 and 8 respectively in Figure 1.1.

To summarise, there is an ongoing trend in using HLL VMs in all domains of computing from Android applications running on Android Runtime on mobile phones to the Hadoop [Whi12] framework running on JVMs on servers in clouds. The main reasons for the popularity of HLL VMs are that they provide portability and adaptability of applications as well as certain security guarantees. The aforementioned properties of MREs make them suitable frameworks to tackle code generation and execution for future heterogeneous asymmetric many-cores.

1.3 Hardware/Software Co-Specialisation and Co-Design of General-Purpose CPUs

As it was identified above, HW is becoming more specialised, and one of the areas for specialisation of general-purpose CPUs can be HLL VMs which are widely used in modern computing. Below are the patterns of dependent HW/SW evolution that can be followed if one wishes to specialise a general-purpose CPU for specific SW or vice versa. These three patterns are schematicaly presented in Figure 1.2. Although these design patterns are applicable to different types of HW, this section will focus on general-purpose CPUs only.

1.3.1 Specialisation of Hardware to Software

The first pattern of dependent HW/SW evolution presented in Figure 1.2 is specialisation of HW to SW. As a HW design cycle is significantly more resource-consuming than a SW design cycle in many cases [BH09], this type of specialisation is rational when a certain SW pattern meets the following criterion:

1. HW specialisation yields significant gains for this SW pattern, and
2. the projected share of the SW pattern executed on the specialised HW during its lifecycle is significant.

This criterion is qualitatively represented by the Amdahl's law [Amd67]. By this law, when the fraction of the workload f is sped up s times and the other fraction of the workload $(1-f)$ is unchanged, the total speedup of the workload S is presented by the following formula:

$$S(f,s) = \frac{1}{(1-f)+f/s}$$

The same formula can be used to estimate how much less energy is required to run the workload if s represents how much less energy is required to run the fraction of the workload f. During a lifecycle of a specialised HW, the share of the SW pattern for which specialisation is done might change, and the fraction f in the equation above might change over time. Thus, the projection of the lifetime of the SW pattern and of possible modifications over time is critical, and possible modifications must be addressed by sufficient degree of programmability of the specialised HW to be able to adapt to them.

One of the most notable examples of HW to SW specialisation is the *Floating-Point Unit* (FPU) for floating-point arithmetic. Floating-point arithmetic operates with real numbers represented by a fixed number of bits. As real numbers are uncountable, only a subset of them can be represented in a floating point format. When making arithmetic operations on floating-point numbers results need to be approximated by a number within the representable subset. The arbitrary precision floating-point arithmetic can be performed in SW using integer numbers. However, floating-point arithmetic can be performed an order of magnitude more efficiently having an FPU [Int16]. As, floating-point arithmetic represents a significant share of general purpose computing, and it is well standardised [Int08], specialisation of modern CPUs by adding FPUs is common.

Among other wide-spread specialisations in general-purpose CPUs induced by SW are [RDS15]:

- *Single-Instruction Multiple-Data* (SIMD) extensions for vectorisable SW,
- extensions for synchronisation of multi-threaded SW,
- cryptographic extesnions for corresponding algorithms, and
- virtualisation extensions for *Virtual Machine Monitors* (VMM) allowing efficient simultaneous execution of different *Operating Systems* (OS) on the same HW.

1.3.2 Specialisation of Software to Hardware

The second pattern of dependent HW/SW evolution presented in Figure 1.2 is specialisation of SW to HW. As soon as new HW or its specification is available, SW can be specialised to take advantage of new features or parameters. There are two major ways of SW specialisation to new HW that can be applied together:

1. implicit specialisation via SW recompilation,
2. explicit specialisation via SW modification.

If new HW features are supported by a compiler, SW specialisation can happen via recompilation if the compiler is able to identify such opportunities and utilise new HW features. In case of floating-point arithmetic described in Section 1.3.1, finding opportunities for FPU utilisation can be facilitated by the fact that floating-point arithmetic is directly expressed in a language for which compilation is done. But for instance, specialisation of a single-threaded program for a multi-core CPU via recompilation is more challenging as thread-level parallelism is not explicitly expressed in a program and it needs to be extracted by a compiler. The practical capabilities of a conversion of a single-threaded program to a parallel multi-threaded code by compilers turned out to be rather limited [MJCM], so explicit modification of a program to a multi-threaded form is the main way of SW specialisation in a multi-core direction. Efficiency of SW specialisation can be quantitatively represented by the Amdhal's law described above.

If some projections can be done regarding HW modifications in the future, SW can be parameterised and contain mechanisms to adapt to them by changing parameters. One example of SW adaptation to possible changes of HW is the *Automatically Tuned Linear Algebra Software* (ATLAS) [WD98]. This parameterised SW runs a number of micro-benchmarks at installation time to determine the best parameters to adapt to

different sizes and levels of caches, latencies and other characteristics of CPUs for better efficiency.

1.3.3 Hardware/Software Co-Design

The final pattern of dependent HW/SW evolution presented in Figure 1.2 is HW/SW co-design. HW/SW co-design is a process of parallel design and co-specialisation of HW and SW in a single effort. In comparison with sequential design of SW and HW, HW/SW co-design reduces the risks of overdesigning or underdesigning both HW and SW and allows a better synergy between them.

A process of HW/SW co-design is an iterative process involving the following steps [Tei12]:

1. bi-partitioning of the system functionality between HW and SW,
2. design of the HW model and the SW prototype,
3. evaluation of the SW prototype on the HW model,
4. revision of steps 1 and 2.

One notable example of a HW/SW co-designed general-purpose CPU and an MRE is the Azul Vega CPU and the Azul JVM [Int17]. The Azul Vega processor features 54 general-purpose cores with additional support to facilitate execution of Java applications. The co-designed Azul VM was specifically designed to support unique features of the Azul Vega CPU to achieve better scalability and reduce delays during automatic memory management in comparison with non co-designed JVMs running on general-purpose CPUs.

The HW/SW co-design in this area may only partly target MREs and have broader application. For instance, Harris *et al.* proposed architectural support for dynamic filtering which can be successfully applied to both automatic memory management in MREs and implementations of software transactional memory [HTCU10].

Although the HW/SW co-design approach has been successfully used for several decades, its utilisation is projected to emerge in the next decades with the end of Moore's law [Tei12]. Firstly, HW and SW will need to be co-designed for extra reliablility and adaptability to faults as HW components with smaller transistors are more error-prone. Secondly, diminishing returns of technological advances in integrated circuit technologies will promote co-design of specialised HW and SW.

1.4 Research Aims

The general aim of this PhD is to research opportunities for SW specialisation and HW/SW co-design in the areas of MREs and general-purpose CPUs. After research space exploration in this broad multidisciplinary area, the general aim was narrowed down to three sub-aims covering different disciplines. These aims are the following:

1. The first aim is to explore the opportunities and limitations of specialisation of a SW synchronisation method for a many-core CPU. The selected method of synchronisation is barrier synchronisation and the selected many-core CPU is a many-core coprocessor named Intel Xeon Phi 5110P. Barrier synchronisation requires synchronisation of all threads at certain points of program execution and is provided in MREs via libraries. The aforementioned coprocessor features 60 4-multi-threaded cores allowing simultaneous execution of 240 threads. Thus, this synchronisation method and this CPU present an interesting research case to explore SW specialisation questions related to synchronisation of threads. In 2013, Java was not supported for programming Intel Xeon Phi[Ree13]. As barrier synchronisation is wide-spread and generic, specialisation of this synchronisation method can be researched in isolation from MREs. The MRE libraries, such as the .NET Framework Class Library [.NE17] and the Java Class Library [JCL17], contain implementations of the barrier synchronisation primitive, so the obtained results are directly applicable to MREs and any SW using barrier synchronisation. The practical goal of this research is to minimise time spent in barrier synchronisation on Intel Xeon Phi 5110P, and the result can be measured quantitatively as a speedup over the reference implementation of this method of synchronisation on this platform. This aim was pursued in 2013-2015.

2. The second aim of this work is to explore modelling techniques for HW/SW co-design in the area of MREs and general-purpose CPUs. These modelling techniques should be accurate enough to be able to achieve the 3rd aim. If modelling techniques are not satisfactory, the identified limitations should be addressed with novel modelling techniques. The practical goal of this work is to build a simulation platform integrating a simulator and a HLL VM implementation together. The result of this effort can be measured qualitatively in terms of usefulness of the developed platform by comparing its features against other state-of-the-art simulation platforms. This aim was pursued in 2014-2016.

3. As it was mentioned in Section 1.1, there is a trend in increasing the computing bitness and utilisation of HLL VMs. On current 64-bit architectures the address bus width is less than 64-bits, however addresses are operated via 64-bit granularity in registers and memory. Thus, some of the address bits are not carrying useful information. This fact motivates the exploration of the idea of utilisation of these unused address bits in the context of HLL VM implementations, as addresses (in these implementations) are usually associated with objects and can be used for storing some information associated with them. Thus, the third aim is to explore opportunities for SW/HW co-design for MREs and CPUs utilising these unused address bits. The practical goal of this research is to improve memory utilisation, performance and energy efficiency of HLL VM implementations, and the result can be measured quantitatively as an improvement in these characteristics for the selected state-of-the-art implementation. This aim was pursued in 2015-2017.

1.5 Contributions

Research in specialisation of barrier synchronisation algorithms to the Intel Xeon Phi coprocessor and opportunities of HW/SW co-design in utilisation of tagged pointers in HLL VMs resulted in the following contributions:

- An open-source multi-target parameterised framework for evaluating different barrier synchronisation algorithms and a thorough evaluation of the behaviour of current state-of-the-art barrier synchronisation algorithms specialised for Intel Xeon Phi.

- A novel hybrid barrier synchronisation algorithm for Intel Xeon Phi applicable to other multi-hierarchical architectures.

- MaxSim: a simulation platform for managed applications.

- Novel simulation techniques related to pointer tagging and address-space morphing on the x86-64 platforms.

- A method of storing array length in tagged pointers on architectures with tagged pointers support.

- Novel hardware extensions to the address generation and load-store units to achieve low-overhead array length retrieval from tagged object pointers.
- A method of type information elimination from objects on architectures with tagged pointers support.
- Novel hardware extensions to the address generation and load-store units to achieve low-overhead type information retrieval and tagged object pointers compression-decompression.

1.6 Publications

Part of the material in Chapters 2 and 3 was used in the following publication:

- EFFECTIVE BARRIER SYNCHRONIZATION ON INTEL XEON PHI COPROCESSOR. **Andrey Rodchenko**, Andy Nisbet, Antoniu Pop, and Mikel Luján. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing (Euro-Par), Vienna, Austria,* August 2015 [RNPL15].

Part of the material in Chapters 4, 5, and 6 was used in the following publications:

- HETEROGENEOUS MANAGED RUNTIME SYSTEMS: A COMPUTER VISION CASE STUDY. Christos Kotselidis, James Clarkson, **Andrey Rodchenko**, Andy Nisbet, John Mawer, and Mikel Luján. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Xi'an, China,* April 2017 [KCR+17].
- PROJECT BEEHIVE: A HARDWARE/SOFTWARE CO-DESIGNED STACK FOR RUNTIME AND ARCHITECTURAL RESEARCH. Christos Kotselidis, **Andrey Rodchenko**, Colin Barrett, Andy Nisbet, John Mawer, Will Toms, James Clarkson, Cosmin Gorgovan, Amanieu d'Antras, Yaman Cakmakci, Thanos Stratikopoulos, Sebastian Werner, Jim Garside, Javier Navaridas, Antoniu Pop, John Goodacre, Mikel Luján. In *Proceedings of the 10th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG), Vienna, Austria,* January 2016 [KRB+16].
- MAXSIM: A SIMULATION PLATFORM FOR MANAGED APPLICATIONS. **Andrey Rodchenko**, Christos Kotselidis, Andy Nisbet, Antoniu Pop, and Mikel

Luján. In *Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), San Francisco Bay Area, California, USA,* April 2017 **(Best Paper Award)** [RKN+17].

Part of the material in Chapter 7 was used in the following publication:

- TYPE INFORMATION ELIMINATION FROM OBJECTS ON ARCHITECTURES WITH TAGGED POINTERS SUPPORT. **Andrey Rodchenko**, Christos Kotselidis, Andy Nisbet, Antoniu Pop, and Mikel Luján. In *the IEEE Transactions on Computers (TC),* January 2018 [RKN+18].

1.7 Thesis Structure

Dependencies between the chapters and their classification are shown in Figure 1.3, so that the reader interseted in the certain areas of this thesis or having solid knowledge in some areas can skip certain chapters or parts of them. The chapters presenting the contribution of this work are finalised with the review of the related work and comparison of the obtained results. In case the reader wants to get familiar with the related work first, it is located at the end of Chapters 3, 6, and 7 right before conclusions.

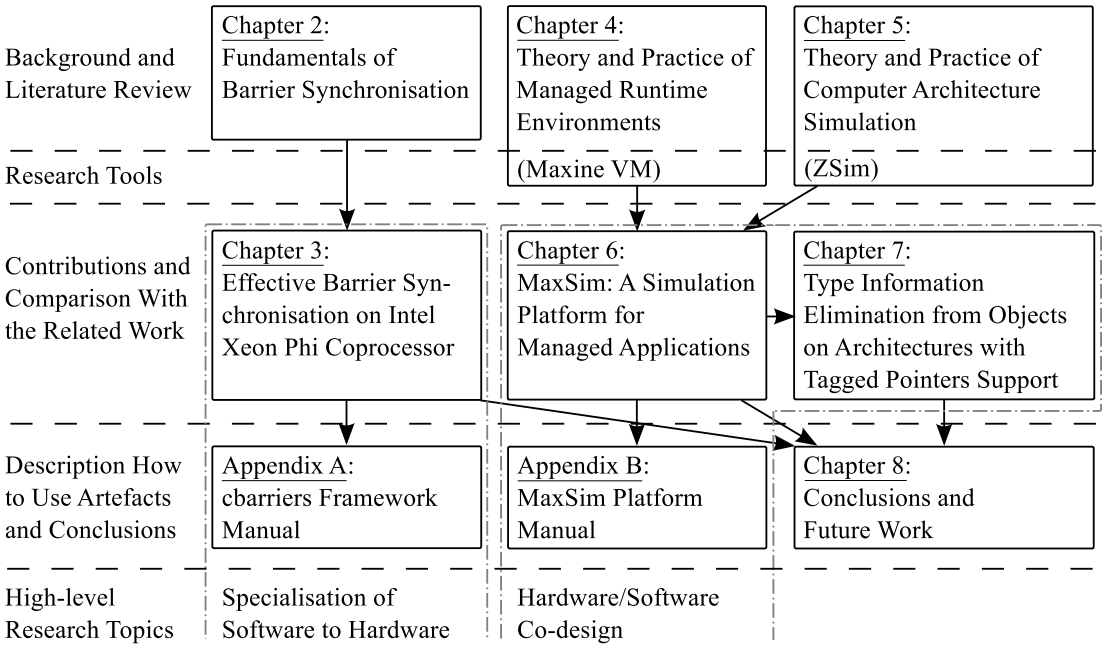


Figure 1.3: Dependencies between the chapters and their classification.

The thesis is structured in the following way:

Chapter 2 presents the fundamentals of barrier synchronisation on cache-coherent shared memory architectures. In addition, it explores various architectural parameters which are relevant to this type of synchronisation. Finally, it discusses state-of-the-art barrier synchronisation algorithms and forms an introduction to the work presented in Chapter 3.

Chapter 3 researches the problem of effective barrier synchronisation on the Intel Xeon Phi 5110P coprocessor. In addition, it discusses the key specifities of this architecture and evaluates the behavior of current state-of-the-art barrier synchronisation algorithms using the novel parameterised experimental framework. Finally, it proposes the novel hybrid synchronisation algorithm, that exploits the topology, the memory hierarchy and the streaming stores of the Xeon Phi architecture. The obtained results are compared with the results of the related barrier synchronisation algortithms specialised for the Intel Xeon Phi.

Chapter 4 presents the fundamentals of managed runtime environments with the focus on Java virtual machine implementations. In addition, it discusses the research Java VM implementation called Maxine as a use case in detail and compares it with other Java VM implementations.

Chapter 5 presents the fundamentals of computer architecture simulation. In addition, it discusses the research user-level software-based simulator called ZSim as a use case in detail and compares it with other simulators of its class.

Chapter 6 describes the platform for simulation of managed workloads called MaxSim. In addition, it presents novel simulation techniques such as modelling of tagged pointers on x86-64 architectures leveraging dynamic binary translation and address space morphing for modelling different object layouts. The novel platform is compared with the related platforms.

Chapter 7 researches the problem of handling type information on architectures with tagged pointers support. In addition, it explores an opportunity of storing class identifier in an object pointer tag removing a class information pointer from an object. Required changes to automatic memory management are proposed. The intial evaluation of this opportunity motivates proposal of novel extension to address generation and load-store units. Final evaluations with the proposed hardware extensions are reported.

Chapter 8 presents overall conclusions of the presented work and targets directions for future work.

Appendix A describes how to use the cbarriers framework for barrier algorithms

evaluation presented in Chapter 3.

Appendix B describes how to use the MaxSim simulation platform presented in Chapter 6.

Chapter 2

Fundamentals of Barrier Synchronisation

This chapter presents the fundamentals of barrier synchronisation on cache-coherent shared memory architectures. It provides a review of the state-of-the-art barrier synchronisation algorithms and explores architectural parameters which are relevant to such type of synchronisation. This chapter provides necessary background knowledge for specialisation of barrier synchronisation algorithms on the Intel Xeon Phi 5110P coprocessor, which will be presented in Chapter 3. Part of the material in this chapter was published in the Euro-Par 2015 conference proceedings [RNPL15].

2.1 What is Barrier Synchronisation?

Multi- and many-core systems have become the norm, and their efficient exploitation requires scalable synchronisation mechanisms. Synchronisation barriers are one of the fundamental synchronisation primitives, underpinning the parallel execution models of many modern shared-memory parallel programming languages such as Cilk [BJK+96], OpenCL [EBS17a] or OpenMP [EBS17b], and are one of the main challenges to scaling. **A synchronisation barrier is a logical point during a multi-threaded program execution separating two epochs of computations n and $n+1$ preserving the following property: each thread can proceed to an epoch$_{n+1}$ only when all threads finished execution of an epoch$_n$.** A logical diagram of barrier synchronisation is shown in Figure 2.1, and it consists of two phases: the *Registration Phase* and the *Notification Phase*. Similiar terminology was introduced by Hoefler et al. [HMMR04] in their survey of barrier synchronisation algorithms. During the *Registration Phase*

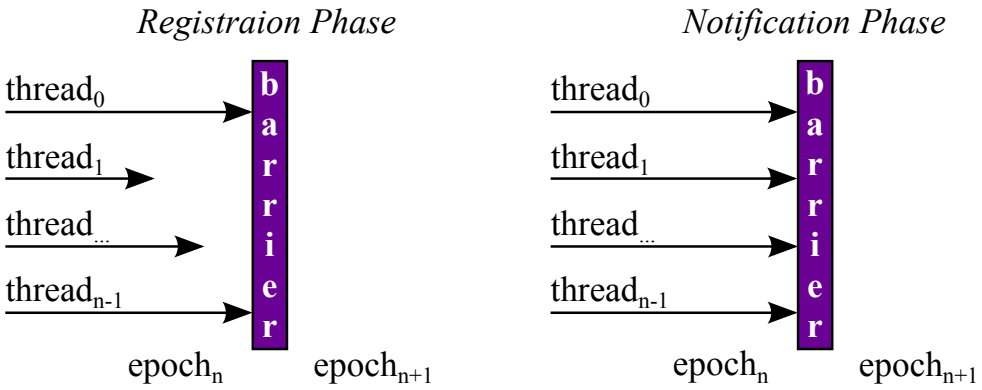


Figure 2.1: Logical diagram of barrier synchronisation.

the arrival of threads is registered, and when the last one arrives, then the *Notification Phase* starts. During the *Notification Phase* all threads are notified that they can proceed to an epoch$_{n+1}$ from an epoch$_n$.

Although barrier synchronisation can be performed via dedicated HW [SGC+06, SK10], if a dedicated HW support is not present, barrier synchronisation can be implemented via loading and storing values in memory. The following section will review major classes of memory architectures of multiprocessors with the emphasis on shared memory architecture, and this background knowledge is essential to comprehend barrier synchronisation algorithms on shared memory architectures.

2.2 Shared and Distributed Memory Architectures

With the advent of multiprocessors, researchers and engineers faced the problem of memory distribution between the cores. Memory architectures could be presented by two major types: *shared memory architecture* and *distributed memory architecture*. Shared memory architecture can be split into two subtypes: *symmetric shared memory architecture* and *distributed shared memory architecture*, the latter being a mixture of shared and distributed memory architectures. The aforementioned types of memory architecture are described below in this section, and the next paragraph explains the basic notions of *cache* and *cache coherence* mentioned in the description.

In order to reduce memory access latency and traffic, accessed memory is stored in multi-level caches. They are smaller in size but are located closer to processors and can be accessed faster. In multiprocessors, a maintenance of uniformity of shared data stored in different caches associated and accessed by different cores is called cache

coherence [SHW11], which ensures that data shared and cached by different processors is read by one processor in correspondence with writes by the other processors. Cache coherence between caches associated with different processors can be maintained manually in SW or automatically in HW through a *cache coherence protocol*. Cache coherence protocols can be classified into the two main categories by the type of its design: *directory-based* and *snooping*. Cache coherence and cache coherence protocols are described in detail in Section 2.3.

2.2.1 Symmetric Shared Memory Architecture

Symmetric shared memory is a type of memory architecture which is addressable by multiple processors through memory access operations provided by an ISA, and each processor is connected to a single physical memory by a shared bus [HP06]. Snooping protocols, described in Section 2.3, dominate on symmetric shared memory architectures because they are able to utilise a shared-memory bus to query the state of the caches. With respect to the scalability of the many-core systems, it is impractical to follow a symmetric shared memory architecture design due to bandwidth, delay and power limitations of a centralised memory bus.

2.2.2 Distributed Shared Memory Architecture

In a distributed shared memory architecture, as the name implies, memory is distributed across the nodes of a chip and linked by interconnection network, but each processor has a view of the distributed memory as a whole through the shared memory abstraction. Thus, all memory nodes are addressable by all processors in the system by memory access instructions, but the latency of access depends on the disposition of the memory and accessing processor in the interconnect with regard to interconnect topology. This phenomenon is called *Non-Uniform Memory Access* (NUMA), and programmes executed on distributed shared memory architecture can observe NUMA latencies. Furthermore, on distributed shared memory many-core architectures the last level cache can be physically distributed and cache coherence can be maintained via a distributed directory-based cache coherence protocol, described in Section 2.3. Programmes executed on such systems can experience not only NUMA but also *Non-Uniform Cache Access* (NUCA) latencies. The main concern for NUMA and NUCA

architectures is not only increasing data locality but also reducing congestion on memory controllers and interconnects [DFF+13] [MG11]. Directory-based cache coherence protocols dominate on NUMA architectures, though some architectures like the Cray T3D [Ada92] do not provide cache coherence, and so cache coherence needs to be maintained explicitly in SW.

2.2.3 Distributed Memory Architecture

In a distributed memory architecture memory is fragmented, with the fragments being privately owned by processing units. Direct addressing of remote memory is not possible through memory access operations but is performed indirectly through passing messages to remote processors. Such architectures do not require cache coherence and are scalable, but require programs to be written following the *message passing* communication model.

This section presented three types of memory architectures and introduced the notion of cache coherence. The rest of the thesis will focus on cache-coherent shared memory architectures, and the next section will review cache coherence protocols and the related notion of memory consistency.

2.3 Cache Coherence Protocols and Memory Consistency

In the previous section, the notion of cache coherence was introduced. Formally, cache coherence is the maintenance of the following two coherence invariants [SHW11]:

1. At any point in time when a certain memory location is accessed, either a single core in a multi-core system can be writing to this memory location or several cores can be reading from this memory location.

2. The value of a certain memory location read by a core should be equal to the value of the most recent write to this memory location.

2.3.1 Cache Coherence Protocols

The implementation of cache coherence can be logically described by finite state machines called *coherence controllers* [SHW11], associated with a memory location and

dedicated blocks (also known as lines) in caches. Coherence controllers co-operate via sending messages to each other. A coherence protocol defines messages and interactions between coherence controllers and changes of their states. Cache coherence protocols can be classified into the two major categories by the type of its design:

1. **Snooping protocol.** In this protocol, every coherence controller broadcasts messages to all other coherence controllers. Thus, every coherence controller in such a protocol has full awareness of all the other controllers, and it can change its state with respect to other controllers to preserve cache coherence invariants.

2. **Directory-based protocol.** In this protocol, one coherence controller also known as *home* acts as an arbiter, and all requests are addressed to this home controller. This home controller maintains a *directory* giving the name for this class of protocols. This directory contains information on the states of all the other controllers, so only the home controller has full awareness of all the other controllers and generated responses to messages received from other controllers.

Both protocols are commonly implemented in a single many-core system to preserve cache coherency at different levels of hierarchy. In this case, a snooping protocol is usually implemented to preserve cache coherency within groups of caches located close to each other, while a directory-based protocol is used to preserve cache coherence globally between such groups. Examples of such many-core processors are Intel Xeon Phi [Int14] and Sparc M7 [Phi14].

The other important feature of cache coherence protocols is a set of logical *stable states* at cache controllers [SHW11]. Cache controllers maintain cache coherence by changing these states for a group of cache lines associated with the same memory location. One of the widely used sets of cache coherence states is called MESI [PP84], which is named after the following four states:

- *Modified* (M): data inside a cache line with this state is different from what is stored in the memory location, and it is not present in the other caches.

- *Exclusive* (E): data inside a cache line with this state is the same as in the memory location, and it is not present in the other caches.

- *Shared* (S): data inside a cache line with this state is the same as in the memory location, and it may be present in the other caches.

- *Invalid* (I): data inside a cache line with this state is not valid.

The *Modified* state in the MESI protocol enables multiple reads and writes to a memory location privately used by a single core in a many-core system without the necessity to write data back to memory on every write operation saving memory bandwidth. Addition of the 5th *Owned* (O) state enables sharing of modified data between caches without writing it back to memory using the MOESI protocol [SS86]. The latencies necessary to change these coherence states incur significant performance penalties [HMN09].

2.3.2 Memory Consistency

Each thread can perform memory access operations during multi-threaded program execution. The possible behaviour of these memory operations in respect to each other is defined by a *memory consistency model*.

The most intuitive consistency model is called *Sequential Consistency* (SC) model [Lam79]. This consistency model declares that all memory access operations performed by different threads can be ordered in a single sequence, and in this sequence, for every read operation the value read from a memory location will be equal to the value stored by the most recent write to this memory location in the sequence. Although the sequential consistency model is very clear and intuitive for programmers to write correct multi-threaded programs, its implementation in many-core CPUs can cause significant delays during memory access operations.

In order to reduce these delays, more relaxed memory consistency models were proposed. One of such widely used classes of memory consistency models is called *Total Store Order* (TSO). The two notable examples of consistency models from this class are SPARC-TSO and x86-TSO [SSO+10]. TSO has the following relaxation in comparison with SC model: loads within a thread can be performed before independent store operations during execution. This relaxation allows reducing latency of load operations in CPUs implementing this model as they are not constrained by completion of independent store operations in case of SC. During multi-threaded execution, loads-stores reordering allowed by TSO can lead to different valid ways of a program execution. Listing 2.1 presents two snippets of code executed in parallel by two threads on different cores. The code snippet on the left is executed by the first thread on the first core, while the code snippet on the right is executed by the other thread on the other core. If this code is executed on CPUs implementing TSO memory model, both threads can print messages that the value of the memory locations `a` and `b` are zeros, although after execution of both sections of code by both threads the values in the

memory locations `a` and `b` will be eventually one. When the same code is executed on a CPU implementing sequential consistency model, the aforementioned behaviour is prohibited and either one (`"a == 0"` or `"b == 0"`) or zero messages will be printed.

```
// initially: a == 0 && b == 0
// two threads are executed in parallel on different cores

// executed by the 1st thread   // executed by the 2nd thread
// on the 1st core              // on the 2nd core
{                               {
  a = 1;                          b = 1;
  if ( b == 0 ) {                 if ( a == 0 ) {
    print( "b == 0");                 print( "a == 0");
  }                               }
}                               }

// if finally both "a == 0" and "b == 0" are printed:
//   TSO memory consistency model
// else:
//   TSO or SC memory consistency models
```

Listing 2.1: Example allowing to distinguish SC and TSO memory consistency models.

Thus, TSO is considered to be a weaker memory model than SC. CPUs with weaker than SC memory models usually provide explicit instructions called *memory barriers* or *memory fences*, which are inserted between two memory access operations for which memory reordering should be constrained. Alternatively, in some ISAs reordering constraints can be specified in opcodes of memory access instructions (*e.g.* IA-64). There are many other consistency models weaker than TSO, but for the rest of the thesis, the descriptions of the TSO and sequential memory consistency models are sufficient.

2.4 Barrier Synchronisation Algorithms for Shared Memory Architectures

In this section, state-of-the-art barrier synchronisation algorithms are reviewed for shared memory architectures with TSO memory consistency. Store operations related to barrier synchronisation will appear as **`store`**`( <address>, <value>)` in all code listings. The implementation of this function is architecture-dependent, and the default implementation is a basic store operation available on a given architecture. The

other common function call to appear in the listings is a call to **pause**( ) method. As it will be shown later, barrier synchronisation via shared memory requires periodic checks of values related to notification of threads to be able to proceed to the next epoch of computation. In this case, the **pause**( ) function can implement control over when the next notification check is done. The implementation of this function is architecture-dependent, and the default implementation is an empty method so that the call to **pause**( ) can be completely eliminated in this case. Furthermore, each barrier implementation relies on shared and thread-private data, passed as arguments to the barrier **<prefix>_wait**(...) function. Shared data is accessed through a pointer to a shared data structure of type **<prefix>_Bar_t** which contains the appropriate data fields for each barrier, and thread-private data uses a structure of type **tp_t**.

2.4.1 Sense-Reversing Centralised Barrier

One of the simplest and least scalable barrier synchronisation algorithms is the *sense-reversing centralised barrier*. This algorithm, presented in Listing 2.2, uses two shared variables, a synchronisation counter `bar->count` and a flag `bar->sense`, and one thread-local variable, `tp->sense`. The synchronisation counter is initialised with the number of threads partaking in the barrier, `bar->num_threads`, the global shared sense is set to `1` and all local flags are set to `0`.

```
void                        // sense-reversing centralised barrier
sr_wait( sr_Bar_t * bar, // shared data
         tp_t * tp)      // thread-private data
{
  if ( !add_and_fetch( & bar->count, -1) )
  { // last thread arrives
    store( & bar->count, bar->num_threads);
    store( & bar->sense, tp->sense);
  } else
  { // other threads arive
    // threads wait for release notification
    while ( tp->sense != bar->sense )
      pause( );
  }

  // sense is reversed
  store( & tp->sense, !tp->sense);
}
```

Listing 2.2: Sense-reversing centralised barrier.

Upon reaching a barrier, each thread registers its arrival by atomically decrementing the synchronisation counter `bar->count`, and then waits while the value of the global flag `bar->sense` differs from its local flag `tp->sense`. The last thread to complete the atomic operation, when the counter reaches `0`, re-initialises the counter and reverses the global sense of the barrier. The other threads eventually perceive that the global sense has changed and they pass the barrier flipping their own flags. This technique, called *sense-reversing*, allows the reuse of the same barrier variables for the next synchronisation round. This is one of the simplest, and least scalable, algorithms. It relies on an expensive, sequentially consistent, atomic operation and suffers from contention on accesses to global shared variables.

2.4.2 Combining Tree Barrier

To reduce the contention on shared variables, the *combining tree barrier* [YTL87] (see Listing 2.3) organises the participating threads in a tree, using an algorithm similar to a centralised barrier at each node of the tree: the last thread to decrement the synchronisation counter of a node recursively proceeds to decrement the counter of its parent node. At each level, threads that do not arrive last all wait on node-level release flags for notification that the barrier has passed. It is also possible to use a global release flag, trading a shorter critical path for additional contention on the global release flag.

```
void                          // combining tree barrier
ctr_wait( ctr_Bar_t * bar,    // shared data
          tp_t * tp,          // thread-private data
          ctr_Node_t * node) // tree node
{
  if ( !add_and_fetch( & node->count, -1) )
  { // last thread arrives
    store( & node->count, node->num_threads);

    // thread recursively participates at the next level
    if ( node->parent )
      ctr_wait( bar, tp, node->parent);

    // thread notifies siblings that the barrier is passed
    store( node->sense, tp->sense);
  } else
  { // other threads arrive
    // threads wait for release notification
    while ( tp->sense != node->sense )
      pause( );

    // sense is reversed
    store( & tp->sense, !tp->sense);
  }
}
```

Listing 2.3: Combining tree barrier.

2.4.3 Static Tournament Barrier

The combining tree barrier, described in Section 2.4.2, is an efficient solution to avoid contention, yet it still relies on atomically decremented counters. To avoid all atomic operations, Hensgen *et al.* proposed the *static tournament barrier* [HFM88], starting from the realisation that the atomic operation was, in large part, required to reach a consensus on which thread arrives last, the *winner* in their terminology, this barrier does not discriminate between threads based on their arrival order, but relies on a statically determined thread, defined for each node of the tree, that will automatically progress to the next round once all other threads, the *losers*, have arrived. This is illustrated in Listing 2.4, where each node is initialised with a statically-defined winner. In this way, it is only necessary to determine whether all threads are accounted for, irrespectively of their arrival order. Checking for the arrival of threads, however, can be expensive. The

original implementation suggested restricting the arity of the tree to ensure that a single load instruction would be sufficient. For example on 32-bit architectures, registration of a thread arrival **register_arrival**(<address>, <value>, <thread_id>) can be implemented with a single 1-byte store instruction writing to a byte in a word dedicated to the arriving thread, and a check of all threads arrival associated with a node **is_registry_full**(<address>, <value>) requires only a 4-byte load of the dedicated word.

```
void                              // static tournament barrier
strn_wait( strn_Bar_t * bar,      // shared data
           tp_t * tp,             // thread-private data
           strn_Node_t * node)    // tree node
{
  if ( tp->id == node->winner_id )
  { // winner thread for this node arrives
    // thread waits for all threads expected at this node
    while ( !is_registry_full( & node->arrival_registry, tp->sense) )
      pause( );

    // thread recursively partakes at the next tree level
    if ( node->parent )
      strn_wait( bar, tp, node->parent);

    // thread sends notification to siblings
    store( & node->sense, tp->sense);
  } else
  { // loser threads for this node arrive
    register_arrival( & node->arrival_registry, tp->sense,
        tp->thread_id);

    // loser threads wait for release notification
    while ( tp->sense != node->sense )
      pause( );
  }

  // sense is reversed
  store( & tp->sense, !tp->sense);
}
```

Listing 2.4: Static tournament barrier.

The initial version of the algorithm by Hensgen et al. [HFM88] was later improved by Mellor and Crummey [MCS91] with a tree-based notification phase and sense-reversing to avoid re-initialising the barrier state; both included in Listing 2.4. As

a generalisation of the static tournament approach, the static f-way tournament was proposed by Grunwald et al. [GV94] where the number of participants in one round can be chosen arbitrarily and the initialisation algorithm allows the construction of more balanced trees.

2.4.4 Dynamic Tournament Barrier

The dynamic f-way tournament barrier [GV94] is an evolution of the static f-way tournament barrier in the sense that the winner of a round is determined dynamically. This algorithm is presented in Listing 2.5.

```
void                            // dynamic tournament barrier
dtrn_wait( dtrn_Bar_t * bar,    // shared data
           tp_t * tp,           // thread-private data
           dtrn_Node_t * node)  // tree node
{
  // all threads register arrival
  register_arrival( & node->arrival_registry, tp->sense,
     tp->thread_id);

  // store-load fence for the TSO memory consistency model
  memory_barrier( );

  if ( is_registry_full( & node->arrival_registry, tp->sense) )
  { // winner threads for this node arrive
    // threads recursively partake at the next tree level
    if ( node->parent )
      dtrn_wait( bar, tp, node->parent);

    // threads send notification to siblings
    store( & node->sense, tp->sense);
  } else
  { // loser threads for this node arrive

    // loser threads wait for release notification
    while ( tp->sense != node->sense )
      pause( );
  }

  // sense is reversed
  store( & tp->sense, !tp->sense);
}
```

Listing 2.5: Dynamic tournament barrier.

The winning thread will identify itself as a winner by checking the value of adjacent memory locations marked by other threads upon arrival. This solution is more efficient than a static tournament when a static winner comes early and busy-waits on a location yet to be set by one or multiple losers. The dynamic tournament approach comes with two caveats. The first caveat is a possibility to have multiple threads winning at the same node, which would introduce some overhead, but without impacting the correctness of the synchronisation. The second caveat is that this form of synchronisation requires a memory fence **`memory_barrier`** between thread registration **`register_arrival`**`( <address>, <value>, <thread_id>)` and checking for arrival status **`is_registry_full`**`( <address>, <value>)` on architectures using the TSO memory consistency model. Such a fence is necessary to avoid a case where no winner emerges for a node, in the case similar to the one presented in Listing 2.1 when both threads load the unchanged value. In the case of the sequential memory consistency model, **`memory_barrier`** is not necessary, and it is eliminated.

2.4.5 Dissemination Barrier

A shortcoming shared by all previous algorithms is that they all require two phases: a *Registration Phase* where threads reaching the barrier communicate to let others know that they arrived; and a *Notification Phase* where once a consensus is reached, the decision is propagated to all threads. However, these phases can be merged, at the cost of additional communication, by providing each thread with sufficient information to locally decide when the barrier can be passed. The first instance of this class of barriers was the *butterfly barrier* [Bro86], with the data flow topology similar to that of the *Fast Fourier Transform* (FFT). When the number of threads participating in the barrier is not a power of 2, the *dissemination barrier* [HFM88] can be a more efficient solution. This algorithm is presented in Listing 2.6.

In the dissemination barrier, a thread i in round r notifies another thread $j = (i+2^r)\, mod\, N$ by writing its local flag, which is sense-reversed, to a dedicated memory location, where N is the number of threads, $i, j \in [0, N-1]$ and $r \in [0, \lfloor \log_2 N \rfloor]$. Each thread can proceed to the next round as soon as its notification variable is set to the appropriate value for the current round. In this way, synchronisation is achieved in $\lceil \log_2 N \rceil$ rounds. The *f-way dissemination barrier* [HMMR06] is a generalised version, where each thread can notify f other threads in one round, requiring only $\lceil \log_{f+1} N \rceil$ rounds to complete. For example, a 2-way dissemination barrier can synchronise 9

threads in 2 rounds of communication, while an ordinary (1-way) dissemination barrier will require 4 rounds. If $f = N - 1$, then it will be a broadcast barrier (all-to-all communication) requiring $N * (N - 1)$ notifications.

```
void                            // dissemination barrier
dsmn_wait( dsmn_Bar_t * bar, // shared data
           tp_t * tp)        // thread-private data
{
  int parity = tp->parity;
  int sense = tp->sense;

  // for each round:
  for ( int r = 0; r < bar->roundsNum; ++r )
  {
    // thread notifies its arrival in this round
    store( & tp->partnerFlags[parity][r]->data, sense);

    // thread waits for release notification in this round
    while ( tp->myFlags[parity][r]->data != sense )
      pause( );
  }

  // sense is reversed if parity is one
  if ( parity == 1 )
    store( & tp->sense, !sense);

  // parity is reversed
  store( & tp->parity, 1 - parity);
}
```

Listing 2.6: Dissemination barrier.

In respect to data flow topology, the barrier synchronisation algorithms via shared memory presented in this section are logically shown in Figure 2.2. Reads and writes to shared memory locations are shown in red (*Registration Phase*) and green (*Notification Phase*), while execution of threads is shown in black.

The operation of the sense-reversing centralised barrier described in Section 2.4.1 can be interpreted using the corresponding diagram in Figure 2.2 as follows. The synchronisation counter `bar->count` is represented by the red rectangle, and the global flag `bar->sense` is depicted by the green rectangle. Upon reaching the barrier, each thread registers its arrival by atomically decrementing the synchronisation counter `bar->count`, and the corresponding memory accesses are shown by the solid red lines. After decrementing the counter, each thread waits while the value of the global flag

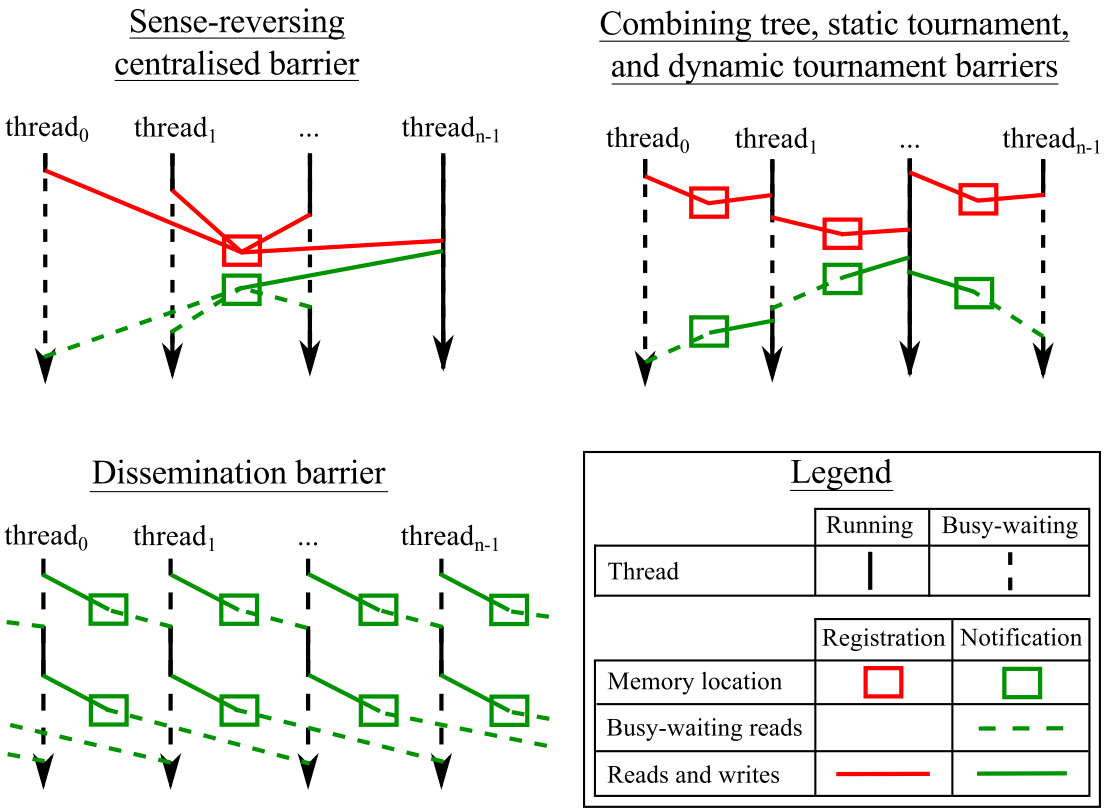


Figure 2.2: Diagrams of barrier synchronisation algorithms via shared memory.

`bar->sense` differs from its local flag `tp->sense`, and this busy-waiting is shown by the vertical dashed black lines. It should be noted that only shared memory locations are represented by rectangles on these diagrams, thus the local flags and other thread-private variables are not depicted. The last thread performing the atomic operation is the rightmost on the diagram, and it re-initialises the counter, and this is depicted by the solid red line. Then, this thread reverses the global flag of the barrier, and this is represented by the solid green line. The other threads eventually perceive that the global flag `bar->sense` has changed, and this is shown by the dashed green lines. After reversing the local flags and passing the barrier, all the threads continue execution, and this is shown by the solid black lines with arrows indicating the direction of execution. The other barrier synchronisation algorithms can be interpreted using such diagrams in a similar way.

If latencies of memory access operations are dependent on mutual co-location of cores on which threads are executed and memory locations used in synchronisation, such diagrams are useful for reasoning about the mapping of logical variables used in

barrier synchronisation algorithms to physical memory cells on target many-core systems. On distributed shared memory architectures with NUMA and NUCA latencies, memory access latencies depend on the aforementioned co-location, so these diagrams allow to reason how it can affect the total latency of passing a synchronisation barrier.

2.5 Summary

In this chapter, the barrier synchronisation data flow topologies were discussed, and the state-of-the-art algorithms of barrier synchronisation via shared memory were reviewed. It was shown that memory architecture, memory consistency and cache coherence protocols are fundamental factors for correctness and efficiency of these algorithms. Specialisation of the discussed state-of-the-art barrier synchronisation algorithms on the Intel Xeon Phi 5110P coprocessor will be presented in the next chapter.

Chapter 3

Effective Barrier Synchronisation on Intel Xeon Phi Coprocessor

In this chapter, the efficiency of the five barrier synchronisation algorithms described earlier is evaluated on the Intel Xeon Phi 5110P coprocessor. The practical goal of this research is to minimise time spent in barrier synchronisation on Intel Xeon Phi 5110P, and the result can be measured quantitatively as a speedup over the reference implementation of this method of synchronisation on this platform. A novel hybrid barrier synchronisation algorithm is presented that exploits the topology, the memory hierarchy and streaming stores of the Xeon Phi architecture to achieve a 3.28× lower overhead than the Intel OpenMP barrier implementation (ICC 14.0.0), thus outperforming all other known implementations for this architecture. Algorithms specialised for Intel Xeon Phi are evaluated on the CG and MG kernels from the NAS Parallel Benchmarks, the direct N-body simulation kernel and the EPCC barrier OpenMP microbenchmark. The optimised barriers presented in the chapter are available at `https://github.com/arodchen/cbarriers` released as free software. Appendix A describes how to use the cbarriers framework. Most of the material in this chapter was presented at the *Euro-Par 2015* conference [RNPL15]. The additional material, not presented in the conference, is related to refined utilisation of non-globally ordered streaming stores in the hybrid barrier synchronisation algorithm.

3.1 Introduction

The execution time optimisation of software barrier synchronisation has been widely studied [Bro86, YTL87, HFM88, MCS91, GV94, HMMR06], yet no algorithm has

proven to be optimal across the wide variety of parallel architectures. Indeed, each algorithm comes with its own set of trade-offs with respect to data flow topology, lengths of the critical paths, and memory consumption. For any given architecture, the optimal algorithm is largely dependent on factors such as the system's topology, the structure of the memory hierarchy, and the characteristics of the cache coherence protocol.

The focus of this chapter is to analyse the performance of barrier synchronisation algorithms on the Intel Xeon Phi 5110P coprocessor and to specialise these algorithms for this coprocessor utilising its advanced features. Based on the Intel *Many Integrated Core* (MIC) architecture, that provides a commodity off-the-shelf many-core system, the Xeon Phi coprocessors of the Knights Corner product line have up to 61 cores, each 4-way multi-threaded, for running a maximum of 244 logical threads. At this scale, the efficiency of barrier synchronisation is crucial for performance in synchronisation-intensive workloads.

The **first contribution** presented in this chapter is a thorough evaluation of the behaviour of current state-of-the-art barrier algorithms, and an analysis of their trade-offs for the memory hierarchy of Xeon Phi. It is shown that while the best algorithm depends on runtime conditions, a single statically chosen algorithm is only marginally outperformed in a small number of cases. The **second contribution** is a novel and more efficient hybrid barrier synchronisation algorithm, mixing different (existing) barrier algorithms at different levels of granularity of synchronisation, and optimised with *streaming store* instructions to write full cache lines, that eliminate a costly *read-for-ownership* cache coherency operation, which combines a read of the cache line and its invalidation in other caches where it is shared. It is shown that the hybrid approach outperforms in execution time all previous algorithms on the Intel Xeon Phi 5110P coprocessor.

Section 3.2 presents key features of Xeon Phi. Specialisation of the state-of-the-art barrier synchronisation algorithms for the Xeon Phi coprocessor and the new hybrid synchronisation scheme are presented in Section 3.3. Section 3.4 presents the experimental findings. Finally, Section 3.5 discusses previous work on Xeon Phi barrier synchronisation optimisation and Section 3.6 summarises this chapter. The optimised barriers presented in this chapter are available at `https://github.com/arodchen/cbarriers` released under the Apache v2.0 free software license. This open-source multi-target parameterised framework for evaluating different barrier synchronisation algorithms is the **third contribution**.

3.2 Intel Xeon Phi 5110P Coprocessor

Specialisation of the barrier synchronisation algorithms described in the previous chapter was done for the 60-Core Intel Xeon Phi 5110P coprocessor with a maximum clock frequency of 1.053GHz. As of 2014, when most of the work presented in this chapter was done, this coprocessor belonged to the latest generation of the MIC architecture family, and it was one of the most advanced general-purpose many-core processors available on the market. Xeon Phi 5110P cores are in-order, and they are connected using a bidirectional ring interconnect, as shown in Figure 3.1. Each core is 4-way multi-threaded and has 32 KB of instruction and 32 KB of data caches, as well as 512 KB of the dedicated L2 cache (inclusive of L1 cache). The coherence of the L2 cache lines on different cores is controlled by the distributed tag directories implementing the GOLS cache coherence protocol [Int14]. The coherence of L1 and L2 caches within the core is maintained by a modified MESI cache coherence protocol. However, the GOLS protocol makes it possible to emulate the *Owner* state, enabling a MOESI-like functionality. The *Owner* state indicates that the cache line is both shared and modified. 8GB of GDDR5 RAM is accessed through 8 dual channel memory controllers connected through a ring interconnect interface, as shown in Figure 3.1.

The 512-bit *Single-Instruction Multiple-Data* (SIMD) instructions were used to optimise barrier synchronisation in the *SIMD barrier* [CDM13]. SIMD stores, also known as *streaming stores*, use a vector size matching that of a cache line. As a result, such store instructions do not need to issue a *read-for-ownership* request in the cache coherence protocol. Store instructions with the *no-read* hint can be either *globally ordered*, providing the x86-TSO consistency, or *non-globally ordered*, leading to a

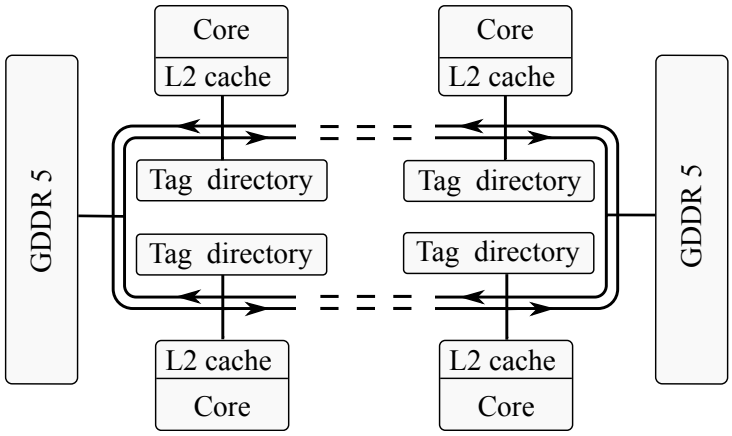


Figure 3.1: Architecture of the Intel Xeon Phi 5110P coprocessor.

weaker memory consistency. Thus, the Intel Xeon Phi ISA has the two following types of streaming stores [Int12]:

- `vmovnrap[d/s]` - vector store instructions with the no-read hint, globally ordered (*total store order* type consistency).
- `vmovnrngoap[d/s]` - vector store instructions with the no-read hint, but without enforcing a global ordering leading to weaker memory consistency, so that stores executed later in the same threads can be observed before them. A memory fence operation `lock addl $0x0,(%rsp)` should be used to prevent re-ordering of store operations in the cases when x86-TSO consistency is required.

The execution of HW threads can be paused, using the `delay r32/r64` instruction [Int12], which forces the processor to halt the fetch and issue of further instructions for a parametric number of cycles. This instruction can be used in barrier synchronisation algorithms during busy-waiting.

3.3 Barrier Synchronisation Specialisation for Intel Xeon Phi

3.3.1 Busy-Waiting Amortisation

Busy-waiting on the same memory location in the barrier synchronisation algorithms presented in the previous chapter can have a negative impact on performance because of the increased memory traffic, leaving less bandwidth to other threads. This effect was previously noted [AC89]. The appropriate delay in the specialised **`pause`**`( )` method was determined empirically (see Sect. 3.4) to amortise the impact of busy-waiting while balancing the additional latency this introduces for a thread to perceive a store operation. On the Intel Xeon Phi 5110P coprocessor, the delay can be introduced with the **`_mm_delay_32`**`( <number_of_cycles>)` intrinsic, as shown in Listing 3.1, the parameter `DELAY_IN_CYCLES` specifies the number of idle cycles.

```
#define DELAY_IN_CYCLES (...)

inline static void
pause( )
{
  _mm_delay_32( DELAY_IN_CYCLES);
}
```

Listing 3.1: Busy-waiting delay.

3.3.2 Streaming Stores

Streaming stores can reduce barrier overhead [KKC+13] when storing notification values to flags in the *Notification Phase* or reinitialising counters in the combining tree or sense-reversing centralised barrier. Listing 3.2 details the implementation of the **store**(<address>, <value>) function specialised for the Intel Xeon Phi 5110P co-processor.

```
static inline void
store( volatile void *addr, int data)
{
  _mm512i siVec = _mm512_set1_epi32( data);

#ifdef ARCH_STORE_NR
  // globally ordered streaming store
  _mm512_storenr_ps( addr, _mm512_castsi512_ps( siVec));
#endif

#if defined(ARCH_STORE_NR_NGO) || defined(ARCH_STORE_NR_NGO_REFINED)
  // non-globally ordered streaming store
  _mm512_storenrngo_ps( addr, _mm512_castsi512_ps( siVec));

# ifdef ARCH_STORE_NR_NGO
  // memory fence
  asm volatile( "lock; addl $0,(%rsp)\n");
# endif
#endif
}
```

Listing 3.2: Utilisation of streaming stores.

To satisfy the x86-TSO consistency requirements of the algorithms presented in the previous chapter, the globally ordered version of the streaming stores should be used (when ARCH_STORE_NR is defined). The second option to satisfy the x86-TSO

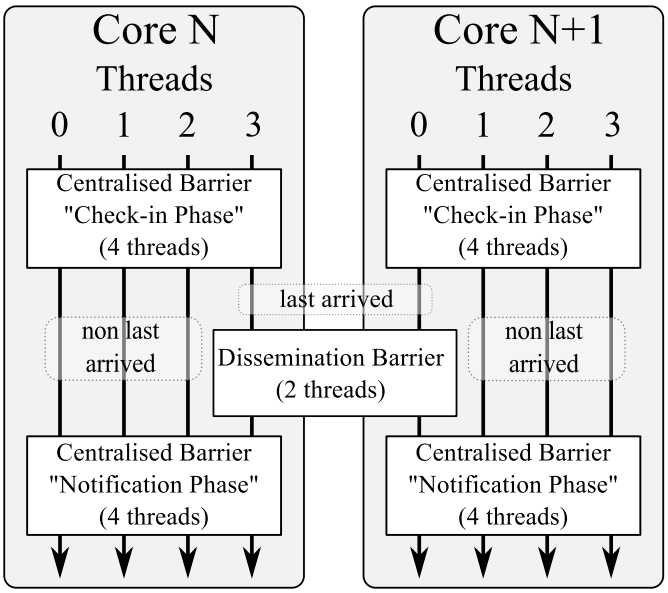
consistency requirements of the described algorithms is to use the non-globally ordered version of the streaming stores followed by the memory fence operation (when `ARCH_STORE_NR_NGO` is defined). The last option is to use non-globally ordered streaming stores, but in comparison with the previous case, not directly followed by the memory fence operations (when `ARCH_STORE_NR_NGO_REFINED` is defined). In this case, the presented algorithms should be modified via insertion of memory fences. Such a refinement will be described for the hybrid barrier synchronisation algorithm presented below.

3.3.3 Hybrid Barrier Synchronisation

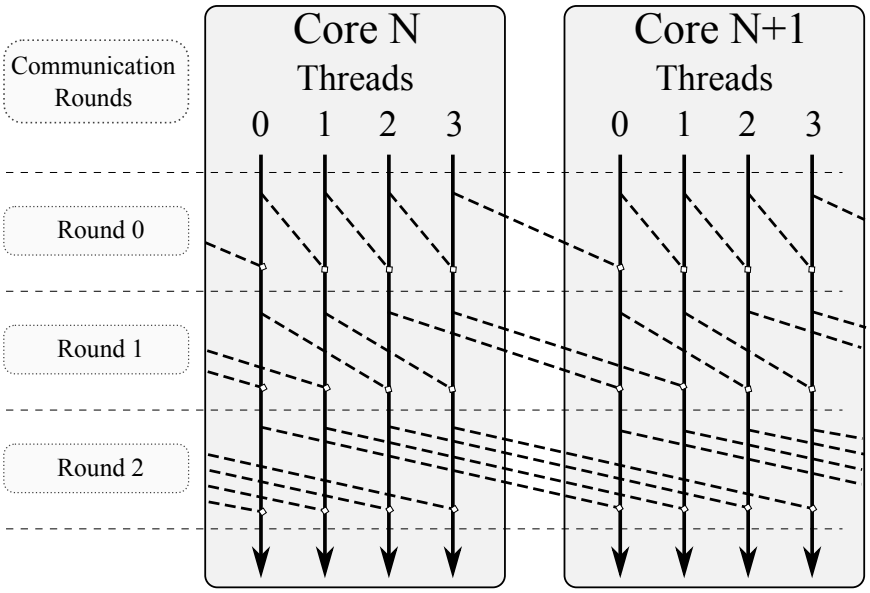
The design of the hybrid barrier originated from the observation that, on systems with a hierarchical topology, different algorithms can be optimal for different levels in the hierarchy. As always, the objective is to minimise inter-core communication while exploiting the low cost of intra-core communication. During the preliminary evaluation, it was observed that the most efficient algorithms were the dissemination barrier and the combining tree with arity 4. Figure 3.2b shows the data flow topology of a dissemination barrier. Evidently, each round contains at least one inter-core communication edge, which will be slower and will consequently determine the critical path for each round. Therefore, it is proposed to rely on the centralised sense-reversing barrier (equivalent to the combining tree with arity 4) for the intra-core phase, then revert to dissemination once a single thread remains per core. A similar approach was discussed by Cownie [Cow13]. Figure 3.2a and Listing 3.3 show the scheme and pseudocode of the hybrid algorithm.

When `ARCH_STORE_NR_NGO_REFINED` is defined and non-globally ordered stores are used, two memory fences are needed. The first memory fence is used to make the reinitialised counter visible to other threads before *Notification Phase* of the sense-reversing centralised barrier. The second memory fence is used to prevent stores in the next epoch of computation from being visible before the stores related to the synchronisation barrier, as described by Caballero [Cab] in Listing 6.8 and by Cownie [Cow13].

In respect to data flow topology, the hybrid barrier synchronisation algorithm presented in this section is schematically presented in Figure 3.3.



(a) Scheme of hybrid dissemination barrier



(b) Data flow topology of dissemination barrier

Figure 3.2: Hybrid dissemination barrier rationale.

```
void                        // hybrid barrier
hb_wait( hb_Bar_t * bar, // shared data
         tp_t * tp)      // thread-private data
{
  sr_Bar_t * srB = bar->srB[ tp->srBId ];
  int srBC = fetch_and_add( & (srB->count), -1);
  int tpsrBS = tp->srBSense;

  if ( srBC == 1 )
  { // last thread inside the core to arrive
    // call to dissemination barrier
    dsmn_wait( bar->dsmnB, tp);
    store( & (srB->count), srB->threadsNum);
#ifdef ARCH_STORE_NR_NGO_REFINED
    asm volatile( "lock; addl $0,(%rsp)\n");
#endif
    store( & (srB->sense), tpsrBS);
  } else
  { // non-last thread to arrive
    while ( tpsrBS != srB->sense )
      pause( );
  }
  store( & (tp->srBSense), ! tpsrBS);
#ifdef ARCH_STORE_NR_NGO_REFINED
  asm volatile( "lock; addl $0,(%rsp)\n");
#endif
}
```

Listing 3.3: Hybrid barrier wait method.

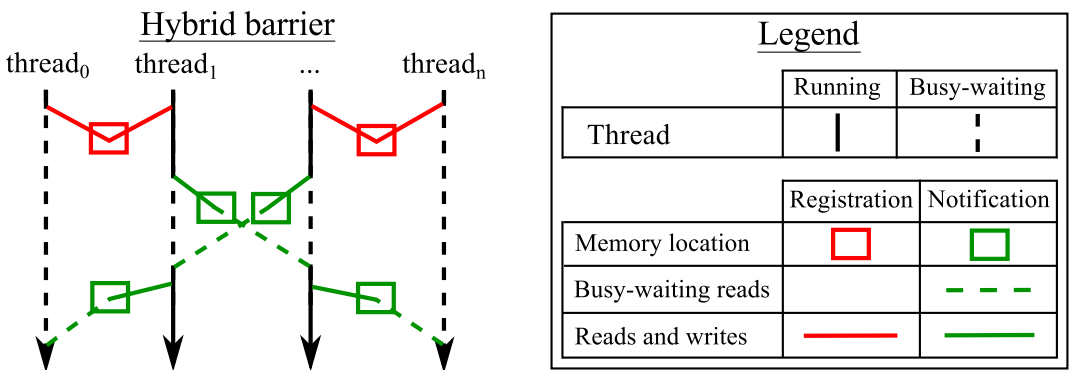


Figure 3.3: Diagram of the hybrid barrier synchronisation algorithm via shared memory.

3.4 Experimental Methodology and Results

3.4.1 Benchmarks

EPCC OpenMP Microbenchmarks

A part of the *Edinburgh Parallel Computing Centre* (`EPCC`) OpenMP microbenchmark [Bul99] was implemented which allows evaluating the overhead of the standalone barrier primitive, which will be referred to as `EPCC`. This benchmark was selected in order to achieve the highest possible frequency of barrier synchronisation.

NAS Parallel Benchmarks

The *Conjugate Gradient* (`CG`) and *MultiGrid* (`MG`) kernels were chosen from the C versions [SJL11] of the *National Aeronautics and Space Administration* (NASA) *Advanced Supercomputing* (NAS) *Parallel Benchmarks* (NASPB) [NAS] in order to evaluate the efficiency of barrier synchronisation. They represent widely-used algorithms for finding numerical solutions of mathematical problems and were used to evaluate the efficiency of barrier synchronisation in other works [CDM13, SK10, SPSS08]. As the original inputs for these benchmarks lead to a low frequency of barrier synchronisation, even using the smallest class S, which makes it difficult to observe barrier overhead, the input class Y for these 2 NASPB kernels was introduced. The frequencies of barrier synchronisation in both classes S and Y are presented in Table 3.1. Class Y consists of inputs { na=240; nonzer=2; niter=300; shiftY=5.0 } for `CG` and { problem_size=16; nit=800 } for `MG`. **`collapse`**`( 2)` clauses were also added to the relevant OpenMP parallel loops in the `MG` kernel, as suggested in [CDM13], to increase the amount of parallelism. The **`collapse`**`( <loop_nest_depth>)` clause is used to specify that a loop nest is not only parallel on the first loop construct annotated but also at a deeper level (parameter of the clause), which allows collapsing multiple loops into a single loop that is subsequently parallelised.

NAS Parallel Benchmark Kernel	CG		MG	
Input Class	S	Y	S	Y
Frequency, 10^3 barriers per second	6.4	21.6	8.7	12.4

Table 3.1: Barrier frequency in NASPB for inputs Y and S.

Direct N-body Simulation

A direct N-body simulation kernel was implemented to evaluate the efficiency of barrier synchronisation in the task where synchronisation cannot be relaxed. It was shown that using more relaxed synchronisation constructs, like *phasers*, is more efficient than using barriers for the `CG` and `MG` kernels [SPSS08]. In the implemented direct N-body simulation kernel, each thread calculates the coordinates and velocities of a single particle, so that the frequency of barriers will be the highest. Coordinates and velocities of a single particle are stored within a memory location which fit into a single cache line. Thus, in the implemented kernel, more parallelism cannot be extracted using other types of inter-thread synchronisation, as computation in each thread consumes data computed by all the other threads in the previous phase. This kernel will be referenced as `NBODY` in the rest of this chapter.

To test the specialised barrier implementations without interfering with the rest of the OpenMP implementation, each call to the **`__kmpc_barrier`**`( )` function, which is the internal barrier function in the Intel OpenMP library, was replaced with a trampoline, **`barrier_trampoline_`**`( )`, that calls the function implementing the desired barrier algorithm. The replacement is done at compile time, so it does not introduce extra overhead.

The correctness of the implemented barrier synchronisation algorithms was validated on the `CG` and `MG` kernels which contain the checkers which verify the obtained results. The `SANITY` kernel was implemented together with the checker of the obtained results to expand the validation test base. In this kernel, various patterns of inter-thread dependencies are exercised providing empirical evidence of the correctness of the implemented algorithms.

3.4.2 Naming Convention and Methodology

For the remainder of this chapter, the *geometric mean* overhead of a barrier (or execution time per barrier) is measured in the experiments on `EPCC` as the geometric mean of its overhead across the different thread counts; so $O_{geomean} = \sqrt[N]{\prod_{i=1}^{N} O_{n_i}}$, where N is the number of different thread counts, n_i is the number of threads in element i in the vector of different thread counts, and O_n is the barrier overhead for n participating threads. For `CG`, `MG`, and `NBODY`, execution time of a kernel is used instead of the overhead of a single barrier. On charts representing the geometric mean overhead (execution time), horizontal lines show the best (lower green line) and the worst (higher red

line) geometric mean overhead respectively, calculated as geometric mean of the best and the worst barrier overheads for each number of threads amongst all algorithms.

The **best geometric mean overhead** represents the *practical lower bound of synchronisation overhead*, which could be achieved given an oracle that predicts the best possible algorithm for a given number of threads. The performance of this **ideal meta-algorithm** shows the loss of performance resulting from using a single algorithm compared to the ideal performance that could (theoretically) be achieved by selecting algorithms dynamically with an oracle.

Individual barrier algorithms on the charts can be identified by their signatures. A signature uniquely identifies an algorithm by two 3-5 letter abbreviations and a number.

The first abbreviation corresponds to the algorithm and its variations:

- **sr** - centralised sense-reversing barrier;
- **dsmn** - dissemination barrier;
- **dsmnH** - hybrid dissemination barrier described in Section 3.3.3;
- **ct** - combining tree barrier;
- **stn** - static tournament barrier;
- **dtn** - dynamic tournament barrier;
- **ls** - tree-based notification with local flags;
- **gs** - broadcast notification using a single global flag;
- **omp** - Intel OpenMP barrier.

The second part of the signature defines the implementation of the `pause()` method:

- **pause** - the waiting loops contained the *64-cycle delay* (which will be discussed below);
- **spin** - the waiting loops contained no delay

This part is meaningless for the Intel OpenMP barrier.

The last part is a **number** corresponding to the arity of the tree for tree algorithms, the number of ways for a *n-way* dissemination barrier and meaningless for the centralised sense-reversing barrier and Intel OpenMP barrier.

The globally ordered version of the streaming stores was used (when `ARCH_STORE_NR` is defined) to identify the best performing barrier synchronisation algorithm on the selected benchmarks. The utilisation of non-globally ordered streaming stores was evaluated for the best performing algorithm (when `ARCH_STORE_NR` or `ARCH_STORE_NR_NGO` is defined).

All of the experiments rely on a balanced strategy for thread mapping, mapping threads to cores with the least load first. This type of thread affinity is enabled in Intel OpenMP by setting the environment variable `KMP_AFFINITY` to `balanced`. The `KMP_LIBRARY` and `KMP_BLOCKTIME` variables were set to `turnaround` and `infinite` respectively to prevent busy-waiting threads from relinquishing the CPU in favour of other scheduled runnable threads. The number of threads was controlled by setting the `OMP_NUM_THREADS` variable.

Each data point was obtained from 10 measurements and represented by a box plot. Unless otherwise indicated in a figure, the number of threads varies from 8 to 232 in increments of 8. The arities tested were: 2, 3, 4, 8, 16 and 32 for the combining tree barrier; 2, 3, 4 and 5 for the static tournament barrier; and 2, 3 and 4 for the dynamic tournament barrier.

When using Intel Xeon Phi at the highest available frequency of 1.04 GHz, the power management system (described in [Int14] Section 3) can change the voltage and frequency dynamically. This *dynamic voltage-frequency scaling* introduces considerable variability in measurements. In order to avoid this variability, the experiments were performed at 0.84 GHz. The direct way to set the frequency to this value via the Intel Xeon Phi software stack utilities was not found[1]. Changing the frequency to 0.84 GHz in an indirect way required performing the following steps:

1. Switching off all the power management configuration parameters:
   ```
   sudo micctrl --pm=set --cpufreq=off --corec6=off --pc3=off --pc6=off
   sudo micctrl --reboot
   ```

2. Running a computationally-intensive workload which can lead to heating of the coprocessor to the level at which the power management system reduces the frequency to 0.84 GHz: `micprun -k dgemm -d 0 -p "--i_num_rep 1000"` When the drop of frequency to 0.84 GHz happens, the workload should be killed. After this point, the frequency stays at 0.84 GHz.

[1]The author of the thesis asked this question at https://software.intel.com/en-us/forums/intel-many-integrated-core/topic/495831 but did not get a concrete answer.

MPSS Version	3.1.1
Flash Version	2.1.02.0390
SMC Firmware Version	1.16.5078
SMC Boot Loader Version	1.8.4326
uOS Version	2.6.38.8+mpss3.1.1

Table 3.2: Intel Xeon Phi software stack components versions.

The versions of components of the Intel Xeon Phi software stack used in the experiments are presented in Table 3.2.

3.4.3 Experimental Data and Discussion

Figure 3.4 shows the results for *geometric mean overhead* of barrier synchronisation algorithms on `EPCC`, and Figure 3.5 shows the results of the *ideal meta-algorithm* on `EPCC`.

Global Sense vs Local Sense

The overhead of tree barriers with global notification flag **gs** is much higher than that of barriers with a combining tree *Notification Phase* **ls** as can be seen on Figure 3.4, where the fastest **gs** variant is close to $2\times$ slower than the slowest **ls** variant.

It can be observed that a combining tree with a global flag in *Notification Phase* has higher geometric mean overhead than a centralised sense-reversing barrier, and the higher the tree arity, the less the overhead. This indicates that another hybrid algorithm can be investigated: a single global counter for the *Registration Phase*, like in the centralised sense-reversing barrier, and a tree-based *Notification Phase*.

Delayed Busy-Waiting

As it can be seen from Figure 3.4, where the geometric mean overhead for different algorithms follows an increasing order, the same algorithm with delayed busy-waiting **pause** outperforms the undelayed spinning variants **spin** in the majority of cases. Due to this fact, undelayed busy-waiting was not considered in the further experiments.

Sizing the Delay for Spinning

To determine a suitable value (number of sleep cycles) to pass as a parameter to the `delay` instruction conveniently provided on Xeon Phi, delay values in the range from 0

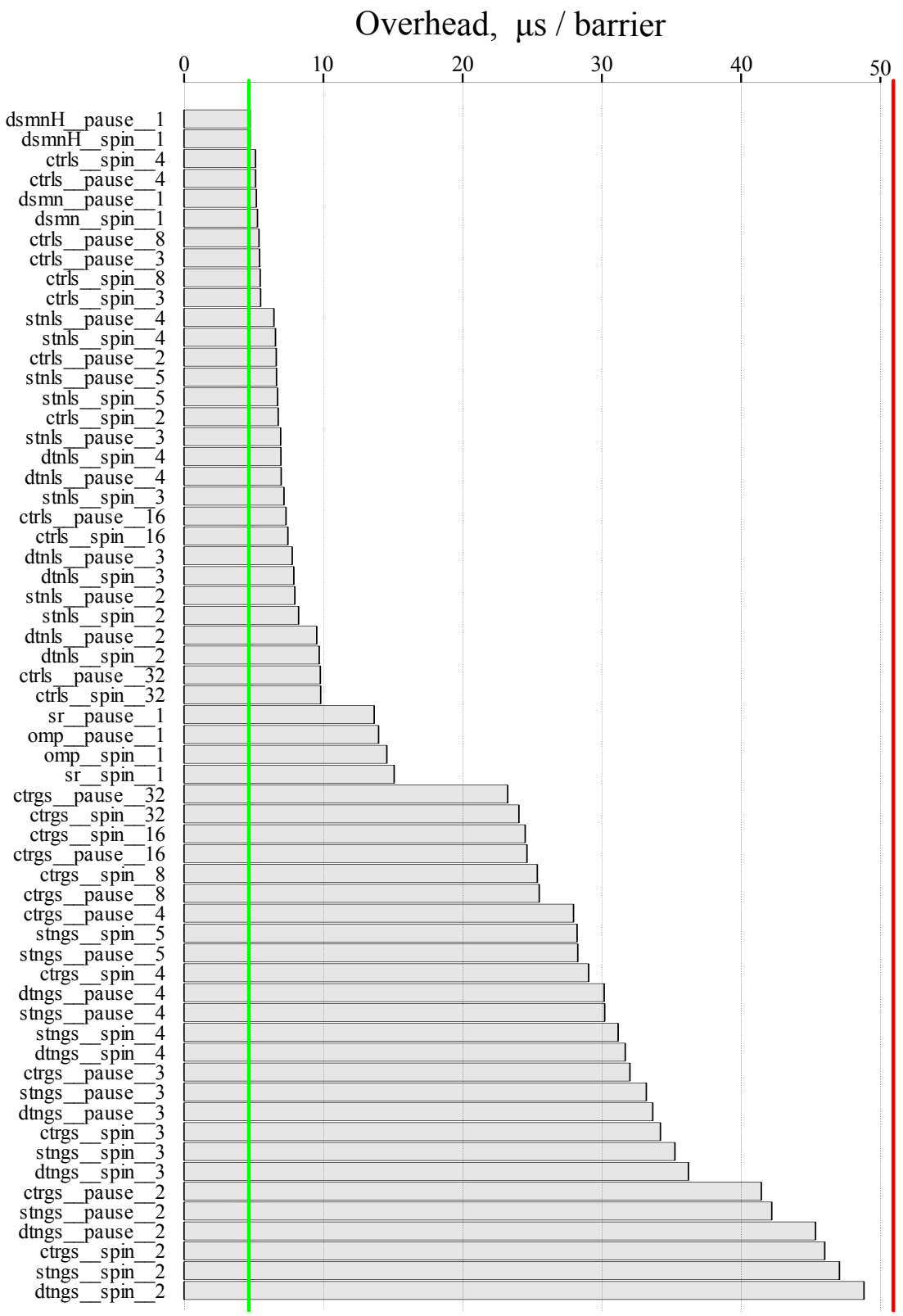


Figure 3.4: Geometric mean overhead of barrier synchronisation algorithms on EPCC (the green and red lines represent the selection of the best and the worst performing algorithm for a given number of threads respectively).

to 128 cycles with step 8 were evaluated on EPCC. Above 128 cycles, the performance starts to degrade as the delay introduces too much latency for waiting threads. The best performance was obtained with a 64 cycle delay that was used in all subsequent experiments. In future studies, this parameter can be investigated further as it is likely to depend on runtime conditions, such as the level of contention on the interconnect.

Hybrid Barrier

Figures 3.4 and 3.5 show that the hybrid dissemination barrier is the closest to the ideal meta-algorithm on synthetic benchmarks. Indeed, as confirmed by the results presented in Figure 3.6, the few instances where the hybrid barrier is not the most efficient in Figure 3.5 only correspond to minor timing variability up to 60 threads, up to which point the algorithms have similar overhead. Above 72 threads, the hybrid barrier overhead is considerably lower than that of the dissemination barrier.

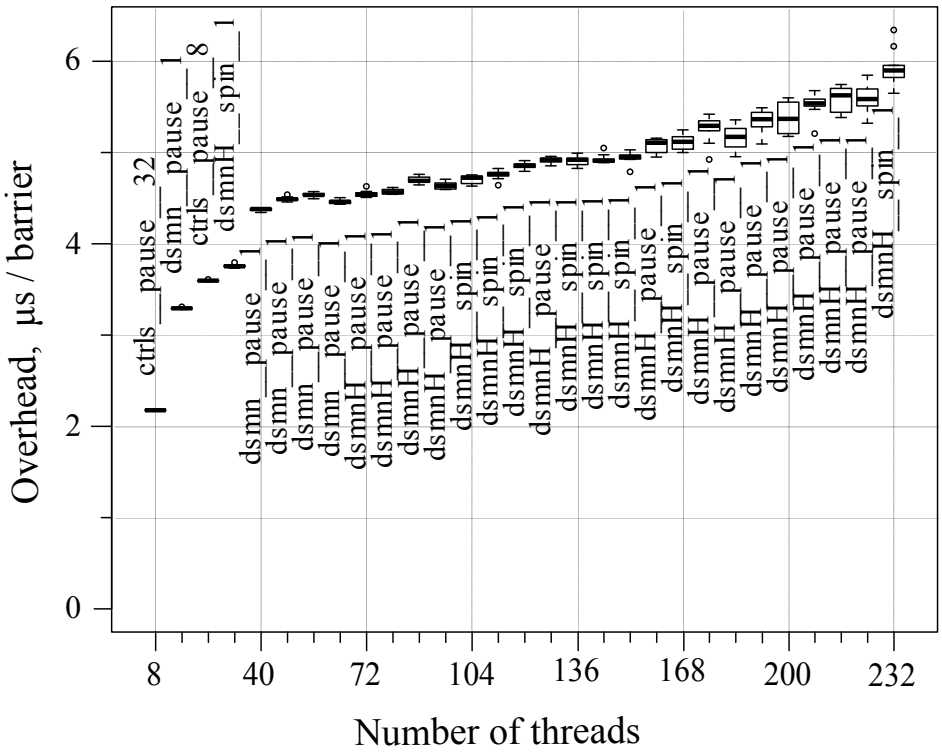


Figure 3.5: Overhead of the ideal barrier synchronisation meta-algorithm on EPCC.

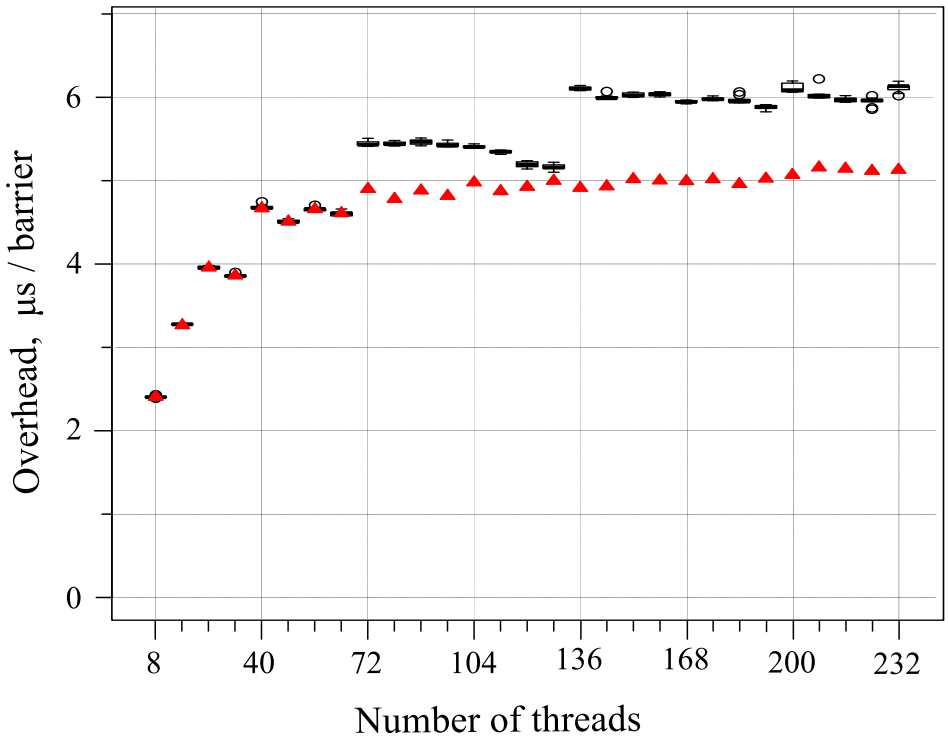
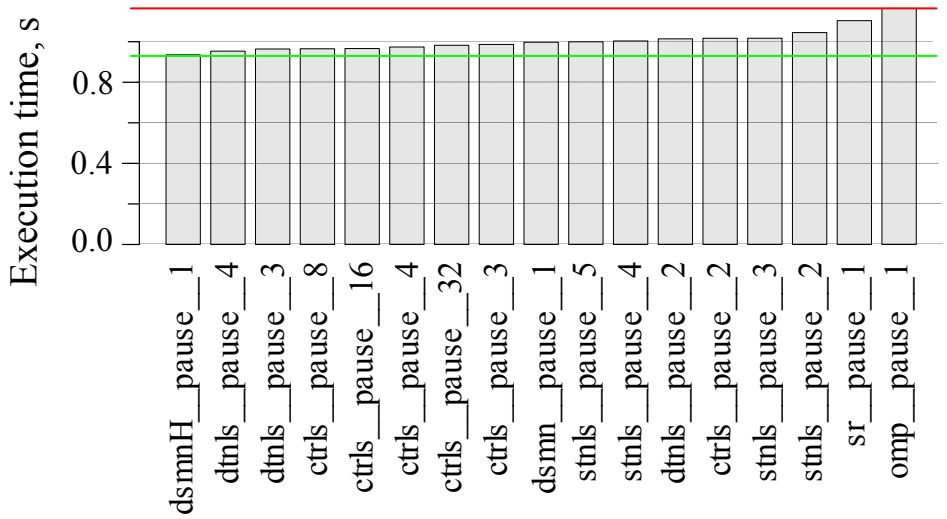


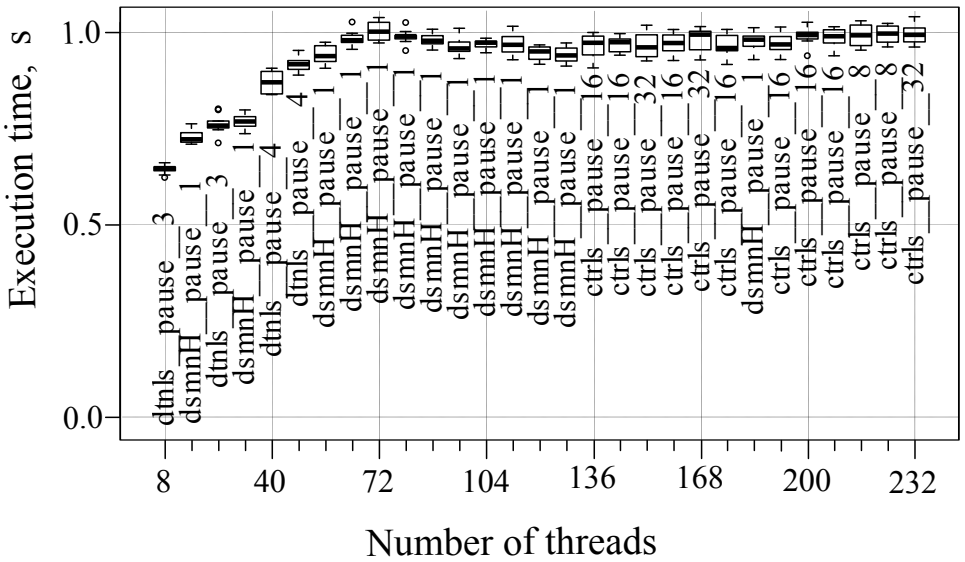
Figure 3.6: Dissemination barrier (black boxplots) compared to hybrid dissemination barrier (red triangles).

Real-world Kernels

The most efficient barrier algorithms selected above were evaluated on the `CG` and `MG` kernels of the NAS Parallel Benchmarks and direct N-body simulation kernel. The results are presented in Figures 3.7, 3.8 and 3.9, respectively. Hybrid dissemination barrier is superior over the other algorithms on `CG` and `NBODY`, while combining tree barrier with arities grater than 2 is slightly better than dissemination barrier on `MG`. The superiority of the combining tree barrier on `MG` may indicate high dispersion of thread arrival times at the barriers in this benchmark. Indeed, if at least one of the threads arrives at the barrier when all the other threads have already finished the *Registration Phase*, then the best performing barrier synchronisation algorithm will be determined by the time spent in the *Notification Phase*. For the dissemination barrier, the *Notification Phase*, in this case, is equivalent to the *Notification Phase* of the combining tree barrier with arity 2. Barriers with higher arities of the *Notification Phase* trees may perform better due to a shorter critical path for the cases with high dispersion of thread arrival times.

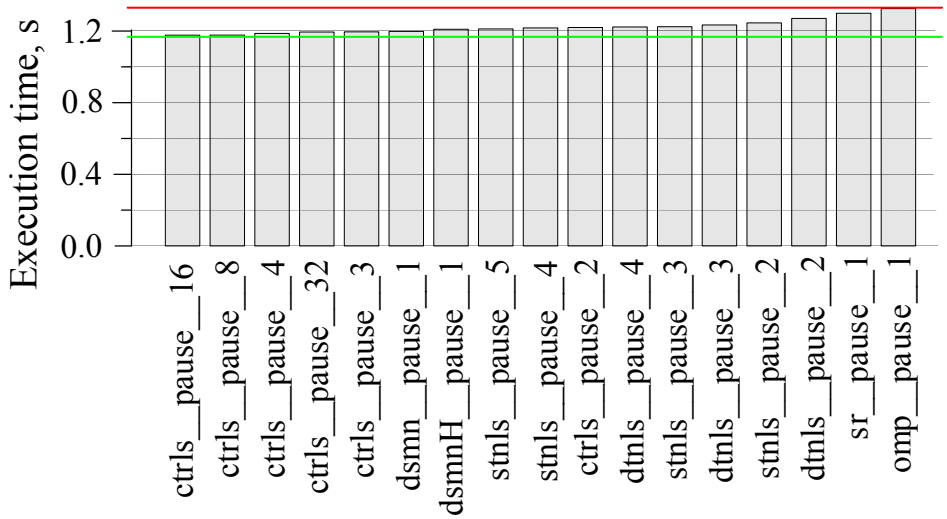


(a) Geometric mean on CG input class Y
(the green and red lines represent the selection of the best and the worst performing algorithm for a given number of threads respectively)

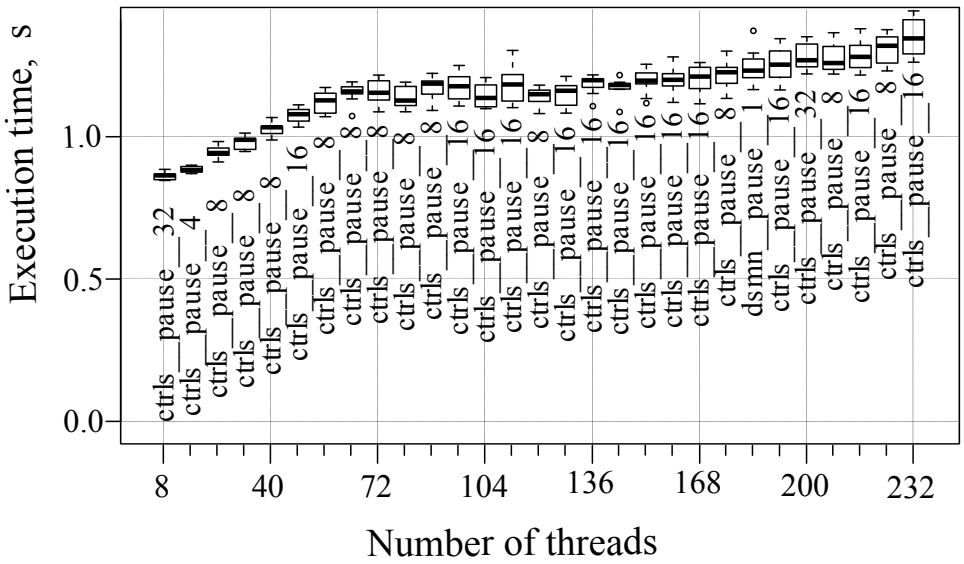


(b) Meta-algorithm on CG input class Y

Figure 3.7: Comparison of barrier synchronisation algorithms on the CG kernel of the NAS Parallel Benchmark.

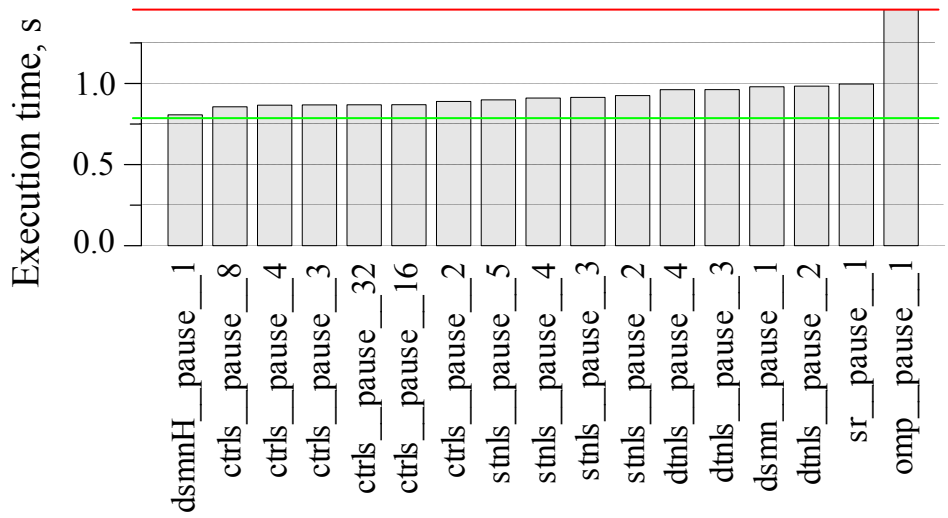


(a) Geometric mean on MG input class Y
(the green and red lines represent the selection of the best and the worst performing algorithm for a given number of threads respectively)

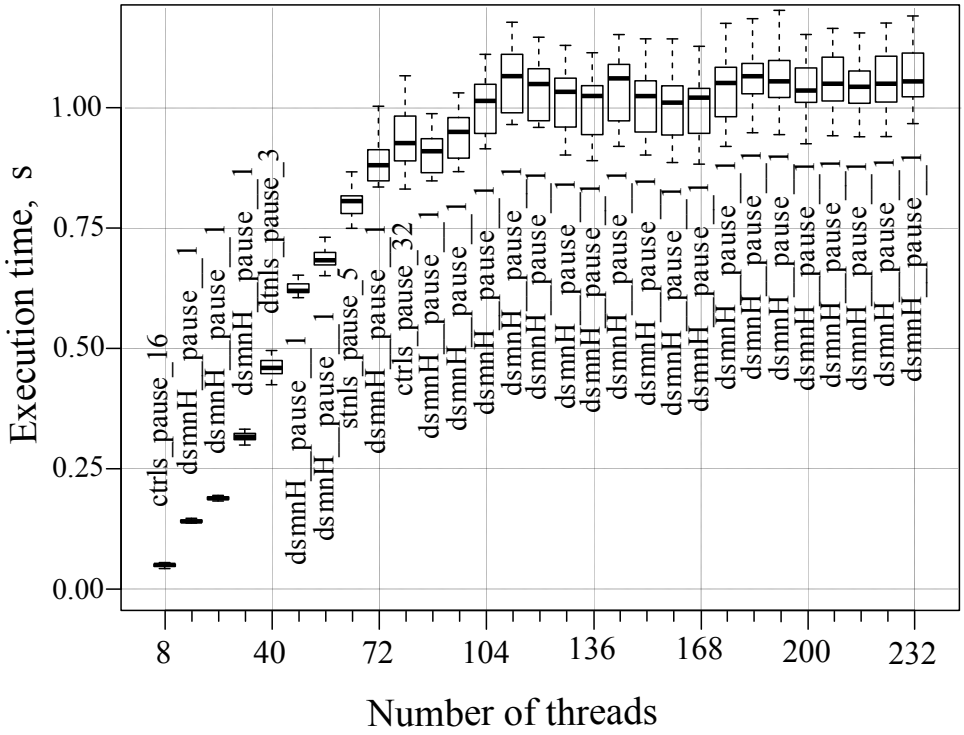


(b) Meta-algorithm on MG input class Y

Figure 3.8: Comparison of barrier synchronisation algorithms on the MG kernel of the NAS Parallel Benchmarks.



(a) Geometric mean on NBODY
(the green and red lines represent the selection of the best and the worst performing algorithm for a given number of threads respectively)



(b) Meta-algorithm on NBODY

Figure 3.9: Comparison of barrier synchronisation algorithms on the direct N-body simulation kernel.

Utilisation of Streaming Stores in Hybrid Barrier

Utilisation of globally ordered streaming stores (when `ARCH_STORE_NR` is defined) leads to 5% less overhead compared to ordinary stores for top performing barrier implementations on `EPCC`. However, utilisation of non-globally ordered streaming stores results in additional improvements.

A non-globally ordered streaming store is unordered in respect to other stores, meaning that other store instructions issued subsequently by the same thread can overtake it and become visible to other threads earlier. This relaxation of the memory ordering constraints makes it tempting to rely on this instruction for implementing barriers, as suggested by Cownie [Cow13] and shown by Caballero *et al.* [Cab] in Listing 6.8.

Indeed, 9.5% less overhead in geometric mean for the refined hybrid barrier algorithm implementation was observed on `EPCC` when using non-globally ordered streaming stores (when `ARCH_STORE_NR_NGO_REFINED` is defined) over hybrid barrier algorithm utilising globally ordered streaming stores (when `ARCH_STORE_NR` is defined) as shown in Figure 3.10. Utilisation of non-globally ordered streaming stores followed by memory fence operations (when `ARCH_STORE_NR_NGO` is defined) resulted in higher hybrid barrier synchronisation overhead over utilisation of globally ordered streaming stores.

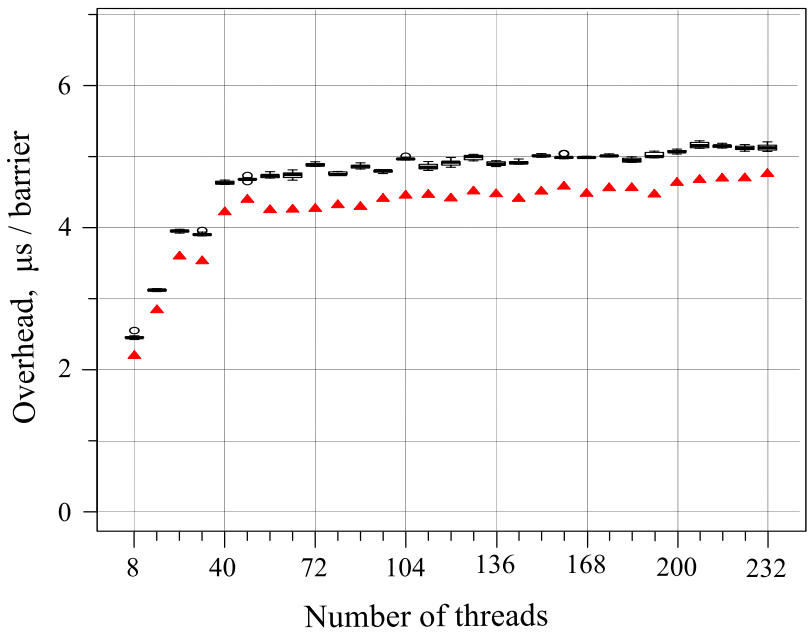
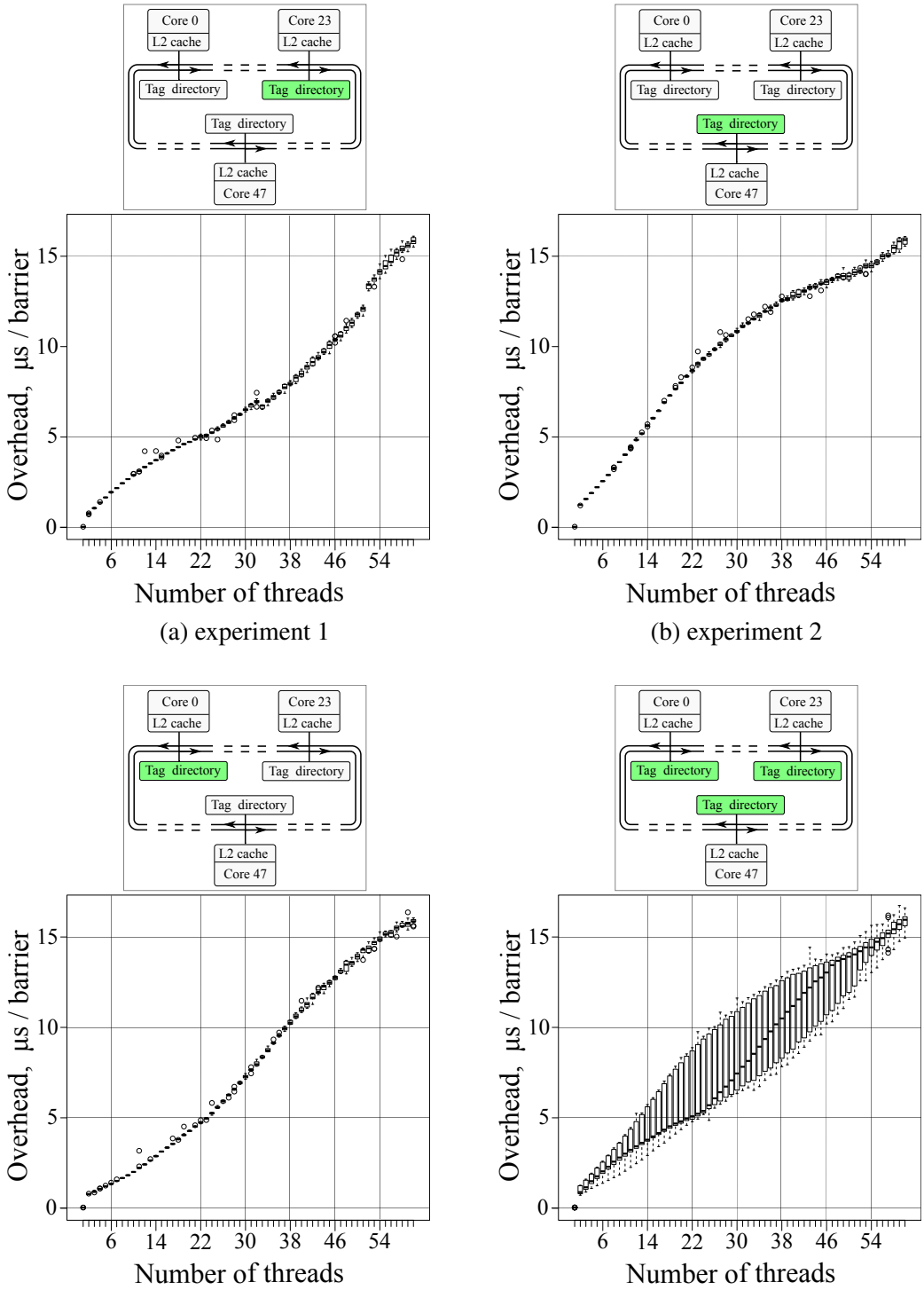


Figure 3.10: Hybrid dissemination barrier utilising globally ordered streaming stores (when `ARCH_STORE_NR` is defined) (black boxplots) compared to hybrid dissemination barrier refined to utilise non-globally ordered streaming stores (when `ARCH_STORE_NR_NGO_REFINED` is defined) (red triangles).

Effects of the Ring Interconnect

As discussed by Dolbeau [Dol14], the address of the shared memory location used for communicating among threads has a significant effect on latency, and therefore on barrier overhead. The same behaviour, induced by the ring interconnect and the distributed tag directories, was observed on the centralised sense-reversing barrier. Figures 3.11a, 3.11b, and 3.11c show three sets of performance results obtained on `EPCC` for 1 to 60 threads and the estimated positions of the tag directories (in green) responsible for the provision of cache coherence for the memory locations of the shared counters. Within each experiment, the data for a given number of threads was obtained in a single execution, containing multiple iterations where the same memory locations are re-used to store the shared synchronisation variables. The only difference between the three experiments is the physical memory addresses allocated and used for the synchronisation variables. It is apparent that the variability of performance results is negligible within a given set, but it is significant in-between sets. The graph in Figure 3.11d shows the joint results from the three experiments presented in Figures 3.11a, 3.11b, and 3.11c.

This variability is explained by the ring topology and the distributed tag directories. Indeed, in this configuration, each cache line is attributed a tag directory which is queried whenever a core misses in both L1 and L2 for that specific line, requiring a round-trip from the core to the related tag directory. This means that the delay of communications between threads during a barrier is dependent on the distances between threads and the tag directories that are responsible for the cache lines used in the synchronisation. As threads are mapped to cores in a circular order along the ring interconnect, the barrier overhead will follow an S-shaped curve as in Figures 3.11a, 3.11b, and 3.11c. Thus, the estimation of the location of tag directories responsible for the provision of cache coherence for the shared counter memory location is performed by finding the points on the S-shaped curves where the slope of the tangent line is at the minimum. If the selection of a tag directory tag directory is equiprobable, then (on average) the overhead of the barrier will increase linearly with the number of threads, following the average straight line apparent in Figure 3.11d. Unfortunately, the selection of tag directories is not under explicit user control, which introduces extra variability in the barrier overhead.



(a) experiment 1

(b) experiment 2

(c) experiment 3

(d) experiments 1-3

Figure 3.11: Impact of non-uniform access time to distributed tag directories for the centralised sense-reversing barrier.

3.5 Related Work

A barrier using SIMD instructions was proposed by Caballero *et al*. [CDM13], achieving a 2.84× lower barrier overhead on EPCC than the Intel OpenMP barrier. In respect to data flow topology, this SIMD barrier synchronisation algorithm is schematically presented in Figure 3.12. As shown in [CDM13] Figure 4, non-globally ordered streaming stores are followed by memory fences. As it was shown by the author of this thesis [RNPL15], such a replacement of a globally ordered streaming store with a non-globally ordered streaming store followed by a memory fence (when `ARCH_STORE_NR_NGO` is defined) leads to degradation. However, using fewer memory fences, as it was later shown by Caballero [Cab] in Listing 6.8, can lead to further performance improvements of synchronisation barriers. The refinement of the hybrid synchronisation barrier [RNPL15] by using fewer memory fences in a similar way (when `ARCH_STORE_NR_NGO_REFINED` is defined) leads to 9.5% less overhead.

Dolbeau [Dol14] showed that address selection is an important factor influencing barrier overhead due to non-uniform access time to distributed tag directories. Thus, a combining tree of specific topology and topology-aware memory allocation would allow lowering the overhead of barrier synchronisation. However, there is no explicit way to control topological aspects of memory allocation on Intel Xeon Phi systems. A direct comparison of this barrier against the Intel OpenMP barrier generated an initial speedup of 2.41×, further showing that address selection leads to an improvement to 2.85×. The technique employed to control memory allocation for this result is based on a trial-and-error approach for reverse-engineering the hashing function used by the tag directories.

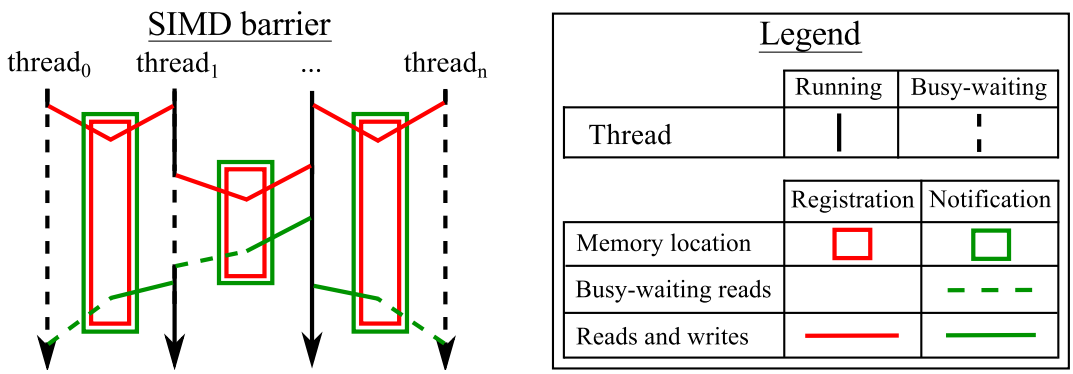


Figure 3.12: Diagram of the SIMD barrier synchronisation algorithm [CDM13] via shared memory.

Finally, Ramos and Hoefler [RH13] proposed a model for dissemination barrier synchronisation and also compare with the Intel OpenMP barrier. However, the experiments only showed equivalent performance with the Intel implementation.

3.6 Conclusions

The five state-of-the-art barrier synchronisation algorithms were specialised for the Intel Xeon Phi coprocessor. A novel hybrid specialised variant was presented based on different algorithms to synchronise at intra-core and inter-core levels. Comparing the hybrid algorithm with previous implementations, lower overheads have been observed in the experiments on EPCC barrier microbenchmark, and an improved performance has been observed on direct N-body simulation kernel and two NAS Parallel Benchmarks, CG and MG. In other words, the fastest known barrier implementation for Intel Xeon Phi was presented. These optimised barriers are available at `https://github.com/arodchen/cbarriers` released as free software.

In addition, the analysis of the impact of the ring interconnect and distributed tag directories of the Xeon Phi system on barrier synchronisation has been provided. The inability to have explicit control over tag directories leads to missed optimisation opportunities when specialising SW for the Intel Xeon Phi coprocessor.

Chapter 4

Theory and Practice of Managed Runtime Environments

As discussed in Chapter 1, a *Managed Runtime Environment* (MRE) represents an implementation of a high-level language *Virtual Machine* (VM) such as the *Common Language Infrastructure* (CLI) [Sta12] or Java VM [LYBB14]. This chapter describes the fundamentals of MRE internals. It discusses the open-source research JVM implementation called Maxine VM in detail and compares it with other research and industrial-strength VMs. On the basis of this comparison, Maxine VM was selected as a component of the MaxSim simulation platform (presented in Chapter 6), that is designed to achieve part of the research aims of this thesis. Part of the material in this chapter was published in the MULTIPROG 2016 workshop proceedings [KRB+16], the VEE 2017 conference proceedings [KCR+17], and the ISPASS 2017 conference proceedings [RKN+17].

4.1 Fundamentals of Managed Runtime Environments

A *Managed Runtime Environment* (MRE) is a software layer shown in Figure 4.1 that provides a *Virtual Machine* (VM) abstraction to applications via an interface that hides the implementation details of the underlying layers. State-of-the-art MREs are dynamic execution environments, which contain a *Just-In-Time* (JIT) compiler, a memory manager with *Garbage Collection* (GC), in addition to libraries and other components. The MRE libraries, such as the .NET Framework Class Library [.NE17] and the Java Class Library [JCL17], contain implementations of the barrier synchronisation primitive, so the results obtained in the previous chapter are directly applicable

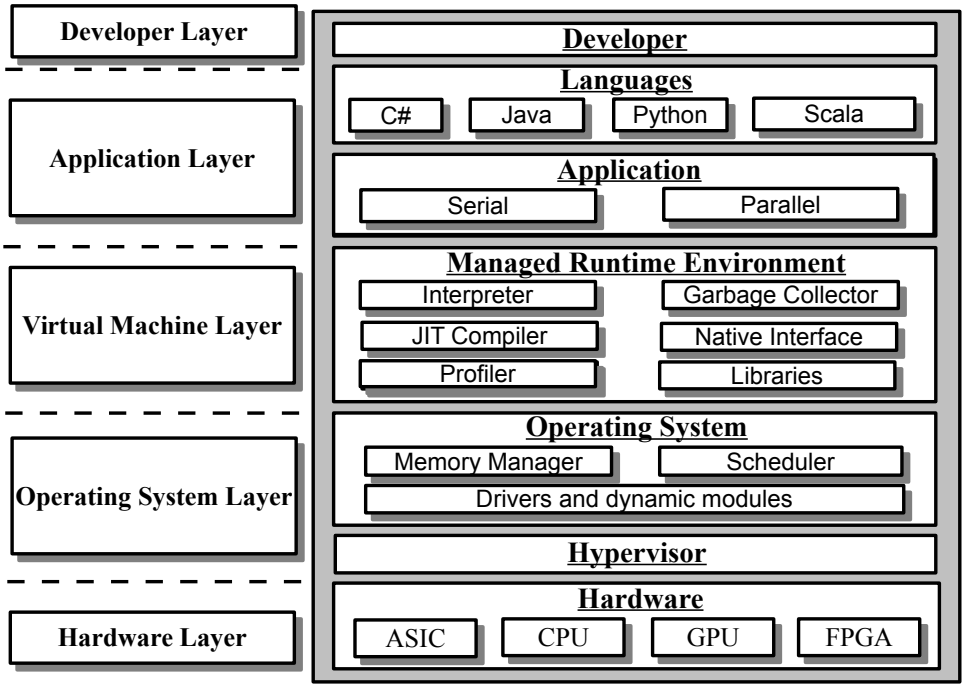


Figure 4.1: Managed runtime environment in the context of hardware, software and developer stack.

to MREs for the Intel Xeon Phi 5110P coprocessor. Unfortunately, Java was not supported for programming Intel Xeon Phi at the time the work in the previous chapter was done [Ree13]. The components, mentioned in the subsequent chapters of this thesis, are discussed in detail below.

4.1.1 VM Emulation Engine

As discussed in Chapter 1, managed code is distributed in a *bytecode* format targeting VM V-ISA [SN05]. Emulation of a VM operation during bytecode execution can be performed either by (1) *interpretation* or (2) by *compilation* to a target's ISA and subsequent execution by the underlying HW. Interpretation decodes and emulates the operation of single instructions one by one according to the control flow of the program. In case of compilation, a block of target code is generated for one or several bytecode instructions. The generated blocks of code can be linked together and optimised across basic block and function call boundaries in correspondence with control and data dependencies. Interpretation is performed by the module of an MRE called an *interpreter*, while runtime compilation, called *Just-In-Time* (JIT) compilation, is

performed by an MRE module called a *JIT compiler*. In contrast to unmanaged static compilation, JIT compilation can utilise dynamic profiling information for guiding optimisations. Dynamic profiling information is a set of counters associated with certain events happening during execution, such as control transfers or cache misses for example. Profiling information facilitates program adaptation to changes in the dynamic environment by re-optimising code.

Interpretation

Interpretation can be classified into two major groups by the pattern of operation [SN05]: *decode-and-dispatch interpretation* and *threaded interpretation*. Decode-and-dispatch interpretation happens in a loop, and the body of this loop performs decoding of the current VM instruction and calls a method responsible for the corresponding V-ISA operation emulation.

Threaded interpretation [Kli81] in contrast to decode-and-dispatch interpretation does not have the main loop. Instead, each method responsible for emulation of a certain V-ISA operation contains a finalising block of code performing decoding of the next VM instruction and jumping to an address of the method responsible for emulation of the subsequent operation. Addresses of emulation methods can be stored in a lookup table indexed by operation codes – *opcodes*. Alternatively, each method can be located at such an address that it can be calculated via pointer arithmetic avoiding a table lookup.

Profiling

Profiling is the process of collecting information about program execution. It can be used both for developer insight regarding application behaviour as well as for guiding compiler optimisations. Profilers can be classified into two major groups: *instrumentation-based* and *sampling-based*.

Instrumentation-based profiling is performed by insertion of a piece of code for incrementing a counter during execution of an associated point of interest. A basic example is a counter associated with the first instruction of a method. Such a counter provides information about the number of times each method was executed and allows the identification of frequently executed methods – *hot spots*. Instrumentation can be done at finer granularity allowing the identification of hot spots within each method. Instrumentation-based profiling allows precise information to be obtained about program execution at the cost of additional execution of code to increment counters. Thus,

this type of profiling can be performed during interpretation to identify (1) hot spots for optimising JIT compilation, and (2) code that has never been executed for which optimising compilation can be omitted.

During sampling-based profiling, an application of interest is interrupted periodically at random points of execution. Counters are associated with these random points and incremented if an interrupt happens again in the same place. Sampling-based profiling is less intrusive compared to instrumentation-based profiling as counters are associated with application interrupt points only. Thus, it enables probabilistic (imprecise) information to be obtained about hot spots.

JIT Compilation

JIT compilation is a process of target HW code generation for VM emulation at runtime. Generated target code is stored in a region of memory called the *code cache*. JIT compilers can differ in the level of applied optimisations, and in general, the higher the level of optimisations the greater time is required for compilation. Thus, there is a trade-off between performance of generated code and compilation time.

Non-optimising *baseline compilation*, as the name implies, is performed without any optimisations. Functionally, this kind of compilation represents emitting and linking methods responsible for emulation of V-ISA operations used for decode-and-dispatch interpretation. Thus, in contrast to interpretation, decoding of emulated VM operations is done only once – during compilation. Therefore, performance of VM code compiled by the baseline compiler is higher than the performance of interpretation of the same VM code given that the same methods are used for emulation of V-ISA operations.

High-performance VM emulation can be achieved via optimising compilation. In order to achieve peak performance, aggressive optimising JIT compilers can take advantage of dynamic profiling information to generate highly optimised code for hot paths of execution and totally ignoring cold paths that have not been executed before. If an ignored cold path is taken at runtime then a special action should be taken by an MRE which is called *deoptimisation* [HCU92]. During deoptimisation, execution is transferred from a triggering region of code to a corresponding place of unoptimised code generated by a baseline compiler or to an interpreter. Profiling information is updated, and optimised code with a speculative assumption which led to deoptimisation can be recompiled.

4.1.2 Garbage Collection

Automatic memory management is a set of techniques used in MREs to assist programmers with memory allocation and deallocation. The lifetime of a new object starts with an allocation of a block of memory necessary to accommodate data related to this object. New objects are allocated in the region of memory with a limited amount of space called a *heap*. The important feature of automatic memory management is that it frees programmers completely from manual memory deallocation at the end of the lifetime of an object. Instead, *Garbage Collection* (GC) identifies memory occupied by objects that are no longer referenced in the VM representation, so that this memory can be reallocated. Automatic GC is demand-driven and is usually triggered by memory starvation. There are a wide number of algorithms to perform GC, which can be classified into five major classes [JHM11]: *mark-sweep*, *mark-compact*, *copying*, *generational* and *reference counting*. These classes are described in the small survey below.

Mark-sweep GC traverses all reachable objects by references recursively starting from root objects and marks them during traversal. Root objects are a set of objects referenced from VM stacks, VM registers and static VM storage. The marked objects in the heap are considered to be live objects, and those that are unmarked are considered to be dead. On the sweep stage, the memory of the dead objects is reclaimed. Mark-sweep GC leads to free memory fragmentation which makes memory allocation more complex in comparison with allocation from a single unfragmented region of memory.

Mark-compact GC tackles the fragmentation problem of the previously described method by moving live objects to the top or the bottom of the heap after the mark phase, which leads to release of unfragmented unused space. After the compaction phase, references to compacted objects are updated with their new addresses. To optimise the number of references to be updated it is possible to reference an object indirectly through an associated pointer – *handle*. Utilisation of handles allows decreasing the number of reference updates at the expense of extra indirection for every object access.

Copying GC improves the mark-compact method by allowing concurrent operation of the mark and compact phases. The heap is split into two equal parts, so GC is initiated when only half of the heap is used. In this case, GC can compactly copy live objects from one half of the heap to the other whilst performing the mark phase. Copying GC allows merging mark and compact phases, but at the cost of effective utilisation of only half of the heap at any time.

Generational GC utilises object lifetime information to increase the probability of

checking liveness of objects that are more likely to be unreferenced and to reduce the overhead of copying objects with long lifetimes. Thus, the heap can be divided into several parts containing objects with certain lifetimes only. The lifetime can be defined empirically, by promoting an object from the younger generation to the older after it has survived a certain number of GC invocations.

Reference counting GC assigns each object a counter denoting the number of active references to it. Each time the reference to the object is created its counter is incremented, and when a reference is overwritten by another object the counter of the referenced object is decremented. Thus, when a reference counter reaches zero, the referenced object is considered to be dead.

4.2 Maxine VM

The Maxine VM [WHVDV+13] is the selected JVM to investigate opportunities for HW/SW co-design and co-specialisation of general-purpose CPUs and MREs in this thesis. The justification of this choice is presented in Section 4.2.4. Maxine VM is a meta-circular Java VM written in Java, so it manages not only application execution but its own execution as well. The primary design goals of Maxine VM are modularity and increased research productivity. Maxine VM consists of a number of interchangeable modules that are accessed through module interfaces, which are called *schemes*. The schemes describe heap and GC functionalities, multi-level JIT compilation policy, object layouts, and other aspects of JVM implementation details. In addition, it has a co-designed integrated debugging support (Maxine Inspector) that allows the binding of low-level execution entities (such as assembler instructions and memory addresses) with high-level VM information (such as methods and objects).

4.2.1 Baseline Compiler

Maxine VM utilises only JIT-compilation as the VM emulation engine. It has no interpreter and uses the baseline T1X compiler [WHVDV+13] to execute some VM code for the first time. T1X is a fast template-based compiler, and it emits, initialises, and links a piece of pre-compiled code (a template) for emulation of every VM instruction. Thus, T1X compilation generates code very quickly at the expense of lost optimisation opportunities. The T1X compiler associates a counter with every compiled method, which is incremented at the first instruction and inside every loop of a method. When

a counter reaches some threshold (5000 by default), an associated method is compiled by an optimising compiler, and all calls to the baseline method are replaced by calls to the optimised version.

4.2.2 Optimising Compilers

For high-performance VM emulation, the Maxine VM employs two optimising compilers: C1X and Graal. C1X is the default optimising compiler in Maxine VM. It originated from the Hotspot client compiler C1 [KWM+08], and it has similar architecture and optimisation phases.

Graal [Gra16] is the most aggressive optimising compiler available in the Maxine VM. In contrast to C1X, it relies heavily on profiling information to guide speculative optimisations that yield better performance when speculative assumptions are correct. If speculative assumptions are incorrect, then a deoptimisation mechanism diverts control flow to the corresponding pieces of code generated by the T1X compiler. In addition, Graal compiler benefits from the integrated Truffle [WW12] self-optimising *Abstract Syntax Tree* (AST) interpreter that enables JavaScript, R, and Ruby applications to run on top of the Maxine VM.

4.2.3 Heap Allocation and Garbage Collection

Maxine VM implements a heap with two equal semi-spaces, and allocation of objects happens in only one of the two semi-spaces at the same time. When GC is triggered by memory starvation, all live objects are copied from one semi-space to the other, and allocation continues in the other semi-space. The heap is dynamically split into regions assigned to specific threads, and each thread performs allocation in its own *Thread Local Allocation Buffer* (TLAB) in the general case. TLAB allocation happens by returning and post-incrementing a pointer to free memory space inside a TLAB by the size of a recently allocated object. Memory reclamation is done by a stop-the-world flat semi-space collector based on Cheney's breadth-first copying GC [Che70].

4.2.4 Comparison With Other JVM Implementations

As discussed earlier, MREs are complex SW systems typically consisting of a baseline compiler and/or an interpreter coupled with an optimising compiler, GC algorithms, facilities for deoptimisation, and many other functionalities. Ideally, a VM should be

Research VM	ISAs	Class Libraries	Support of Other Languages
Jikes RVM	PowerPC, IA-32	Apache Harmony GNU Classpath	-
Maxine VM	x86-64, ARMv7	JDK 7	+ (via Graal and Truffle)

Table 4.1: Research VMs comparison.

designed in such a way to allow the plug-in of different modules that can extend optimisation capabilities. Unfortunately, this is not always feasible since high performance and high degrees of modularity are two aspects that counteract each other. In order to achieve high performance, VMs are optimised across the components sacrificing modularity.

Consequently, VMs broadly fall into two categories: production-quality and research VMs. Production quality VMs such as the HotSpot JVM [PVC01] can achieve high performance at the expense of limited experimentation capabilities due to the lack of modularity. On the contrary, research VMs such as the Jikes RVM [AAC+99] and Maxine VM [WHVDV+13] offer high degrees of implementation freedom and research productivity due to their modular design at the cost of lower (worse) performance in comparison with HotSpot VM. The comparison of Jikes RVM to Maxine VM is presented in Table 4.1, and Maxine VM has the following advantages over Jikes RVM:

1. It supports the widely-adopted x86-64 architecture.

2. It is compatible with the JDK7 Class Libraries and can run the full set of the DaCapo-9.12-bach [BGH+06], SPECjvm2008 [SPE08], pjbb2005 [PJB05], and other benchmarks.

3. It supports the Graal [Gra16] optimising compiler, which is the next-generation optimising compiler of HotSpot JVM.

4. It supports the Truffle [WW12] optimising AST interpreter, that allows the execution of other languages, apart from Java, such as JavaScript, R, Ruby, and others.

In order to assess the performance of Maxine VM, it is compared against the production-quality HotSpot VM. The performance is measured as the reciprocal of the execution time of a given benchmark. To that end, the recent version of Maxine VM[1] (rev. 8810) with its two optimising compilers, C1X and Graal customised for Maxine[2] (rev.11558), is compared against the production-quality HotSpot VM with its two optimising compilers, C2 (ver. 1.8.0.25) and Graal[3] (rev. 21075). The earlier version of Maxine VM[4] (rev. 8750) and Graal[5] customised for Maxine (rev. 11539) is added for comparison to show the contribution of the author of this thesis to the Maxine VM research project during his PhD.

The DaCapo [BGH+06] benchmarks are used for performance comparison. The DaCapo benchmarks are a set of 14 representative open-source Java applications which are: `avrora`, `batik`, `eclipse`, `fop`, `h2`, `jython`, `luindex`, `lusearch`, `pmd`, `sunflow`, `tomcat`, `tradebeans`, `tradesoap`, and `xalan`. They cover a wide range of domains from simulation of the microcontrollers to transformation of XML documents into HTML. These benchmarks are widely used for performance analysis of JVM implementations and have been utilised for performance evaluation in this and subsequent chapters.

The performance comparison of the five `VM-compiler-version` triplets on the DaCapo-9.12-bach benchmarks[6] is presented in Figure 4.2, where performance is relative to `HotSpot-C2-1.8.0.25`. Each data point was obtained from 16 measurements. Whiskers represent 95% confidence intervals. As depicted in Figure 4.2, the performance of `HotSpot-Graal-21075` is comparable to `HotSpot-C2-1.8.0.25`, while the performance of `Maxine-Graal-8810.11558` and `Maxine-C1X-8810.11558` is 57% and 53% of `HotSpot-C2-1.8.0.25`, which is considered to be satisfactory for research purposes [WHVDV+13]. As already mentioned, the Maxine VM has two optimising compilers, namely C1X and Graal. Theoretically, if the Maxine VM is optimised across its modules, its peak performance with Graal should be on-par with that of the HotSpot VM with the same compiler. From the performance results presented in Figure 4.2 it can be seen that `Maxine-Graal-8810.11558` is around 8% faster than `Maxine-C1X-8810.11558` in geometric mean. However, since C1X is much less complex than Graal and has much lower compilation times, C1X has been selected as the

[1] `https://github.com/beehive-lab/Maxine-VM`
[2] `https://github.com/beehive-lab/Maxine-Graal`
[3] `http://hg.openjdk.java.net/graal/graal-compiler`
[4] See footnote 1.
[5] See footnote 2.
[6] `eclipse` is not present, as it did not pass on `Maxine-Graal-8810.11558`

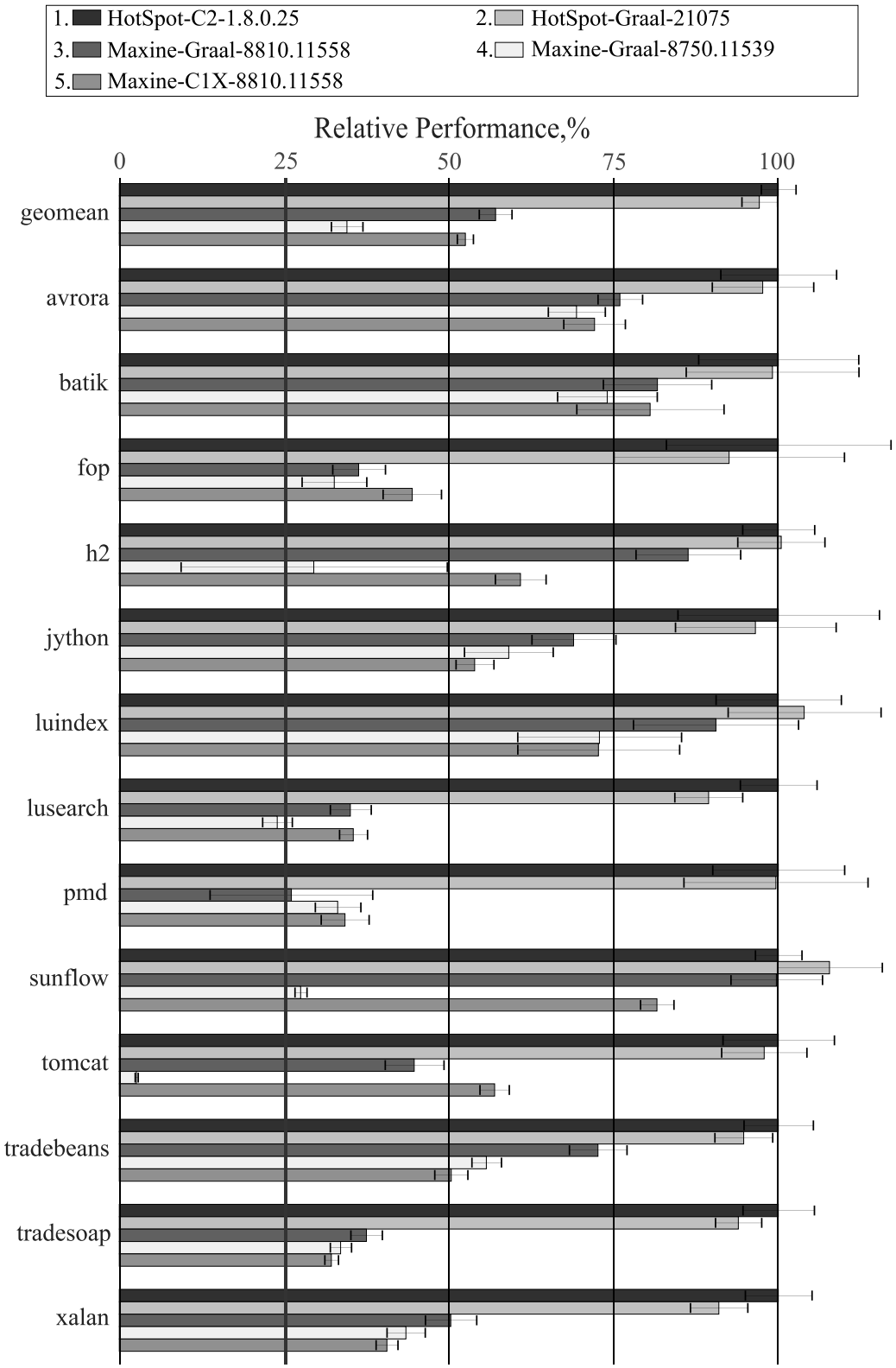


Figure 4.2: Performance of different `VM-compiler-version` triplets relative to `HotSpot-C2-1.8.0.25` (higher is better).

optimising compiler for the purposes of this thesis.

The following major changes have been contributed to the Maxine VM by the author of this thesis during his PhD:

1. Profiling instrumentation in the T1X baseline compiler was implemented.
2. Profile-guided optimisations were enabled in the Graal optimising compiler.
3. Critical math substitutions were enabled.

This work on improving utilisation of the Graal optimising compiler by the Maxine VM resulted in 1.64× speedup as can be seen in Figure 4.2 by comparing performance of `Maxine-Graal-8750.11539` (before the changes) and `Maxine-Graal-8810.11558` (after the changes).

4.3 Summary

In this chapter, the fundamentals of MRE internals were presented, and different implementation options of VM emulation engine and automatic memory manager were reviewed.

In addition, this chapter discussed the open-source research JVM called Maxine in detail and compared it with other production-quality and research JVM implementations. The observed performance difference is less than 2x on the DaCapo-9.12-bach benchmarks against the state-of-the-art HotSpot VM, which is considered satisfactory for research purposes. The benefits of using Maxine VM are its modularity and its powerful co-designed integrated debugging support (Maxine Inspector) leading to research productivity.

Thus, this chapter provides background knowledge for the material of Chapters 6 and 7, which describes a novel simulation platform based on Maxine VM and its applications.

Chapter 5

Theory and Practice of Computer Architecture Simulation

As it was discussed in Chapter 1, evaluation of a SW prototype on a HW model is a key step in the iterative HW/SW co-design process. This chapter presents the fundamentals of HW model evaluation by computer architecture simulation. It discusses the open-source research simulator called ZSim in detail and compares it with other simulators. On the basis of this comparison and validation, ZSim is selected to simulate managed workloads executed on top of Maxine VM, which was described in the previous chapter. Part of the material in this chapter was published in the ISPASS 2017 conference proceedings [RKN+17].

5.1 Fundamentals of Computer Architecture Simulation

5.1.1 Comparison of Simulation with Analytical Modelling

Evaluation of SW executed on a HW model enables quantification of the execution by providing runtime characteristics such as execution time, power consumption, and memory access behaviour. There are two major classes of evaluation methods of SW execution on HW models. *Simulation* is the most straightforward and wide-spread evaluation method, while the other major option is *analytical modelling* [Eec10]. The former method utilises the functional and physical models of HW and simulates SW execution instruction by instruction, while the latter method is based on building a high-level analytical model of HW described by a mathematical formula taking some

characteristics of SW execution as an input. Thus, simulation is usually a more precise method, while analytical modelling is a faster method [Eec10]. These properties define the rationale for applicability of these methods: while analytical models provide a fast first-order approximation and narrowing down of the subspace of interest, simulation may be used for increasing the accuracy of quantification in that subspace.

As it was mentioned in Chapter 1, one of the aims of this work is specialisation of general-purpose CPUs via novel microarchitectural extensions and their utilisation in MREs. Although analytical modelling can be considered for evaluation of HW extensions, another important requirement of the research described in the subsequent chapters is verification of functional correctness of the proposed extensions. Such a verification can be performed via modelling of program execution at the ISA level without evaluation of any other characteristics, such as execution time, consumed power end energy. Such ISA-level modelling is called *functional simulation*. Thus, simulation in contrast to analytical modelling can meet both requirements simultaneously which are functional verification and power/performance evaluation.

5.1.2 Overview of Simulation Techniques

Microarchitectural simulation presents a number of challenges that define trade-offs between the following four characteristics [Eec10]: (1) simulation speed, (2) simulation accuracy, (3) design complexity affecting engineering efforts required to modify or implement new HW models, and (4) simulation coverage, which is a consideration to simulate only selected parts of a HW model. While engineering a simulator, an implementor has to make a number of design decisions which will be described below.

Methods of Operation

Simulators can be classified into two groups by the method of operation: *execution-driven* and *trace-driven*. Execution-driven simulators, as the name implies, (1) execute programs of interest via functional simulation and (2) feed calculated data to timing and other models. Thus, this method of operation contains two stages. Data fed to models can be stored in trace files, and trace-driven simulators use such trace files directly as input. The advantages of trace-driven simulation in comparison with execution-driven simulation are determinism, since the same trace file is used, and

faster operation due to the absence of the functional simulation stage. However, execution-driven simulation is a more flexible method of operation as execution can be dependent on timing and other models, and a trace-driven simulator is not able to account for such dependencies and models a single scenario of execution stored in a trace file. The two examples of timing-dependent instruction execution are (1) control- and data-speculative execution of a single-threaded code and (2) dependent execution of multiple threads. As dependent multi-threaded execution is typical for MREs, execution-driven simulation is a preferable option for the studies presented in the next two chapters.

FPGA-Accelerated and Non-Accelerated Simulation

Simulators can also be classified by the type of integrated circuits that are used to run the simulation. The two major classes of integrated circuits used for simulation are FPGAs and CPUs.

An *FPGA-accelerated* (or *FPGA-based*) simulator is an FPGA circuit configured by loading synthesised HW description of a HW model. FPGA-based simulators can achieve higher simulation speed in comparison to non-accelerated simulators, but their implementation or extension requires substantial engineering efforts due to the greater complexity of programming FPGAs compared to CPUs. Due to this reason, FPGA-based simulators are not considered for the studies presented in the next two chapters. The example of FPGA-based research simulators, infrastructures and methodologies are HAsim [PAK+11], Arete [KVBWA12], FAST [CSK+07], ProtoFlex [CPN+09], and MAST [MPG+17].

Non-accelerated simulators are programs targeting general-purpose CPUs, and they are easier to maintain than FPGA-based simulators. Such simulators can be subdivided into two groups by the type of execution engine: *emulation* and *binary translation*. In the case of emulation, each instruction, one by one, is decoded and modelled, while in the case of binary translation, after the instruction decoding step, a simulator generates a block of code which performs modelling. These blocks of code can be linked together and optimised across their boundaries for adjacently simulated instructions [Haz11]. Thus, during the subsequent modelling of the same instructions, the decoding step can be omitted. Therefore, an execution engine based on binary translation can be faster in case the same code is simulated frequently. An execution engine based on binary translation can be particularly efficient for functional simulation in case the simulated ISA is equivalent to or kindred to the ISA of the CPU used

for simulation [LCM+05, GdL16].

Simulation Parallelisation

Moving in a many-core direction raises a problem of parallel simulation of multi-threaded code on many-core processors. Even when two independent applications are executed on different cores, they can interfere with each other due to sharing resources of the processors, for instance, due to shared last level cache [CMB+13]. Thus, accurate simulation of multiple threads running on several cores can require accounting for inter-core dependencies. In fact, these dependencies can be pretty fine-grained. Preserving all such dependencies can cause significant synchronisation delays outweighing all the benefits of simulation parallelisation. To get speedups, some parallel simulators, such as Graphite [MKK+10] and ZSim [SK13], allow violation of precise dependencies while simulating cores in parallel for a configurable number of cycles. After parallel simulation of cores with relaxed inter-core dependencies, contentions for shared resources are modelled using queuing theory models in Graphite [MKK+10] and using the bound-weave algorithm in ZSim [SK13].

Full-System and User-Level Simulation

Simulators can be subdivided into two groups: *full-system* and *user-level*. Full-system simulators model the whole computer system including input/output devices and operating system code. On the contrary, user-level simulators, as the name implies, model only user-level code omitting invocations of operating system code and input-output activities at all or partially.

The state-of-the-art full-system simulators are more complex and, typically, slower (higher simulation times) than user-level ones. The benefit of using a full-system simulation is the extra accuracy achieved since more components of the computing stack are simulated. However, for workloads[1] that spend the vast majority of their time in user-level code, this is not the case. The two primary examples of the open-source full-system simulators are gem5 [BBB+11] and MARSS [PACG11].

The user-level non-accelerated simulators, sacrificing the ability to simulate the kernel code, provide the best research trade-offs in terms of accuracy, simulation speed, and engineering effort for the purposes of the studies presented in this thesis. The examples of research simulators of this type are Sniper [CHE11, CHE+14],

[1] As it will be shown in Section 5.3.2, the DaCapo benchmarks with the exception of `avrora` are such an example.

ZSim [SK13], and Graphite [MKK+10]. From the currently available open-source user-level simulators, only ZSim allows the execution of arbitrary managed workloads via lightweight user-level virtualisation.

5.2 Power and Energy Consumption Modelling Using McPAT

As mentioned in Chapter 1, power dissipation is one of the major design limitations for future generations of many-core systems. Thus, power dissipation is an important characteristic while evaluating new HW models. The dissipated energy is a dependent characteristic, and it is equal to the integral of consumed power over time. Total consumed power can be represented as a sum of *dynamic* and *static* power. Dynamic power consumption is caused by switching circuit states. Static power is consumed due to leakage current, and it is dissipated even when a circuit is idle.

McPAT [LAS+13] is a parameterised framework for modelling power, area, and timing of multi-core and many-core processors. It requires (1) specification of microarchitectural parameters of a modelled processor and (2) microarchitectural events of a modelled program execution as input. Both inputs are passed in the XML-based format, so that microarchitectural simulators can collect required microarchitectural events and pass them to McPAT for power modelling. Using the provided events, McPAT can evaluate dynamic and static power consumption for each component of a modelled microarchitecture. The examples of the microarchitectural simulators coupled with McPAT are Sniper [HSC+12], gem5 [BBB+11], and Graphite [MKK+10].

5.3 ZSim Simulator

ZSim [SK13] is the simulator of choice to research opportunities for HW/SW co-design and co-specialisation of general-purpose CPUs and MREs. ZSim is an execution-driven (instruction-driven) simulator with the binary translation execution engine based on the Pin [LCM+05] dynamic binary instrumentation and modification tool. This optimised execution engine allows achieving simulation speeds of 20 *Mega-Instruction Per Second* (MIPS) for *Out-Of-Order* (OOO) core models on conventional CPUs. The most advanced OOO core that it can model is Intel Nehalem. One of the design goals of this simulator is scalability, which is achieved via the “bound-weave" simulation

Name		1C	2C	4C	1CQ
Cores	type	x86-64 Nehalem OOO core at 2.66 GHz			
	total	4			1
	enabled	**1**	**2**	**4**	**1**
Prefetchers		disabled			
L1I caches		32KB, 4-way, LRU, 3-cycle latency			
L1D caches		32KB, 8-way, LRU, 4-cycle latency			
L2 caches		256KB, 8-way, LRU, 6-cycle latency			
L3 cache	type	16-way, hashed, 30-cycle latency			
	size	8MB			2MB
Memory controller		1, 3 DDR3 channels, 47-cycle latency			
DRAM		3GB, DDR3-1066, 1GB DIMM per channel			

Table 5.1: ZSim configurations.

parallelisation technique. Another feature, required for the research purposes of this thesis, is the ability to model execution of managed workloads fast and accurately.

5.3.1 Comparison With Other Research Simulators

A detailed comparison of ZSim with other non-accelerated simulators mentioned in this chapter is presented in [SK13]. The other simulators capable of running managed applications are gem5, MARSS, and Sniper. Sniper can simulate managed applications executed on top of Jikes RVM only [Jik14, SHB+14]. The validation and comparative analysis [AS16] of gem5, MARSS, Sniper, and ZSim on MiBench [GRE+01] and SPEC CPU 2006 [SPE06] benchmarks showed that Sniper and ZSim have the two least and similar simulation errors on configurations with two and four cores. The study also revealed that ZSim is the fastest simulator among the four compared simulators. These facts justified the selection of ZSim for the research purposes of this thesis.

5.3.2 Validation of Simulating Maxine VM Running the DaCapo Benchmarks

With minor modifications to its user-level virtualisation and scheduling techniques[2], ZSim was able to simulate the full set of the DaCapo [BGH+06] benchmarks, discussed in Chapter 4, executed by the Maxine VM with the C1X optimising compiler. The parameters of the simulated systems used for validation are described in Table 5.1.

[2] `https://github.com/arodchen/zsim` rev.102

The configurations `1C`, `2C`, and `4C` represent the Intel Nehalem microarchitecture with 1, 2, and 4 enabled cores respectively. Furthermore, `1CQ` represents the 1-core CPU with just a **Q**uarter of the 8MB *Last Level Cache* (LLC). This configuration is used in the next chapter in order to simulate the case when only a quarter of the available `4C` resources is available to the workload (if the LLC could be partitioned, which is not possible on the Intel Nehalem microarchitecture).

ZSim was validated against a real system with the results presented in Figure 5.1. The performance of the simulated models `1C-ZSim`, `2C-ZSim`, `4C-ZSim` is validated against the performance of the real systems `1C-Real`, `2C-Real`, `4C-Real` respectively, where `Real` represents an Intel Core i7 920 (Bloomfield) CPU based on the Nehalem microarchitecture. The performance shown is relative to the `4C-Real` configuration. The performance is measured as the reciprocal of the execution time of a given benchmark. Whiskers represent 95% confidence intervals. It can be seen that the difference in geomean execution times between the real platform and the simulated models is from 8% to 12%, which is in alignment with the ZSim original validation [SK13]. Furthermore, the performance scalability pattern (from 1 core to 4 cores) of the simulated models is consistent with the real system. However, two major inconsistencies were observed. Firstly, the execution times of `eclipse` and `tradesoap` on the one-core model `1C-ZSim` were more than two times greater than the real system `1C-Real`. This is due to the different thread scheduling algorithms used: on the real system a *Completely Fair Scheduling* (CFS) [CFS14] scheme is employed while on ZSim a simple round-robin scheduling is used. Secondly, the `avrora` test on the Maxine VM spends more than half of its execution time in the Linux kernel on `*-Real` configurations. However, ZSim is capable of simulating only user-level code, significantly over-estimating `avrora`'s performance. These limitations should be taken into consideration when using the presented platform.

5.4 Summary

In this chapter, the fundamentals of computer architecture simulation were presented, and various simulation techniques and their design trade-offs were described. The examples of simulators implementing mentioned techniques were provided.

In addition, this chapter discussed the open-source research simulator called ZSim in detail and compared it with other simulators. On the basis of this comparison and validation, ZSim is selected to simulate managed workloads executed on top of Maxine

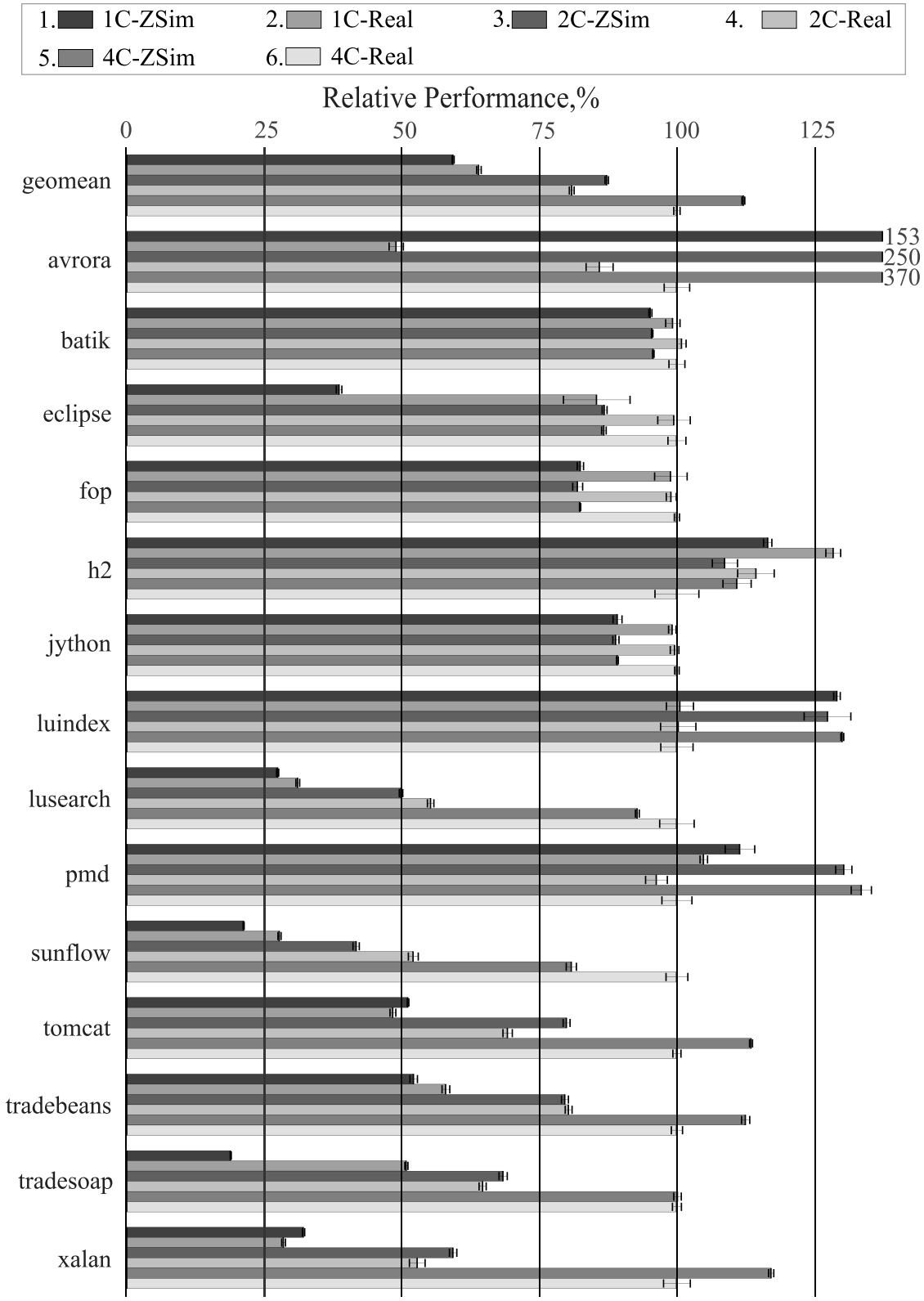


Figure 5.1: Validation of different simulated HW configurations `*-ZSim` against real system configurations `*-Real`. The depicted performances are relative to `4C-Real` (higher is better).

VM, which was described in the previous chapter.

Thus, this chapter provides background knowledge for the material of the next two chapters, which describe a novel simulation platform for productive research in the areas of HW/SW co-design of general-purpose CPUs and MREs and its use cases.

Chapter 6

MaxSim: A Simulation Platform for Managed Applications

Managed applications, written in programming languages such as Java, C# and others, represent a significant share of workloads in the mobile, desktop, and server domains. Microarchitectural timing simulation of such workloads is useful for characterisation and performance analysis, of both hardware and software, as well as for research and development of novel hardware extensions.

This chapter introduces MaxSim, a novel simulation platform based on the Maxine VM, the ZSim simulator, and the McPAT modelling framework. MaxSim is able to simulate fast and accurately managed workloads running on top of Maxine VM and its capabilities are showcased with novel simulation techniques for: (1) low-intrusive microarchitectural profiling via pointer tagging on the x86-64 platforms, (2) modelling of hardware extensions related, but not limited to, tagged pointers, and (3) modelling of complex software changes via address space morphing.

Low-intrusive microarchitectural profiling is achieved by utilising tagged pointers to collect type- and allocation-site- related hardware events. Furthermore, MaxSim allows, through a novel technique called address space morphing, the easy modelling of complex object layout transformations. Finally, through the co-designed capabilities of MaxSim, novel hardware extensions can be implemented and evaluated.

MaxSim's capabilities are showcased by simulating the whole set of the DaCapo-9.12-bach benchmarks in less than a day while performing an up-to-date microarchitectural power and performance characterisation. Furthermore, a hardware/software co-designed optimisation is demonstrated that performs dynamic load elimination for array length retrieval achieving up to 14% and 7% geometric mean L1 data cache loads

reduction and up to 4% and 2% geometric mean dynamic energy reduction.

MaxSim is available at `https://github.com/arodchen/MaxSim` released as free software. Appendix B describes how to use the MaxSim platform. The material in this chapter was presented at the *ISPASS 2017* conference [RKN+17].

The practical goal of the work described in this chapter is to build a simulation platform integrating a simulator and a HLL VM implementation together in order to achieve the 3rd research aim of the work described in Chapter 1. The result of this effort can be measured qualitatively in terms of usefulness of the developed platform by comparing its features against other state-of-the-art simulation platforms.

6.1 Introduction

Managed runtime environments (MRE) have been widely adopted in a variety of computing domains ranging from mobile phones to enterprise servers. Managed languages, and Java, in particular, have been utilised not only in application and middleware domains but also in system programming for the development of research prototypes such as the Maxine *Virtual Machine* (VM) [WHVDV+13], Jikes RVM [AAC+99], the Singularity operating system [LH10], the Graal compiler [Gra16], and the Truffle [WW12] *Abstract Syntax Tree* (AST) interpreter.

The end of single-core scaling [Moo65, DGR+74] makes the achievement of further energy and performance improvements, solely by enhancements in *Hardware* (HW), an extremely challenging task. A way to address this challenge is to design domain-specific HW extensions for certain *Software* (SW) tasks in general, and for managed workloads in particular. In order to design HW extensions that address distinctive features of managed workloads, such as object orientation and *Garbage Collection* (GC), a specialised simulation platform is needed to improve research productivity. Such a platform must enable close integration of a fast and accurate microarchitectural simulator and a modern MRE while providing a feedback loop between these two components. In this chapter, MaxSim is presented which is a simulation platform targeting managed applications.

MaxSim, in contrast to previous efforts described in Section 6.4, allows fast, accurate, and low-intrusive performance analysis of managed workloads by employing a novel pointer tagging scheme. Fast, accurate, and low-intrusive performance analysis is typically performed by utilisation of HW counters [SHC+04, HSDH04], which has three main limitations. First, the frequent accesses to HW counters can introduce

performance overheads. Second, the association of collected events with high-level information related to managed workloads can be limited [GBEDB04]. Finally, HW counters are not always portable between architectures and may not be complete for arbitrary purposes. Also in MaxSim, the simulator has an awareness of the VM, so it is able to distinguish what code is being executed (GC, non-GC) and what data is being accessed (thread local storage, stack, heap, code cache, native).

In detail, the work documented in this chapter contributes the following:

- **MaxSim – a novel experimental platform for HW/SW co-design exploration** on the basis of the state-of-the-art Maxine VM, the ZSim microarchitectural simulator [SK13], and the McPAT power, area, and timing modelling framework [LAS+13].

- **A novel pointer tagging scheme in x86-64 architectures** that is based on *Dynamic Binary Translation* (DBT) that: (1) allows the fast, accurate, and low-intrusive fine-grain microarchitectural profiling of managed workloads, and (2) enables the implementation of HW/SW co-designed optimisations, such as HW-assisted retrieval of array lengths encoded in object pointers. In addition, the collected profiling information can be also loaded back to the Maxine VM, creating a full feedback loop between the simulator and the VM.

- **A novel address space morphing technique** for simulating complex software changes regarding object layout transformations such as fields expansion, contraction and reordering.

The techniques, implemented in MaxSim and described in this chapter, are applicable to other simulators and runtime systems. However, the selection of the state-of-the-art Maxine VM and ZSim simulator provides a unique combination of research productivity, accuracy and speed of simulation.

The chapter is organised as follows: Section 6.2 describes the MaxSim platform and introduces the novel simulation and optimisation techniques. Section 6.3 presents the use cases of the proposed platform. Finally, Section 6.4 presents the related work, while Section 6.5 summarises this chapter. The experimental platform presented in this chapter is available at `https://github.com/arodchen/MaxSim` released under the GPLv2 free software license.

6.2 Integration of Platform Components and Novel Simulation Techniques

In this section, the novel features of MaxSim along with its capabilities are described in detail. These features are: (1) pointer tagging that can be used for light-weight object-based microarchitectural profiling and/or HW/SW co-designed optimisations, (2) integration with the McPAT framework for power and energy estimations, and (3) the address space morphing technique allowing the easy modelling and performance/power estimation of complex object layout transformations. The implementation details and the log of changes are available at `https://github.com/arodchen/MaxSim`.

6.2.1 Pointer Tagging

A pointer tag is a number of bits of an address which are ignored during memory access operations. In general, the main use cases of tagged pointers are: (1) capability-based addressing [Fab74, Lev84] and security [DKK07, CC04, Poi17], which can also require tagged memory, and (2) storage of type information [Org73, Bab00, HSH81]. The shift from 32-bit to 64-bit architectures enables 16 exabytes of memory to be addressable, a number which significantly exceeds the amount of memory needed for applications targeting these architectures. This fact motivated the support for tagged pointers in the modern commodity widely used architectures: AArch64 with 8-bit pointer tags [ARM15] and Sparc M7 with up to 32-bit pointer tags [Phi14].

Although x86-64 architectures do not currently support tagged pointers (see Sect. 3.3.7.1 in [Int11]), the virtual addressing is currently limited to 48 bits[1] with the high 16 bits replicating bit 47. MaxSim exploits these high 16 bits on x86-64 architectures, to encode extra information that can be interpreted during simulation for various purposes. The main use case is the assignment of extra information to an object via its pointer. This extra information can regard either associations with high-level language features (Section 6.2.1) or other metadata for HW/SW co-designed optimisations (Section 6.2.1).

Typically, associating extra information with objects poses a trade-off between extra required memory and access time. Figure 6.1 presents three options for the storage of object metadata. The first option is "in object storage", where the metadata is stored inside an object in an intrusive manner which also increases memory footprint. The

[1] In the upcoming version of the architecture, the virtual addressing will be extended to 57 bits [5-L16].

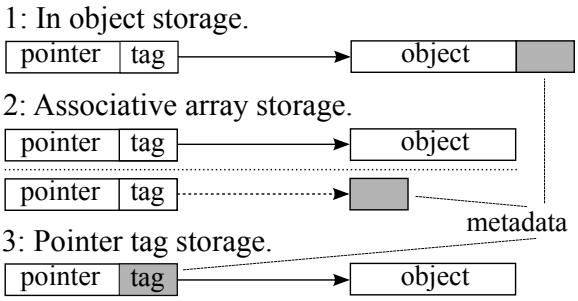


Figure 6.1: Different options for object metadata storage.

second option is "associative array storage" which requires both extra space and lookup time to retrieve metadata. To that end, if metadata is accessed read-mainly and frequently (on every memory access operation) and the amount of metadata to be stored can fit in 16 bits, "pointer tag storage" is preferable which is the third option. Encoding metadata into the available 16 bits of an object's address saves memory bandwidth and reduces access latency.

To enable tagged pointers support in MaxSim, the following three invariants must be preserved in Maxine VM:

1. All pointers to the same object must be tagged with the same tag.

2. When a field inside an object is accessed, `[tag:base + (index * scale) + disp]` addressing mode must be used, where `base` points to the beginning of the object and `(index * scale) + disp` represents an `offset` (later on, this will be referred to as `[tag:base + offset]`).

3. An object pointer tag is immutable between any following adjacent points in an object's lifetime: object allocation, initialisation, and evacuation during GC.

The first invariant allows the comparison of tagged pointers without extensive VM modifications, while the second invariant allows an accessed object's class field to be identified using this canonical form. The third invariant implies that the pointer tag can only be changed in certain places, where all pointers to an object to be tagged are accessible without a full scan of all objects. All live objects are untagged during a stop-the-world VM operation when switching to the ZSim fast forwarding mode [SK13]. During the ZSim fast forwarding mode, execution happens without simulation and extensive binary modification/instrumentation at near-native speed until entering the next *Region Of Interest* (ROI) for simulation. Untagged object pointers are tagged back

during the stop-the-world VM operation when entering the next ROI and switching back from the fast forwarding to the normal simulation mode.

```
// Returns true if instruction operand is a memory reference.
BOOL
INS_OperandIsMemory(INS ins, UINT32 n);

// Returns base register (address = base + disp + index * scale).
REG
INS_OperandMemoryBaseReg(INS ins, UINT32 n);

// Returns index register (address = base + disp + index * scale).
REG
INS_OperandMemoryIndexReg(INS ins, UINT32 n);

// Returns scale (address = base + disp + index * scale).
UINT32
INS_OperandMemoryScale(INS ins, UINT32 n);

// Returns displacement (address = base + disp + index * scale).
INT64
INS_OperandMemoryDisplacement(INS ins, UINT32 n);

// Rewrites memory operand to reference the other memory location.
VOID
INS_RewriteMemoryOperand(INS ins, UINT32 memIndex, REG reg);
```

Listing 6.1: Pin API for tag pointers retrieval and untagging.

Finally, ZSim simulation is based on the Pin dynamic binary instrumentation and modification tool, and pointers' tag detection and untagging is performed via the API shown in Listing 6.1. To summarise, pointer tagging allows to: (1) perform light-weight object-based microarchitectural profiling and, (2) perform a number of HW/SW co-designed optimisations by encoding data in tagged pointers.

Light-weight Object-based Microarchitectural Profiling

Simulation-based profiling, an important technique in performance analysis, is one of the key features of MaxSim. In order to enable this functionality, it is essential to

bind microarchitectural events with high-level language information. This binding is achieved via the pointer tagging mechanism described in the previous section.

The Maxine VM assigns a tag to a pointer, and ZSim collects events related to this tag during memory access operations. MaxSim currently supports several implementations of language information association with object pointers among which are: `ClassIdTagging` and `AllocationSiteIdTagging`. `ClassIdTagging` assigns object class IDs to all object pointers allowing the association of microarchitectural events per class. A class ID is a compact unsigned integer representing the class of an object and is usually stored in the class information object which is accessible via a pointer stored in an object's header. By storing class ID in the pointer tag, it is possible to save two load operations at the expense of untagging and tag retrieval, which are two and one shift operations, respectively. `AllocationSiteIdTagging` assigns allocation site IDs to object pointers. An allocation site ID is a compact unsigned integer representing a pair of an allocation site estimation of an object and a class ID. Allocation site IDs are requested from ZSim via magic NOP operations [MSB+05], which have NOP semantics during non-simulated execution. On each allocation site ID request, ZSim returns a compact ID, which is associated with an allocated object's class ID and an allocation site estimation using stack trace estimation in ZSim. Stack trace estimation is performed using per-thread circular buffers by pushing return addresses on function calls and popping them on function returns.

The state-of-the-art techniques to associate allocation sites with objects usually require either hashing [OOK+10] or storage of extra information in or adjacent to objects [CPST15]. Such techniques introduce noticeable overheads and interference with a normal workload execution. In comparison with the aforementioned techniques, the proposed technique is much less intrusive, as it takes just a few lightweight operations during an object allocation to set a tag.

Figure 6.2 shows the integration scheme of MaxSim and the flow of profiling information between its components. The profiling information is stored in the Protocol Buffers format [Pro14], and it consists of two parts. The first part, stored in the `ZSimProf.db` file, contains microarchitectural events collected by ZSim. Examples of such events (memory accesses and cache misses) related to a class field are shown in Figure 6.2. The second part, stored in the `MaxineInfo.db` file, contains information necessary to bind collected events to high-level language information. In Figure 6.2, for example, field information (name, class ID, offset) is represented. In case there are several ROIs during the same simulation, several `ZSimProf.db` files and a single

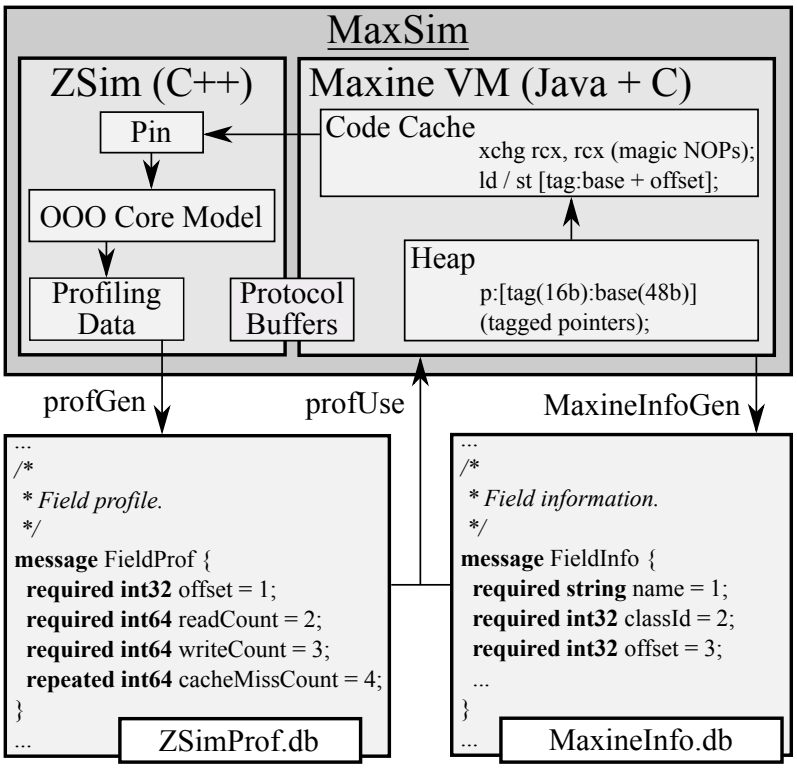


Figure 6.2: Handling of profiling information in MaxSim.

`MaxineInfo.db` file will be generated.

The profiling is performed during memory access operations, and collected events are associated with triplets of an instruction pointer, a pointer tag, and a memory address offset. Allocation site IDs are reported from ZSim to Maxine via magic NOPs with an allocation site ID stored in the `rcx` register by ZSim.

```
// Memory access profiling.
java.util.HashMap$Entry[](...)@
  [java.util.HashMap.<init>(...)+354(...)]
    (asi:0 mf:2976(s:152(19) s:88(1)) ac:983 ... l3rm:11 l3wm:7):
      (o:80 r:33 w:9 ... l3rm:9 l3wm:0)

// L3 cache miss reads profiling.
[java.util.HashMap.put(Object, Object)+107(...)]
  (m:9 asi:0 ol:80 oh:80)
```

Listing 6.2: Snippet of profiling information textual output.

The detailed collected information can later be uploaded to Maxine VM to guide

optimisations, or it can be printed in a textual format. The snippet of the textual output is presented in Listing 6.2. In this example, `AllocationSiteIdTagging` was active, and the profiling information is shown for objects of `HashMap$Entry[]` class allocated during a call to `HashMap.<init>` method at offset `354` (in the constructor of HashMap). In total, `1` object of `88` bytes and `19` objects of `152` bytes were allocated reaching a total allocation footprint of `2976` bytes. Furthermore, `983` memory accesses were performed with `11` L3 cache read misses and `7` L3 cache write misses. At offset `80`, `33` reads and `9` writes were performed with `9` L3 cache read misses. Finally, all `9` misses at offset `80` occurred at offset `107` of method `HashMap.put`. The presented tagged-based profiling scheme is especially useful for profiling object-oriented SW in which objects can be relocated (*e.g.* copying garbage collection), as pointer tags preserve objects' identities for profiling.

HW/SW Co-designed Optimisations Enabled by Tagged Pointers

The presence of available bits, when tagged pointers are enabled, creates a number of HW/SW co-designed optimisation opportunities. It is possible to encode some information related to an object in a pointer tag and to extend functionality of memory access operations via a tag for performance/power optimisations or security enhancements. An example of such an optimisation is related to array length encoding in tags and is one of the use-cases of this chapter. Its evaluation is presented in Section 6.3.2.

6.2.2 Integration with the McPAT Framework

To be able to perform energy estimations, the energy estimation model (which uses McPAT) from the Sniper simulator [HSC+12] for the same microarchitecture simulated by ZSim was integrated . Conversion of microarchitectural events from the ZSim to Sniper format was adopted from the ZSim-NVMain simulator [ACU15]. The modelling tool required the collection of a number of extra microarchitectural events in ZSim such as the number of predicted branches and floating point microoperations.

6.2.3 Simulator/VM Co-Operative Address Space Morphing

For many managed languages in general, and for Java in particular, layouts of objects in memory are not specified and depend on the VM implementation. Changing layouts of objects can improve cache locality and decrease memory footprint. However, such

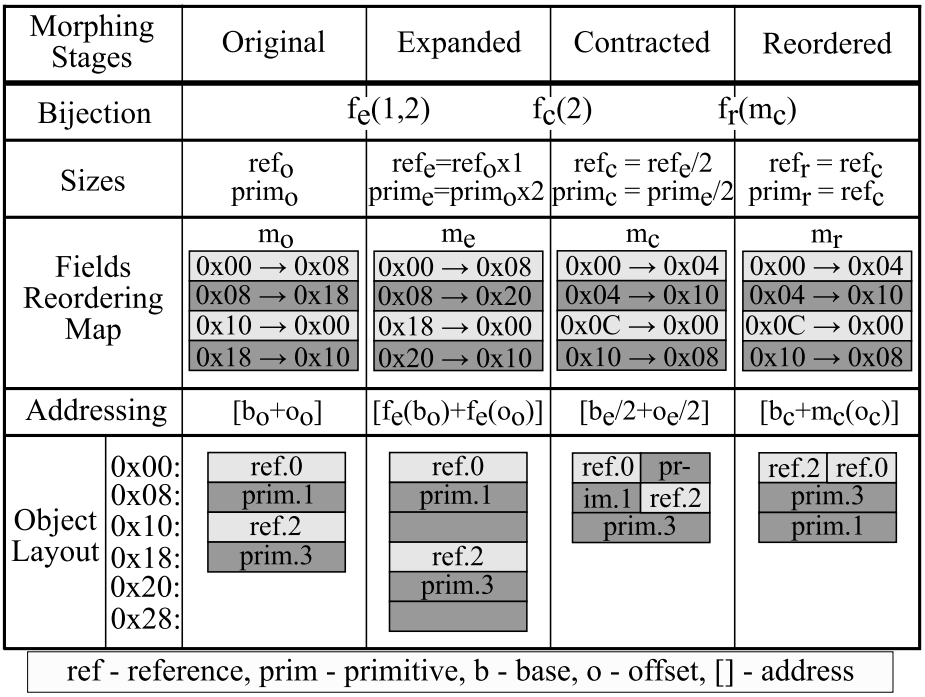


Figure 6.3: Example of address space morphing in MaxSim.

transformations are difficult to implement without adding extra complexity or breaking the modularity of a VM. MaxSim implements a novel address space morphing technique to perform simulation of complex object layout transformations, specifically fields expansion, contraction, and reordering.

As shown in Figure 6.3, the proposed technique is a co-operative multi-stage object layout transformation. Furthermore, it leverages the flexibility of Maxine VM to expand object fields and the ability of ZSim to remap memory addresses during simulation. Thus, in order to perform fields reordering and contraction by a factor of `N`, the following three stages are performed: (1) all fields except those to be contracted are expanded by a factor of `N` by Maxine VM, (2) ZSim contracts the heap by a factor of `N` via address space remapping, and (3) ZSim remaps the offsets of the fields according to the provided reordering map.

In the example of Figure 6.3, the original object layout has two reference fields, `ref.0` and `ref.2`, and two primitive fields, `prim.1` and `prim.3` (the leftmost object layout). During simulation, it is morphed in three stages to the new layout (the rightmost object layout) which results in its fields being reordered, as described by the m_o reordering map, and its references being contracted by a factor of `2`. In order to perform such transformations, four parameters to three bijections are provided. The

first bijection f_e from the `Original` to the `Expanded` space takes two arguments: `1` - expansion factor for references, `2` - expansion factor for primitives. The transformation defined by this bijection is performed via changing layouts of objects in Maxine VM. The fields reordering map m_o is also modified according to this bijection. The second bijection f_c from the `Expanded` to the `Contracted` space takes the contraction factor as its argument. This transformation is performed in ZSim by dividing by `2` bases and offsets of memory access operations to objects. Furthermore, the fields reordering map m_e is modified by dividing by `2` all to-offsets. The third bijection f_r from the `Contracted` to the `Reordered` space takes the reordering map m_c from the `Contracted` stage and performs fields reordering according to this map resulting in the simulation of the desired layout. Heap and thread-local allocation buffer sizes are also doubled in Maxine VM on the `Expanded` stage.

Another issue that should be considered during simulator/VM co-operative address space morphing is expanded objects copying and initialisation. After expanding primitives twice in Maxine VM, it will take twice as many dynamic instructions to perform copying or initialisation than it would take in the case of the final layout presented in the example. This issue is handled via filtering during simulation of execution of object copying and initialisation which happens in a loop. In this loop, every second iteration is omitted from the timing simulation. The indication that loop filtering should be enabled or disabled is performed by the VM via magic NOP operation in the loop's prologue and epilogue respectively. An example of such loop, with filtered iterations, is shown in Listing 6.3.

```
// Sets value v for n words at pointer p.
void
setWords( Pointer p, int n, Word v) {
  // loop prologue
  zsimMagicOp( FILTER_LOOP_BEGIN, p);
  for ( int i = 0; i < n; i++ ) {
    writeWord( p, i * WORD_SIZE, v);
  }
  // loop epilogue
  zsimMagicOp( FILER_LOOP_END);
}
```

Listing 6.3: Example of loop iterations filtering.

In order to validate the proposed address space morphing simulation technique, the

following experiment was performed. Both references and primitives of heap objects were expanded twice in the Maxine VM via the bijection $f_e(2,2)$. During simulation in ZSim, memory accesses to expanded fields are projected back to original unexpanded address space (by contracting twice) via the bijection $f_c(2)$, thus simulating the original object layout. No fields reordering is performed via the bijection $f_r(\varnothing)$. The execution times were compared to the simulation of the original object layout, and the measured execution time geometric mean difference was less than 1% for the DaCapo benchmarks validating the proposed technique.

The simulation of objects' fields reordering transformation via address space morphing is driven by a configuration file passed to MaxSim in the Protocol Buffers format, presented in Listing 6.4. Fields reordering is described by the type descriptor `typeDesc` to be simulated having a different layout. The objects to be simulated as having an alternative layout are tagged by a `transTag`. On memory accesses to objects tagged by a `transTag`, address remapping is done during simulation by using an associative array represented by `fieldOffsetRemapPairs`, replacing matching `fromOffset` by `toOffset` during simulation. This technique allows fast experimentation with various objects layouts. It also allows having different layouts of objects of a superclass and its subclasses so that the same field can have different offsets in them.

```
// Fields offset remapping pair.
message FieldOffsetRemapPair {
   required int32 fromOffset = 1;
   required int32 toOffset = 2;
}

// Data transformation information.
message DataTransInfo {
  required string typeDesc = 1;
  required int32 transTag = 2;
  repeated FieldOffsetRemapPair fieldOffsetRemapPairs = 3;
}
```

Listing 6.4: Configuration file in the Protocol Buffer format driving fields reordering transformation simulation.

Expansion and contraction of references and primitives via address space morphing allow simulating ordinary object pointers compression [ATBC+04] in MaxSim. Compression of object pointers [ATBC+04] can lead to significant improvements in

Name		4C	1CQ
Cores	type	x86-64 Nehalem OOO core at 2.66 GHz	
	total	**4**	**1**
Prefetchers		disabled	
L1I caches		32KB, 4-way, LRU, 3-cycle latency	
L1D caches		32KB, 8-way, LRU, 4-cycle latency	
L2 caches		256KB, 8-way, LRU, 6-cycle latency	
L3 cache	type	16-way, hashed, 30-cycle latency	
	size	8MB	2MB
Memory controller		1, 3 DDR3 channels, 47-cycle latency	
DRAM		3GB, DDR3-1066, 1GB DIMM per channel	

Table 6.1: ZSim configurations.

memory utilisation, and it is used in the 64-bit JVM implementations [Ope17]. An example of another transformation which could be implemented and simulated via the presented technique is a replacement of precisions and sizes of certain fields to different ones (long to int or double to float). To summarise, address space morphing allows to evaluate the performance impact of changing the order and/or size of fields.

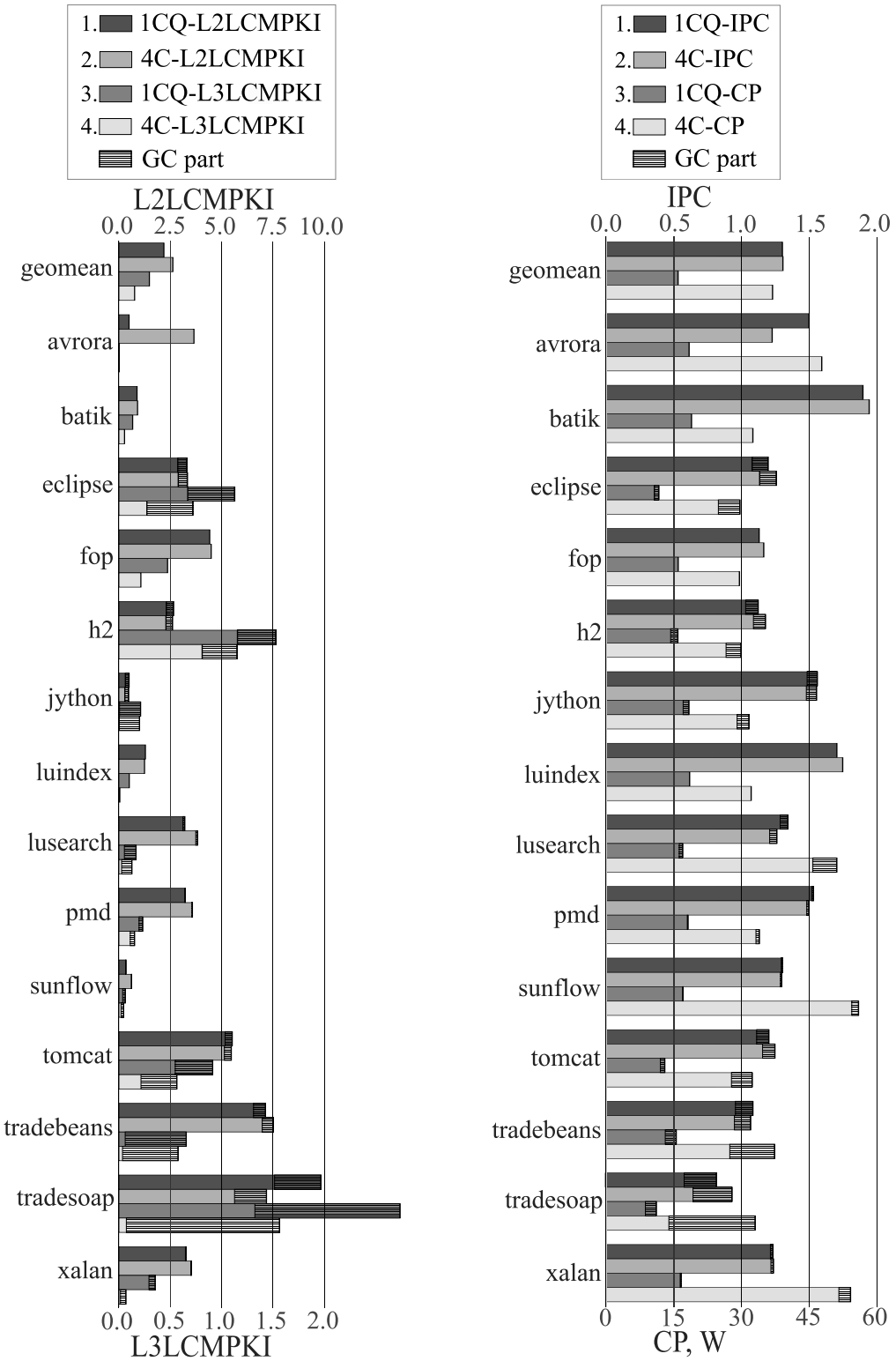
6.3 Use Cases

This section will present two use cases of MaxSim. The first one regards the microarchitectural characterisation of the DaCapo benchmarks. The second use-case showcases simulation of the architectural extensions related to the retrieval of array lengths stored in pointer tags.

The parameters of the simulated systems referenced in the use cases are described in Table 6.1. The `4C` configuration represents the Intel quad-core Nehalem microarchitecture. Furthermore, the `1CQ` configuration represents the single-core CPU with just a **Q**uarter of the 8MB *Last Level Cache* (LLC). The `1CQ` configuration was used in order to simulate the case when only a quarter of the available `4C` resources is available to the workload (if the LLC could be partitioned).

6.3.1 Characterisation of the DaCapo Benchmarks

Workload characterisation is crucial for performance analysis of both HW and SW. MaxSim was able to simulate the whole set of the DaCapo-9.12-bach benchmarks in less than a day, with the results depicted in Figures 6.4a and 6.4b. Figure 6.4a shows the



(a) L2 and L3 Load Cache Misses Per Kilo Instruction (LCMPKI)

(b) Instructions Per Clock (IPC) and Consumed Power (CP)

Figure 6.4: Characterisation of the DaCapo-9.12-bach benchmarks on MaxSim.

L2 and L3 *Load Cache Misses Per Kilo Instruction* (LCMPKI) for both configurations. As shown, the majority of the DaCapo benchmarks are not cache-miss-intensive, which corresponds with the previous findings [IN12]. Figure 6.4b contains the information on *Instructions Per Clock* (IPC) and *Consumed Power* (CP). The geometric mean IPC is close to 1.4, while the CP is between 10 and 60 watts depending on the configuration. Hatched parts of the bars in Figures 6.4a and 6.4b represent parts of the presented metrics related to *Garbage Collection* (GC).

6.3.2 Evaluation of the HW/SW Co-Designed Optimisation Related to Array Length Encoding into Array Object Pointers' Tags

Implementations of managed languages associate array lengths with array objects allowing them to perform array bound checks at runtime. Moreover, outside the scope of managed languages, SW-based integrity checking packages may retrieve array lengths before memory access operations to verify that the accessed addresses are within array bounds. A common way of storing an array length is inside an array object at some constant offset from a base pointer. In Maxine VM, array lengths are stored at offset `0x10` of an array object and can be in the range of $[0; 2^{31} - 1]$.

Having 16-bit pointer tags, it is possible to store a range of array lengths $[0; 2^{16} - 2]$. The value $2^{16} - 1$ serves as a *Not an Array Length* (NaAL) indicator. The retrieval of an array length can be performed via the method shown in Listing 6.5. This code can be emitted in 5 instructions of 19 bytes size with an average execution height of 4.5 instructions. On the contrary, the baseline scheme utilises just one instruction of four bytes size (load by `objectAddress` at offset `0x10`).

```
// Retrieving array length.                    // ! x86-64 GNU Assembler
inline int                                     // ! %rdi = objectAddress
retrieveArrayLength( Address_t objectAddress) { // movq %rdi, %rax
  TAG_t tag = extractTAG( objectAddress);      // shrq $48, %rax
  if ( tag != NaAL ) {                         // cmpq $65535, %rax
    return (int) tag;                          // jne .L1
  }                                            //
  return * ((int *) (objectAddress + 0x10));   // movq 16(%rdi), %rax
}                                              // .L1:
                                               // ! array length in %rax
```

Listing 6.5: Array length retrieval with tagged pointers.

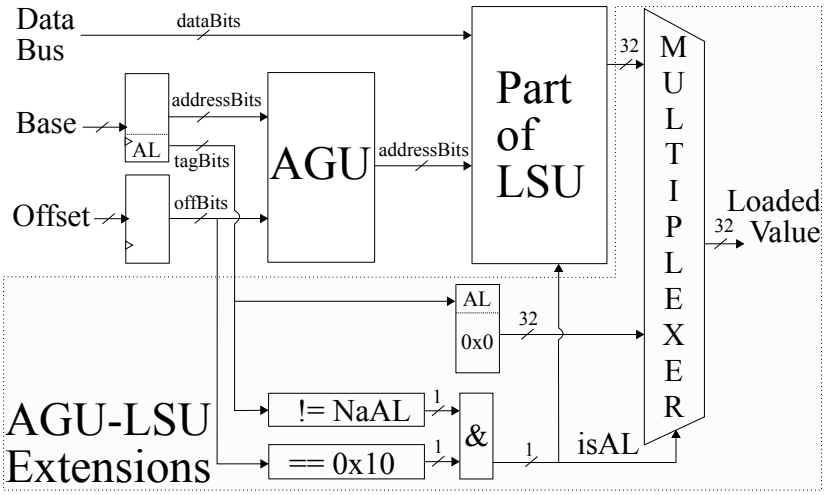


Figure 6.5: Extensions to Address Generation Unit (AGU) and Load Store Unit (LSU) for array length retrieval from tagged pointers.

In order to perform the whole code snippet in just one instruction, corresponding to the load instruction preceding the return statement in Listing 6.5, the HW extension shown in Figure 6.5 is proposed. The presented HW extension relies on the invariant, preserved by the VM, that the array length field of an array object is always accessed via a `[tag:base+offset]` addressing mode. Furthermore, the `ArrayLengthTagging` scheme has to be enabled. In this scheme, all non-array objects and arrays with lengths greater than $2^{16} - 2$ are tagged with the `NaAL` tag, while all the other array objects are tagged with their lengths. Thus, when an array length is accessed, the aforementioned tagged address pattern can be identified by the *Address Generation Unit* (AGU) in the proposed HW extension. Upon detecting an access to an array length field, which is also encoded in a pointer tag, the `isAL` signal is set. Consequently, the value `AL` from the tag bypasses the *Load-Store Unit* (LSU) on its way to a consumer.

The values of the matching offset (`0x10`) and the matching tag (`NaAL`) for the presented AGU extension can be fixed or variable. In the latter case, these values can be set via a control register, making this scheme more general. If an array length is loaded from a pointer tag then one cycle latency is assumed, which is modelled in the ZSim simulator.

The proposed HW/SW co-designed optimisation is evaluated on the DaCapo-9.12-bach benchmarks on the `1CQ` and `4C` ZSim models of Table 6.1. Figure 6.6 presents the results for *L1 Data Cache Loads* (L1DCL) and *Dynamic Energy* (DE) reductions. The proposed extensions to the AGU and the LSU were not added in the power estimation model as the energy overhead of these functional units extensions is significantly less

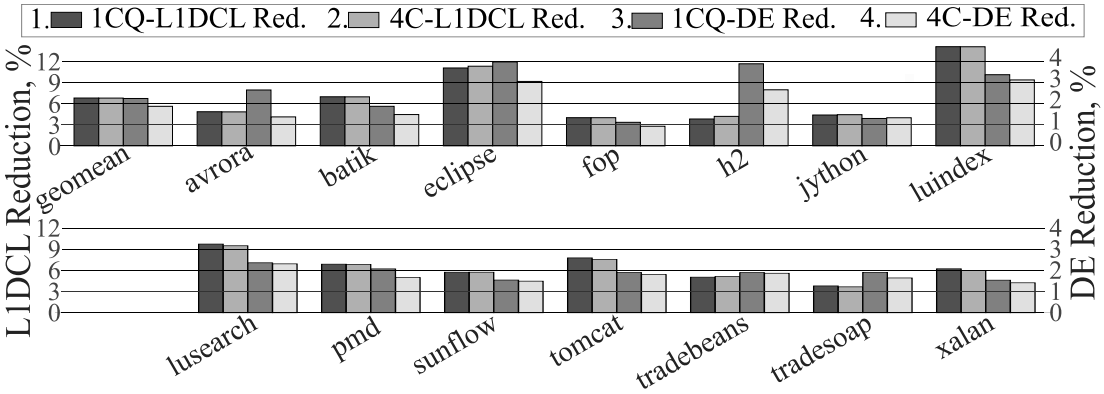


Figure 6.6: L1 Data Cache Loads (L1DCL) and Dynamic Energy (DE) Reductions on the DaCapo-9.12-bach benchmarks after employing the HW/SW co-designed optimisation related to array length tagging.

than the energy savings from the reduction of memory traffic. Although no significant performance gains were observed, the proposed technique resulted in up to 4% and 2% geometric mean dynamic energy reduction, and up to 14% and 7% geometric mean L1 data cache loads reduction.

6.4 Related Work

The closest platform [Jik14] allowing user-level simulation of managed workloads is based on the Sniper multi-core simulator [CHE11] and the Jikes RVM [AAC+99]. The main limitation of this platform against MaxSim, is that it only supports 32-bit Jikes RVM and is not capable of running the full set of the DaCapo benchmarks. Regarding the simulator, Sniper uses the instruction-window centric *Out-Of-Order* (OOO) core model [CHE+14] with an average relative error of 11% for single-core and 21% for eight-core simulations on the SPLASH-2 benchmarks [WOT+95]. It is very close to ZSim's average relative error, which on a selection of tests from PARSEC [BKSL08], SPLASH-2, and SPEC OMP2001 [ADE+01] is 10% for single-core and 11% for six-core simulations. The tandem of Sniper and Jikes was used to explore a number of HW/SW co-designed techniques. These techniques improve memory bandwidth and reduce power and energy consumption by preventing write backs of cache lines containing parts of dead objects and by preventing fetches-on-writes while initialising cache lines containing parts of newly allocated objects with zeros [SHB+14].

The platform described in [WMGW06] is based on the Hotspot JVM and the full-system Simics simulator [MCE+02]. It does not require any changes to the Hotspot

JVM and it can be very helpful in non-disruptive simulation-based performance analysis. It has high visibility of the Java high-level information (with the exception of thread and stack state). The design goal of that platform was to decouple it as much as possible from the concrete JVM implementation via a clear interface. The MaxSim platform, on the contrary, followed the co-design approach of the VM and the simulator development to facilitate extra functionality.

The ZSim simulator is written in C++, and communication of high-level information with the Maxine VM happens via Protocol Buffers. If the simulator was written in Java, the communication between the two components could have happened via reflection. The simulator called Tejas [SKK+15] is written in Java and can run on any platform the Java VM can execute. However, it has two limitations: firstly, it is a trace-driven simulator, and secondly, it uses an intermediate virtual ISA, which can introduce inaccuracy.

The *Virtual Performance Analyzer* (VPA) framework [THC+14] follows the approach of partial selective simulation of HW-SW interaction. In this framework, a cycle-approximate model is used. The motivation of this approach is the observation that I/O operations are sensitive to delays, and a simulation speed above 10 MIPS should be preserved not to alter the behaviour of the program. The ZSim simulator solves this problem via the lightweight user-level virtualisation technique, achieving for the OOO model an average simulation speed of 12 MIPS (in the experiments carried out for this chapter).

Introspection of target agnostic JIT compilation in the Smalltalk VM on top of gem5 [Shi15] was shown to be useful for debugging and power/performance analysis. However, gem5 has a low simulation speed of 200 KIPS. Moreover, with Graal [Gra16] and Truffle [WW12], it could be possible to run Smalltalk and other managed languages on the presented platform in future.

6.5 Conclusions

In this chapter, the MaxSim platform was presented. It is a novel and open-source experimental platform for HW/SW co-design research and characterisation of managed workloads. MaxSim is based on the state-of-the-art Maxine VM, the ZSim microarchitectural simulator, and the McPAT power, area, and timing modelling framework. MaxSim features the simulation of 16-bit-tagged pointers, which are utilised for: (1)

low-intrusive memory access profiling, (2) tagged pointers modelling on x86-64 architectures, and (3) experimenting with novel HW/SW co-designed optimisations by extending the semantics of memory access operations via pointer tagging. In addition, the address-space morphing technique was presented, which allows modelling and simulation of complex software changes, such as compressed object pointers optimisation and other data layout transformations. MaxSim's capabilities were showcased by: (1) performing an up-to-date microarchitectural characterisation of the full set of the DaCapo benchmarks in less than a day, and (2) presenting a novel HW/SW co-designed optimisation that performs dynamic load elimination for array length retrieval achieving up to 14% and 7% geometric mean L1 data cache loads reduction and up to 4% and 2% geometric mean dynamic energy reduction. MaxSim is available at `https://github.com/arodchen/MaxSim` released as free software. This platform is used in the next chapter to explore opportunities for type information elimination from objects on architectures with tagged pointers support.

Chapter 7

Type Information Elimination from Objects on Architectures with Tagged Pointers Support

Implementations of object-oriented programming languages associate type information with each object to perform various runtime tasks such as dynamic dispatch, type introspection, and reflection. A common means of storing such relation is by inserting a pointer to the associated type information into every object. Such an approach, however, introduces memory and performance overheads when compared with non-object-oriented languages.

Recent 64-bit computer architectures have added support for *tagged pointers* by ignoring a number of bits – *tag* – of memory addresses during memory access operations and utilise them for other purposes; mainly security. This chapter presents the first investigation into how this hardware support can be exploited by a Java Virtual Machine to remove type information from objects. In addition, novel hardware extensions are proposed to the address generation and load-store units to achieve low-overhead type information retrieval and tagged object pointers compression-decompression.

The evaluation has been conducted using the MaxSim platform presented in the previous chapter. The results, across all the DaCapo benchmark suite, SLAMBench, pseudo-SPECjbb2005 and GraphChi-PR executed to completion, show up to 26% and 10% geometric mean heap space savings, up to 50% and 12% geometric mean dynamic DRAM energy reduction, and up to 49% and 3% geometric mean execution time reduction with no significant performance regressions. It is also observed that utilisation of eight-bit tags provides 99% of the achievable heap space savings.

The practical goal of this research is to improve memory utilisation, performance and energy efficiency of HLL VM implementations, and the result can be measured quantitatively as an improvement in these characteristics. The material in this chapter was published in the *IEEE Transaction of Computers 2017* journal [RKN+18].

7.1 Introduction

Managed runtime environments are extensively used in many computing domains ranging from mobile devices to cloud servers. Managed object-oriented languages have been employed not only in application and middleware domains but also in system programming for the development of research prototypes such as the Maxine *Virtual Machine* (VM) [WHVDV+13, KCR+17], Jikes RVM [AAC+99], and the Singularity OS [LH10].

Implementations of managed object-oriented languages associate object type information with each object by inserting a pointer to type information into every object as part of its object header. However such an approach, prevalent for most object-oriented languages, increases memory utilisation (and can introduce performance overheads) when compared with non-object-oriented languages.

At the same time, modern 64-bit computer architectures have added support for *tagged pointers*. In such architectures, a number of bits – *tag* – of memory addresses are ignored during memory access operations and utilised for other purposes; mainly security. Furthermore, they provide less than 64-bit addressable memory space, leaving a number of bits in object pointers for useful purposes.

In this chapter, the first investigation is presented into how tagged pointers in 64-bit architectures can be exploited by an object-oriented language implementation to remove a pointer to type information from objects. Besides, novel hardware extensions to the address generation and load-store units are proposed to achieve low-overhead type information retrieval and tagged object pointers compression-decompression, respectively. In other words, a *Hardware* (HW)/*Software* (SW) co-designed technique is explored in the context of the Java programming language, although the proposed technique is applicable to other managed and unmanaged object-oriented languages.

The key contributions of the work described in this chapter are:

- **A technique of type information elimination** from object headers on architectures with tagged pointers support. Two options are explored which are (1) a

SW-only specialisation and (2) a HW/SW co-designed solution that yields the best results.

- **Novel backward-compatible HW extensions** (1) to the *Address Generation Unit* (AGU) to efficiently retrieve type information for objects with type information elimination enabled (optimised) and for objects without (un-optimised), and (2) to the *Load-Store Unit* (LSU) to efficiently handle compression and decompression of tagged object pointers.

- **Demonstration of the MaxSim experimental platform for HW/SW co-design space exploration** on the basis of the ZSim [SK13] microarchitectural simulator, the Maxine VM, and the McPAT [LAS+13] power, area, and timing modelling framework.

- **The evaluation of the proposed technique** in the context of the MaxSim platform against the DaCapo-9.12-bach [BGH+06] benchmark suite, SLAMBench [NBZ+15], pseudo-SPECjbb2005 [PJB05] and GraphChi-PR [KBG12] on several HW models achieving up to 26% and 10% geometric mean heap space savings, up to 50% and 12% geometric mean *Dynamic Random-Access Memory* (DRAM) dynamic energy reduction, and up to 49% and 3% geometric mean execution time reduction with no significant regressions in these characteristics.

The chapter is organised as follows: Sections 7.2 and 7.3 present a survey of how type information is associated with objects in modern *Java Virtual Machine* (JVM) implementations, as well as how tagged pointer support works in modern architectures. Section 7.4 details the proposed technique along with the accompanying changes to the JVM implementation. Section 7.5 describes the proposed architectural support for retrieval of type information with extensions to the AGU and for tagged pointers load-decompression and store-compression with extensions to the LSU. Section 7.6 and 7.7 describe the experimental framework and methodology used in this work along with the obtained results, respectively. Finally, Section 7.8 discusses previous work on type information elimination, while Section 7.9 summarises this chapter.

Acronym	Full name	Description
CI	Class Information	The entity that entails the type information of an object. CI normally contains some metadata for: (1) dynamic dispatch (virtual method table), (2) classification (pointer to supertype), (3) object layout description (size, fields information), and (4) other implementation-dependent metadata.
CIP	Class Information Pointer	A raw pointer to (address of) the CI. Typically stored in extra words preserved for each object before its content.
CID	Class Identifier	A compact non-negative integer identifier of a CI.

Table 7.1: Glossary of terminology.

7.2 Association of Objects with Class Information in JVMs

In this section, it is explained how type information is typically associated with objects in a number of modern industrial-strength and research JVMs. Although the focus of this chapter is on type information elimination from object headers in the context of the JVM implementation and the Java language, it is important to mention that the proposed ideas are applicable to C++ and other non-managed and managed object-oriented languages with runtime type information associated with objects. Hereafter, the terms *Class Information* (CI), *Class Information Pointer* (CIP), and *Class Identifier* (CID) will be used, whose descriptions are presented in Table 7.1.

Figure 7.1 presents the relationship between an object, its pointer, and its associated CI. When a new object is allocated on the heap, its pointer is stored on the stack. When an object tests if it is an instance of some class, type introspection happens via data stored in the associated CI, which is referenced by the CIP stored in the object. Aside from the CIP and the fields of the object's class, an object reserves some extra space for special *miscellaneous data* (MISC) that can be associated with it during its lifetime.

The small survey below concerns only 64-bit JVMs since, to the best of the author's knowledge, there are no modern 32-bit systems that support tagged pointers. The layouts of object headers, to be discussed below, are presented in Figure 7.2.

HotSpot. On 64-bit systems, the first eight bytes of an object are dedicated to the "mark word". This word is multi-purpose and is currently used for hashing, locking, and *Garbage Collection* (GC) information. The second eight bytes of an object contain

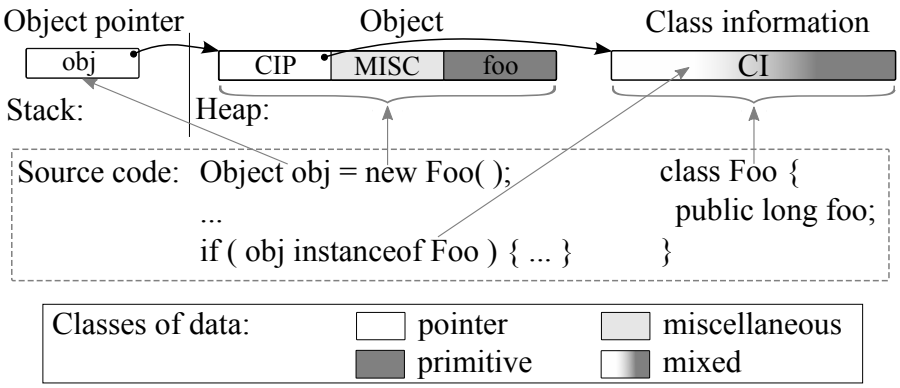


Figure 7.1: Object and class information association in VMs.

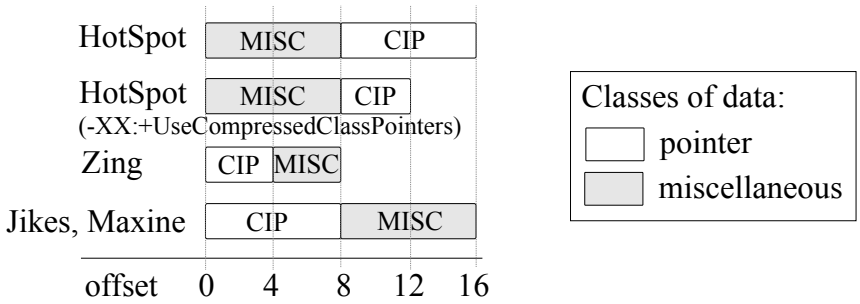


Figure 7.2: Layout of object headers in various 64-bit JVMs.

the "klass pointer" (essentially a CIP), which can be reduced to four bytes when the flag `-XX:+UseCompressedClassPointers` is used (enabled by default in OpenJDK 8 [Ope17]).

Zing. Azul's Zing uses only eight bytes as an object header, four of which are dedicated to a compressed CIP [Cli10].

Maxine. The meta-circular research Maxine VM, written mostly in Java, uses two words as an object header in the default object layout scheme. The first eight bytes contain the reference to the "hub" (essentially a CIP) or a forwarding pointer during copying GC. The second eight bytes are dedicated to the MISC word responsible for object locking and hashing.

Jikes. The meta-circular 64-bit Jikes RVM (which is also written mostly in Java), like Maxine, has a two-word object header layout. The main difference is that a forwarding pointer is stored in the second word.

From the JVM descriptions, it can be noticed that the majority of production-quality JVMs use CIP compression in order to minimise the memory footprint of objects. Fields of objects in the described VMs are located at positive offsets after their headers (by default). Although it is possible to lay out an object in memory in

fragments [CDL99], in all other sections, without loss of generality, it is assumed that objects are allocated in contiguous blocks of memory.

7.3 Architectural Support for Tagged Pointers

This section provides an overview of the latest computer architectures that support tagged pointers and succinctly how they implement this support. The shift from 32-bit computing to 64-bit has started from server and desktop deployments and continued to embedded computing after the introduction of the ARM 64-bit processor family. Current 64-bit architectures, normally, provide less than 64-bit addressable memory space leaving a number of bits of an address unused. Dealing with these unused bits is architecture-dependent, and the following paragraphs describe how several modern architectures handle them.

AArch64. ARM's latest 64-bit AArch64 architecture provides support for tagged pointers. Virtual address tagging in AArch64 is enabled by setting the `Top Byte Ignore` field in the `TCR_ELn` control register. In this case, the high eight bits are ignored during addressing and can be utilised by the developers in an unmandated way. However, hints for exploitation in object-oriented languages are given in the associated programmer's guide [ARM15]. The most recent `ARMv8.3-A` version of the architecture [Bra16] features pointer authentication to prevent unauthorised memory accesses and associated exploits [Poi17].

Sparc M7. Oracle's Sparc M7 architecture also provides tagged pointers support. Sparc M7 supports virtual address masking, allowing the use of 8, 16, 24 or 32-bit metadata. This metadata is, consequently, ignored by the underlying HW during addressing [Phi14, AJK+15].

x86-64. In current Intel's and AMD's x86-64 architectures virtual addressing is limited to 48 bits, while the high 16 bits of the virtual address are required to replicate bit 47. Consequently, tagged addressing is not supported on such architectures [Int11] (Sect. 3.3.7.1). However, the property that 16 bits are not effectively used can be utilised during simulation, which was showcased in the previous chapter. In future generations of processors, virtual addressing will be limited to 57 bits [5-L16].

From the architectural descriptions, it can be noticed that modern off-the-shelf CPUs offer tagged pointer support. Tagged pointers are ideal candidates for storing CIDs in tags, thus eliminating CIPs from frequently allocated objects.

7.4 Class Information Handling via Tagged Pointers

This section describes how tagged pointers can be utilised to associate object pointers with CIs, what additional data structures are required in a VM, and how objects with eliminated CIPs can be handled by a VM.

7.4.1 Considerations on CIP Placement Inside an Object and Reuse of CIP Location

The main benefit of CIP elimination from objects is memory space saving, so any reuse of the CIP location should be disabled to take advantage of CIP elimination. Among the VMs described in Section 7.2, only the Maxine VM reuses it, as a forwarding pointer is stored in the CIP location of an object during GC. The way to disable the reuse of the CIP location in the Maxine VM will be described in Section 7.6.1.

The other important factor for benefiting from CIP elimination is CIP placement inside the object and the memory management mechanism used in the VM. If free memory chunks are managed in the way of linked lists of fixed-size blocks and utilisation of memory blocks of the size of CIP is high, then the technique can be oblivious to CIP placement. However, if objects are allocated in thread-local allocation buffers by returning and post-incrementing a pointer to free memory space by the size of a recently allocated object[1], then a CIP should be placed on the boundary of the allocated block of memory for an object. Among the VMs described in Section 7.2, only the HotSpot VM does not meet this requirement. However, as the implementations of other VMs show, there are no fundamental restrictions on the CIP placement, and it can be placed at the beginning of the allocated block of memory for an object. In all other sections, without loss of generality, it is assumed that CIP is located at the beginning of the allocated block of memory for an object.

7.4.2 Encoding CIDs in Tagged Pointers

The number of CIDs that can be encoded in a tagged pointer depends on the number of bits dedicated for that purpose. The proposed technique can utilise a variable number of tag bits from 0 to n. If all tag bits are used for security or addressing purposes and no bits are left for encoding CIDs, a VM, implementing the proposed technique, does

[1]This technique is also known as a bump pointer allocation, and it is widely used in JVM implementations.

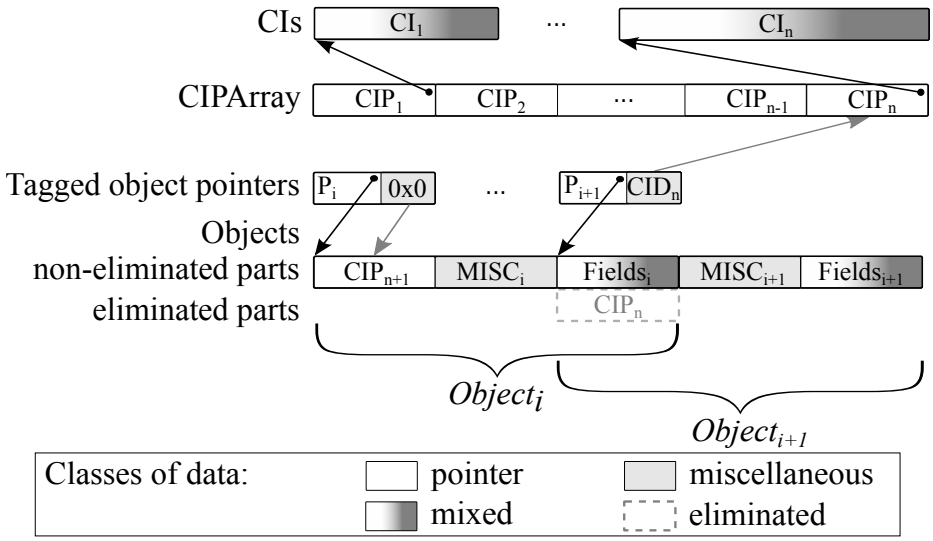


Figure 7.3: Scheme of encoding CIDs in tagged pointers and CIP elimination.

not have to be re-implemented. In other words, the proposed technique can be added to existing VMs without breaking backwards compatibility.

For n bits dedicated for storing CIDs in a tag, the range of CIDs is $[0; 2^n - 1]$ where `UNSPECIFIED_CID` $= 0$ represents any CI. When an object is allocated, the pointer is tagged by its respective CID. If the CID is not equal to `UNSPECIFIED_CID`, its CIP is not stored alongside the object (i.e. in the heap). Instead, the CID is directly encoded in the object pointer. The CI for specified CIDs can be stored directly in an array with a fixed element size, or alternatively, CIPs can be stored in the array. The second option is used because CIs have variable sizes while the cost of performing array modifications with CIPs is lower than in the first option.

The scheme of encoding CIDs in tagged pointers and enabling CIP elimination is depicted in Figure 7.3. There are two tagged object pointers in Figure 7.3: $\mathtt{P}_i$ and $\mathtt{P}_{i+1}$ pointing to $Object_i$ and $Object_{i+1}$, respectively. Pointer $\mathtt{P}_i$ has `UNSPECIFIED_CID` (0x0) in the tag indicating that $\mathtt{CIP}_{n+1}$ is stored in the object, while pointer $\mathtt{P}_{i+1}$ has $\mathtt{CID}_n$ in the tag indicating that $\mathtt{CIP}_n$ is stored in `CIPArray`, and so it is eliminated from $Object_{i+1}$. In this case, $\mathtt{P}_{i+1}$ points to the memory location where the beginning of the object would have been if $\mathtt{CIP}_n$ was not eliminated. Thus, the offsets to the same fields in objects of the same class (or superclass) with and without eliminated CIPs will stay the same.

7.4.3 CIPs Retrieval from Tagged Pointers

From the above description, the retrieval of a CIP from a given tagged object pointer (`objectAddress`) can be performed via the method `retrieveCIP` shown in Listing 7.1. Depending on the available bit extraction instructions of a given architecture and whether the available bits are contiguous, the method `extractCID` can result in a different number of instructions. For the rest of the chapter, it is assumed that CID bits are contiguous high-order bits of the tag, and tag bits are contiguous high-order bits of the tagged pointer. So, one shift instruction for the method `extractCID` can be used.

A check of whether the extracted CID is unspecified is performed. If `cid` is unspecified, CIP is loaded by an object address at a constant offset `CIP_OFFSET` in one instruction. Else, if `cid` is specified, the CIP is loaded from `CIPArray` with one instruction. In this work, `CIP_OFFSET` $= 0$ was used. On architectures without predication, two jumps (conditional and unconditional) will be emitted.

The code in Listing 7.1 is compiled to seven static instructions of 25 bytes size with an average dynamic execution height of 5.5 instructions on x86-64. By contrast, the JVMs described in Section 7.2 utilise just one instruction (load by `objectAddress` at offset `CIP_OFFSET`) of three bytes size on x86-64 to retrieve a CIP. Regarding storing a CIP in an object, the similar code snippet is used with the exception of omitting the store to `CIPArray` if `cid` is not unspecified.

```
// Array of Class Information Pointers          //  ! x86-64 GNU Assembler
CIP_t CIPArray[ 1 « CID_BITS_NUM ];             //
                                                //
// Retrieving CIP                               //  ! %rdi = objectAddress
CIP_t retrieveCIP( Address_t objectAddress) {   //  movq  %rdi, %rax
  CID_t cid = extractCID( objectAddress);       //  shrq  $48, %rax
  CIP_t res;                                    //  testw  %ax, %ax
  if ( cid == UNSPECIFIED_CID ) {               //  jne   .L1
    res = * ((CIP_t *)                          //  movq (%rdi), %rax
      (objectAddress + CIP_OFFSET));            //  jmp   .L2
  } else {                                      // .L1:
    res = CIPArray[ cid ];                      //  movq CIPArray(,%rax,8), %rax
  }                                             // .L2:
  return res;                                   //  ! res = %rax
}                                               //
```

Listing 7.1: CIP retrieval algorithm from tagged pointers.

7.4.4 Heap Traversal During Copying GC

When CIPs are eliminated from objects, certain changes to the copying GC schemes have to be applied. The problem is that traversal of copied objects in the copying GC schemes is performed via pointer arithmetic (untagged pointers), and CIPs in the objects are used to find the references inside the copied objects and their sizes. Thus, when CIPs are eliminated, the association of copied objects with their CIs should be maintained via an additional data structure. It is important to note that the proposed changes to the copying GC schemes do not require additional memory allocation.

The necessary changes are introduced in the context of the serial stop-the-world semi-space copying GC which employs Cheney's algorithm [Che70]. Modern generational GC algorithms employ a copying scheme based on Cheney's breadth-first copying GC scheme for frequent young generation collections, and the proposed changes can be applied to a wider spectrum of copying collectors (including parallel).

Cheney's Scheme Overview and Its Reliance on CI

Overview. In Cheney's scheme, objects are allocated in the "from-space", and upon a GC invocation, live objects are copied to the "to-space". Upon completion, the two spaces are swapped, and allocation continues in the "from-space". During a GC invocation, all threads stop at a safepoint where an initial set of live heap references can be retrieved. The initial set of live objects (typically includes objects whose references reside on the stack, in thread-local variables, etc.) is known as *root set*. When objects from the root set are copied to the "to-space", the GC threads start traversing the objects in the "to-space" in order to copy all the objects referenced by them. This iterative process is repeated until all live objects are copied from the "from-space". Essentially it is a breadth-first traversal of all live objects.

Reliance on CI. As the GC thread starts scanning the copied objects of the "to-space", it usually uses the CIs in order to find both the reference maps of the processed objects as well as their sizes in order to continue its traversal on subsequent objects. In the proposed CIP elimination scheme, the GC thread which traverses the "to-space" objects does not have CIPs in order to extract CIs during execution. Hence, a mechanism to maintain the mapping between copied objects and their CIs is required.

Proposed Changes

Overview. A scheme is proposed in which CIDs of the copied objects are stored in the

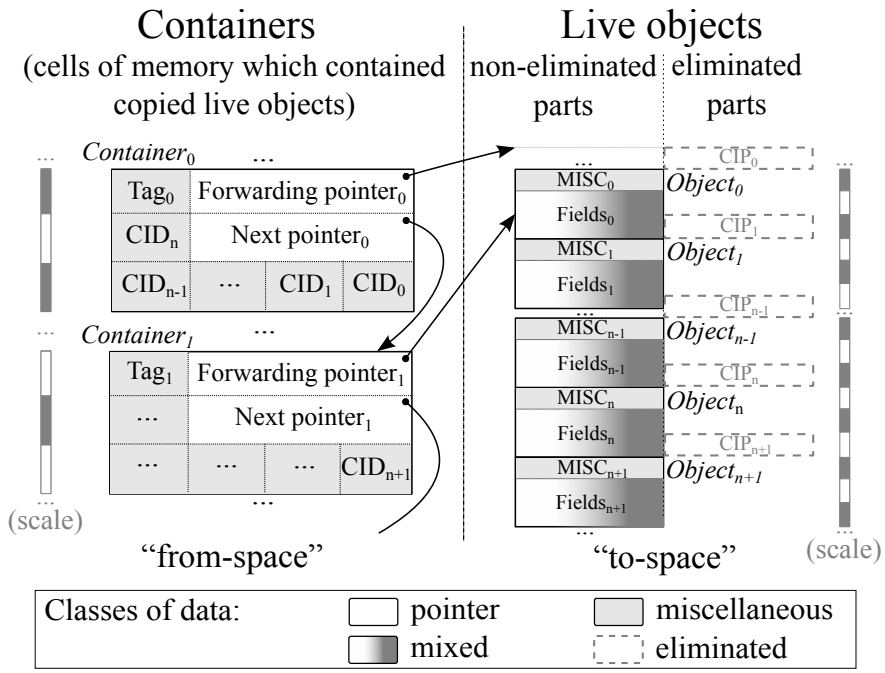


Figure 7.4: List of CIDs in the "from-space" during GC representing list of copied objects in "to-space".

"dead space" created upon the evacuation of live objects from the "from-space". The cells of memory in the "from-space" which contained copied live objects will be referred to as *containers*. The outline of the proposed scheme is depicted in Figure 7.4[2].

Copying objects from the "from-space". When objects are copied from the "from-space" to the "to-space", a tagged forwarding pointer is installed in the MISC words of the evacuated objects in the "from-space". The role of the forwarding pointer during GC is to indicate whether the object has been evacuated and where its new location is. Containers are used to store CIDs of evacuated objects until they are full (*e.g.* $[\mathtt{CID}_0;\mathtt{CID}_n]$ in $Container_0$), forming a singly linked list (*e.g.* $Container_0[\mathtt{CID}_0;\mathtt{CID}_n] \rightarrow Container_1[\mathtt{CID}_{n+1};...] \rightarrow ...$). Furthermore, the space from an evacuated object should meet *a minimum necessary container size requirement* to be used in the list: the space should be enough to accommodate a tagged forwarding pointer, an untagged pointer to the next container, and at least one CID.

Traversing objects in the "to-space". When GC roots are copied from the "from-space" to the "to-space", the GC thread starts traversing linearly the objects that have just been copied there. As described before, the pointers to these objects are calculated

[2]Scales of "from-space" and "to-space" are different, so the sizes of $Container_0$ and $Object_0$ are equal.

via pointer arithmetic during traversal and, therefore, the GC thread has to retrieve their CIs by reading the CIDs from the list of CIDs in the "from-space". In order to achieve that, the GC thread maintains a CID list iterator, which consists of an address to the current container and an offset inside it. When a GC thread traverses an object in the "to-space", it reads its CID using this iterator. By iterating the objects of the "to-space" and the list of CIDs synchronously currently traversed objects and their CIDs can be associated.

Traversing the list of CIDs in the "from-space". The containers are traversed with untagged pointers. Initially, when the first container ($Container_0$) is traversed, the offset of the iterator is reset to the size of the container minus the size of the CID. The size of the container can be found by using the CID extracted from $\texttt{Tag}_0$. After the initial offset is set to the offset of $\texttt{CID}_0$, the iterator will start traversing the CIDs backwards by decrementing the offset every time by CID size. When the CID list iterator points to $\texttt{CID}_{n-1}$, which is a constant offset from the beginning of a container, it is known that the point is reached where there is only one $\texttt{CID}_n$ left in the container. Therefore, when $Object_n$ is traversed, the iterator offset will be set to the offset of $\texttt{CID}_n$. Since the next container pointer $\texttt{Next pointer}_0$ is untagged, space is left to save $\texttt{CID}_n$. When $Object_{n+1}$ is traversed, the iterator pointer is set to $Container_1$, as well as the iterator offset is set to the offset of $\texttt{CID}_{n+1}$, and the process is repeated for that container.

Special cases. Regarding the first container, when GC is triggered, the free space from the "from-space" is used, if it is equal to or greater than the minimum necessary container size. In order to guarantee that there will be enough space to store the list of CIDs, the space needed conservatively for GC is accounted for during allocation. Small objects, not meeting the minimum necessary container size requirement, are counted during allocation, and their total number multiplied by the CID size is subtracted from the maximum heap space occupied before triggering a GC. Such small objects are quite rare in practice and do not reduce significantly the effective heap usage. For instance, in 64-bit VMs in which objects are 8-byte aligned and the size of the MISC word is 8 bytes (all the VMs described in Section 7.2 with the exception of Zing), only objects that contain just one MISC word do not meet the minimum necessary container size requirement (*e.g.* objects of class `java.lang.Object`).

To summarise, the benefits of the proposed scheme are: (1) re-use of existing "dead

space" requiring no off-heap memory allocation to save CIDs, (2) exploitation of temporal locality of cache lines related to dead objects, (3) low-overhead traversal, by employing a light-weight heuristic during object evacuation which utilises only large evacuated objects for containers.

7.5 Architectural Support

7.5.1 CIP Retrieval

In Section 7.4 the CIP retrieval from tagged pointers has been described (Listing 7.1). As shown in Section 7.7, the SW-only scheme of CIP retrieval from tagged pointers can lead to degradation of the execution time due to the extra latency introduced and code footprint increase. Novel architectural support to accelerate CIP retrieval is proposed to avoid performance regressions, as well as to minimise the modifications needed to adopt the CIP elimination scheme in existing VMs.

The scheme relies on the property that memory accesses to object fields are performed by using a two-or-more operand addressing mode (in the form of `[Base + Offset]`) [3]. Consequently, CIPs should always be accessed by addresses in the form of `[objectAddress + CIP_OFFSET]`. If more than two operands are used during addressing, one of them should represent the base address while the rest should represent the offset. This pattern, assuming it is preserved by the VM, can be identified by HW and handled accordingly depending on the CID value encoded in the tag.

The goal of the HW extension is to perform the CIP retrieval presented in Listing 7.1 simply by executing a single load instruction at an object address plus a constant offset `CIP_OFFSET`, as it happens in the VMs described in Section 7.2. The aforementioned address pattern can be identified by the *Address Generation Unit* (AGU) in the proposed AGU extension functionality presented in Figure 7.5. In this scheme, the functional block of an AGU of a given processor is presented, similarly to [MAKB03]. The input to the AGU is represented by two operands: `Base` and `Offset`. The extended AGU has an extra operand, *Class Information Pointer Array Address* (`CIPAA`), which is stored in a non-frequently changed control register. The control register has the same value for all AGUs, in case a core has several AGUs. The `CIPAA` control register holds the address of `CIPArray` and it is defined by the VM. Furthermore, it is required to be aligned to its size ($= 2^n$) in order to calculate the effective address of the retrieved CIP

[3] This addressing mode is supported by the majority of architectures, and this property has to be preserved by the VM.

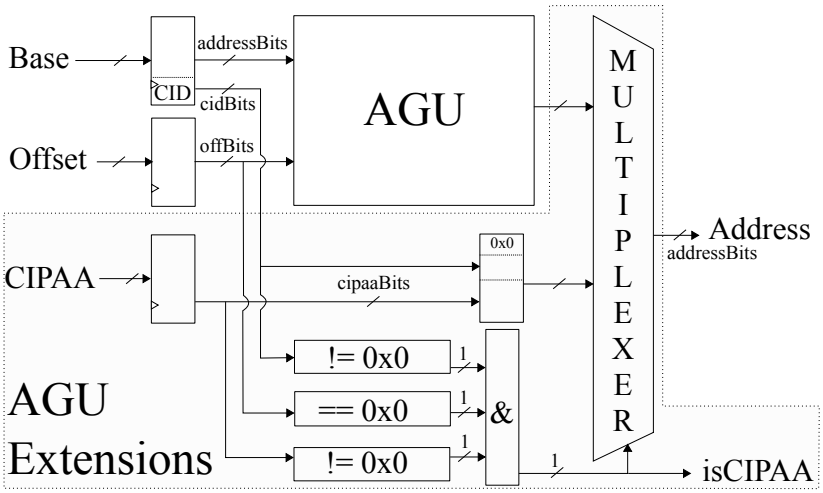


Figure 7.5: Extensions to Address Generation Unit (AGU) for CIPs retrieval.

location just by combining the `CID` at an offset shifted by log_2(`sizeof(CIP_t)`) bits and with zeros before this offset.

The output address depends on the *"is Class Information Pointer Array Access"* (`isCIPAA`) signal. Since both addresses from the AGU block and the extension (combiner) can be generated in parallel, the proposed scheme adds no significant delays to the address generation. The `CIPAA` is stored in a special purpose control register and can be read only by the AGU. When the `CIPAA` is zero, the extended address generation can be fully deactivated. When the `CID` is equal to `UNSPECIFIED_CID` or the `Offset` is different from `CIP_OFFSET`, then generation happens in the unextended way. Both `UNSPECIFIED_CID` and `CIP_OFFSET` are represented by all zeroes in the scheme for AGU extension simplicity, although they can be set by the control register as well.

7.5.2 Tagged Pointers Compression-Decompression

To reduce the memory footprint of objects, 64-bit VMs apply the object pointers compression optimisation [ATBC+04]. Depending on the heap size and object alignment, the possible values of object pointers can be represented by 32 or fewer bits, and in this case any 64-bit object pointer can be stored in a 32-bit location. An untagged object pointer compression and decompression can require shift and/or add operations or no extra operations depending on the heap size and base address of the heap. However, tagged pointers compression can require extra bit manipulation instructions to gather and scatter tag and non-tag object pointer bits. To avoid this overhead, the following *Load-Store Unit* (LSU) extension is proposed.

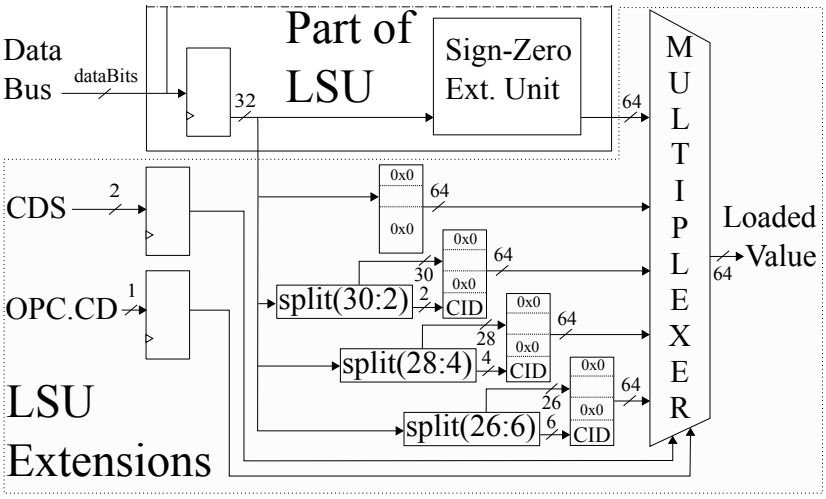


Figure 7.6: Extensions to Load-Store Unit (LSU) for object pointers decompression.

The goal of the proposed LSU extension is to perform scattering of 32 loaded bits of a compressed tagged object pointer during load operation to a 64-bit register and gathering them during reverse store operation in a limited number of ways efficiently. Figure 7.6 presents the necessary extensions to the LSU for load-decompress operation and the part of the LSU responsible for the sign-zero extension. The proposed LSU extension is activated by an opcode of the memory access instruction (`OPC.CD`) indicating that compression-decompression should be performed.

The extended LSU has an extra operand, *Compression-Decompression Selector* (`CDS`), which is a non-frequently changed control register. The `CDS` defines which bits are compressed-decompressed and it can have four states (s0-s3) depending on the heap sizes and used tag bits. In the scheme, these states are: (s0) 32GB and 0 bits, (s1) 8GB and 2 bits, (s2) 2GB and 4 bits, and (s3) 512MB and 6 bits. In the presented scheme, addresses of objects are required to be 8-byte aligned. State (s0) is needed to be able to reach the maximum heap size without code recompilation when compressed pointers are used. Before changing the `CDS` state to another state with fewer tag bits in use, objects, whose pointers are utilising the bits which will no longer fit into compressed pointers, must be handled to release them. This handling means restoring the eliminated CIPs of such objects and untagging their pointers. A larger number of states and different object alignments can be supported by extending the `CDS` and the scheme. The reverse store-compress operation is performed likewise but is not shown in the scheme for simplicity.

7.5.3 ISA Modifications

The control of the proposed AGU extension can be performed by the introduction and modification of a dedicated control register CR_n. If CR_n is set to zero, the extra functionality of the AGU will be switched off. The functionality is enabled when CR_n is set to a valid aligned `CIPAA`. When a tagged address in the form of `[CID:Base + CIP_OFFSET]` is passed to the extended AGU, the generated memory access will be [`CIPAA` | (`CID` $<< log_2($`sizeof(CIP_t)`$)$)]. If such an address is generated during a store operation, it will be executed as a NOP instruction. The `CIPAA` stored in CR_n should be maintained by the VM and new CIPs should be stored in `CIPArray`. If `CID` is unspecified or the offset is different from `CIP_OFFSET`, then the address generation will happen in the unextended way.

The control of the proposed LSU extension can be performed by the introduction and modification of a 2-bit dedicated control register CR_{n+1} to support 4 compression-decompression ways as described above. If the instruction set provides load-pointer and store-pointer instructions, LSU extensions can be activated by them and no extra instructions are necessary; for instance, PowerPC has such instructions to support *Technology Independent Machine Interface* [SN05]. However, if load-pointer and store-pointer instructions are not available, the extension can be activated by extra opcodes for memory-access operations:

```
ld32.cd Reg64, [ Base + Offset ]; // load-decompress
st32.cd [ Base + Offset ], Reg64; // store-compress
```

7.6 Experimental Platform and Methodology

As described in Section 7.3, architectures with tagged pointers support may have different tag sizes, cache hierarchies, and other parameters. Moreover, these architectures are not supported by state-of-the-art research JVMs. To make the study more general, and to utilise research JVMs while evaluating the proposed HW-assisted CIP elimination technique, it was decided to opt for a simulation-based approach.

7.6.1 MaxSim Platform

MaxSim platform, described in Chapter 6, is the experimental platform of choice since it provides high accuracy and fast simulation speed ($\approx$ 10 MIPS in the experiments

presented in this chapter). It was designed to run managed workloads and required minor modifications to research CIP elimination from objects on architectures with tagged pointers support. Alternative options, such as the Sniper [CHE11, CHE+14] simulator that runs with JikesRVM, or the full-system gem5 [BBB+11] simulator, were considered but abandoned due to a number of limitations. Regarding the Jikes RVM running on top of the Sniper simulator [Jik14, SHB+14], it is only possible to run it in a 32-bit mode, while gem5 has a relatively low simulation speed.

ZSim Suitability

ZSim, being a simulator in the MaxSim platform, has a number of declared idealisations and limitations, among which are: not fully supported prefetchers, an ideal branch target buffer, not supported loop stream detector, micro-sequenced instructions and other components of the microarchitecture (described in [SK13]). With the exception of prefetchers, these limitations and idealisations do not directly affect the evaluation of the proposed CIP elimination technique. However as shown in [CMB+13], the impact of the not modelled HW prefetchers on the performance of the DaCapo benchmarks, which are used in this work, is insignificant.

ZSim Extensions

When memory accesses with tagged addresses are simulated by ZSim, they are untagged and executed in an ordinary way during Pin [LCM+05] translation. However, the tags are delegated to the core simulation model making it possible to simulate tagged pointers and evaluate the proposed AGU extensions. Setting the CR_n is performed via a magic NOP instruction. The proposed AGU extensions are functionally modelled, as described in Section 7.5, assuming there is no extra latency overhead introduced by them.

Maxine VM, being a JVM in the MaxSim platform, does not support compressed object pointers, so the address space morphing technique, described in the previous chapter, was used to model them. The following bijections in the address space morphing technique are used to model compressed object pointers: $f_e(2,2)$, $f_c(2)$, $f_r(\varnothing)$. With this technique, no other changes in the simulator were made to simulate LSU extensions, as it is assumed that: (1) load-decompress and store-compress operations do not introduce extra overhead in comparison to ordinary load and store operations; (2) introducing two extra opcodes does not significantly change the code footprint; (3) Compression-Decompression Selector changes infrequently.

Name		4C	4CA	4CAL	1C	1CA	1CAL
Cores	type	x86-64 Nehalem OOO core at 2.66 GHz					
	total	**4**			**1**		
AGU Ext.		-	+	+	-	+	+
LSU Ext.		-	-	+	-	-	+
L1I caches		32KB, 4-way, LRU, 3-cycle latency					
L1D caches		32KB, 8-way, LRU, 4-cycle latency					
L2 caches		256KB, 8-way, LRU, 6-cycle latency					
L3 cache	type	16-way, hashed, 30-cycle latency					
	size	8MB			2MB		
Memory controller		1, 3 DDR3 channels, 47-cycle latency					
DRAM		3GB, DDR3-1066, 1GB DIMM per channel					

Table 7.2: ZSim configurations.

The power and energy estimation model using McPAT is performed as described in the previous chapter. The proposed extensions to the AGU and the LSU were not added in the power estimation model as the energy overhead of these functional units extensions is significantly less than the energy savings from the reduction of memory traffic to DRAM and *Last-Level Cache* (LLC).

ZSim Configurations

Table 7.2 details the seven hardware configurations used for the evaluation. The latencies associated with the levels of the memory hierarchy in the table do not include latencies associated with the lower levels, so, for instance, L2 data hit will be $(4+6=)$ 10 cycles. The memory controller latency is in core clocks.

Configuration `4C` models a 4-core Intel Nehalem CPU. Configuration `1C` models a 1-core Intel Nehalem CPU with a quarter of the available LLC. Configurations `4CA` and `1CA` represent the proposed extensions to the AGU of configurations `4C` and `1C` respectively. Configurations `4CAL` and `1CAL` represent the proposed extensions to both the AGU and the LSU of configurations `4C` and `1C` respectively. Configurations `1C`, `1CA` and `1CAL` were selected to simulate the scenario when only a quarter of the available resources is available to the workload (if LLC could be partitioned, adding pressure to the caches). Finally, all configurations support 16-bit pointer tags.

Maxine VM Suitability

The main direct effect of the proposed CIP elimination technique is the reduction of

the memory footprint of the objects. In all VMs supporting the same object layout, the memory space savings related to the reduced memory footprint of objects are expected to be the same as in Maxine VM.

In the experiments presented in this chapter, a non-generational semi-space heap scheme of a constant 2GB size with a stop-the-world copying GC is used, which is described in Section 7.4. This is the default and stable GC scheme in the Maxine VM. In the context of the single-core configurations that is used, this scheme can be optimal for throughput. Finally, modern generational GC algorithms employ a copying scheme based on Cheney's breadth-first copying GC scheme for frequent young generation collections, and the proposed changes can be applied to a wider spectrum of copying collectors.

Maxine VM Extensions

A tagging scheme is used which assigns CIDs to tags of object pointers in the VM, which is described in the previous chapter. CIP elimination from object headers in the heap was implemented by modifying object memory allocation and GC as discussed in Section 7.4.

Originally, during GC, a forwarding pointer was stored in the CIP word of an object in Maxine VM. Maxine VM was modified to store the forwarding pointers in the MISC words instead of the eliminated CIP words. Furthermore, bit 47 of the MISC word is reserved for forwarding pointer indication (when it is set to one), constraining the range of heap addresses to have bit 47 set to zero. The decision to reserve bit 47 of the MISC word has been taken after considering the least collateral effect to the VM functionality. In this case, the result is a $2\times$ reduction of the possible concurrent threads from 2^{16} to 2^{15} used in thin locking [BKMS98].

Originally, in Maxine VM, a null pointer check is performed by loading the value of the CIP (object pointer plus offset zero). If the pointer is null, an exception will be raised, otherwise the CIP would be loaded. If the CIP is stored alongside the object, a null check can have a positive prefetching effect. In a case of CIP elimination from object headers, CIPs would be loaded from `CIPArray`. It was decided to change the offset of the null pointer check to eight, which points to the MISC word, which is the first word in the object when CIP is eliminated. Thus, positive prefetching effects of null checks are preserved.

Name	Description
`B`	**B**aseline Maxine VM.
`E`	`B` with CIP **E**limination for ZSim configurations without extra HW support.
`EA`	`B` with CIP **E**limination using 16-bit CIDs in tagged pointers for ZSim configurations with **A**GU extensions.
`C`	`B` with **C**ompressed object pointers.
`EAL`	`C` with CIP **E**limination using static profile and 4-bit CIDs in tagged pointers for ZSim configurations with **A**GU and **L**SU extensions.

Table 7.3: Maxine VM configurations.

Maxine VM Configurations

The experiments were carried out with five Maxine VM configurations described in Table 7.3. Later in the chapter, pairs of ZSim and Maxine VM configurations will be used in order to evaluate the combinations of different configurations (*e.g.* `4C-B`).

As a constant 2GB heap size is used, Compression-Decompression Selector is in state (s2), when compressed object pointers optimisation is enabled. In this state, a loaded 32-bit compressed pointer value is split into 28 and 4 bits, the former shifted by 3 bits and the latter placed in the high-order bits of the 64-bit register as CID.

7.6.2 Benchmarks

Two widely acknowledged and two emerging benchmarks were used in this study:

- **DaCapo-9.12-bach** [BGH+06] suite covers a representative number of server and desktop applications.

- **pjbb2005** [PJB05] is a version of the SPECjbb2005 [SPE05] benchmark with a fixed workload.

- **GraphChi-PR** [KBG12] is a PageRank algorithm [PBMW98] running on top of the disk-based system for graph analytics GraphChi.

- **SLAMBench** [NBZ+15] is a Java version of computer vision benchmark for simultaneous localisation and mapping which implements the KinectFusion algorithm [NIH+11].

7.6.3 Experimental Methodology

In the experiments presented in this chapter, a constant heap size of 2GB was used, where only 1GB is used in the semi-space scheme. Although VMs can support variable heap sizes, the variable heap size will only directly affect the amount of GC and heap resize work. Separate evaluations of execution times of GC were done, as one of the positive effects of the proposed technique is the reduction in the amount of GC work to be performed. Thus, the presented data allow making analytical estimations of the proposed technique for other heap sizes.

Each experiment has been run 10 times, and whiskers represent 95% confidence intervals. The variance for some of the tests can be up to 5% due to the dynamic nature of the VM or related to nondeterminism aspects in GC, JIT compilation, threads scheduling, and other factors. ZSim is a DBT-based execution-driven simulator, so the simulation is not deterministic. Assuming normal distribution, when the whisker crosses zero in a chart showing relative changes, there is less than 95% confidence whether a result is positive or negative.

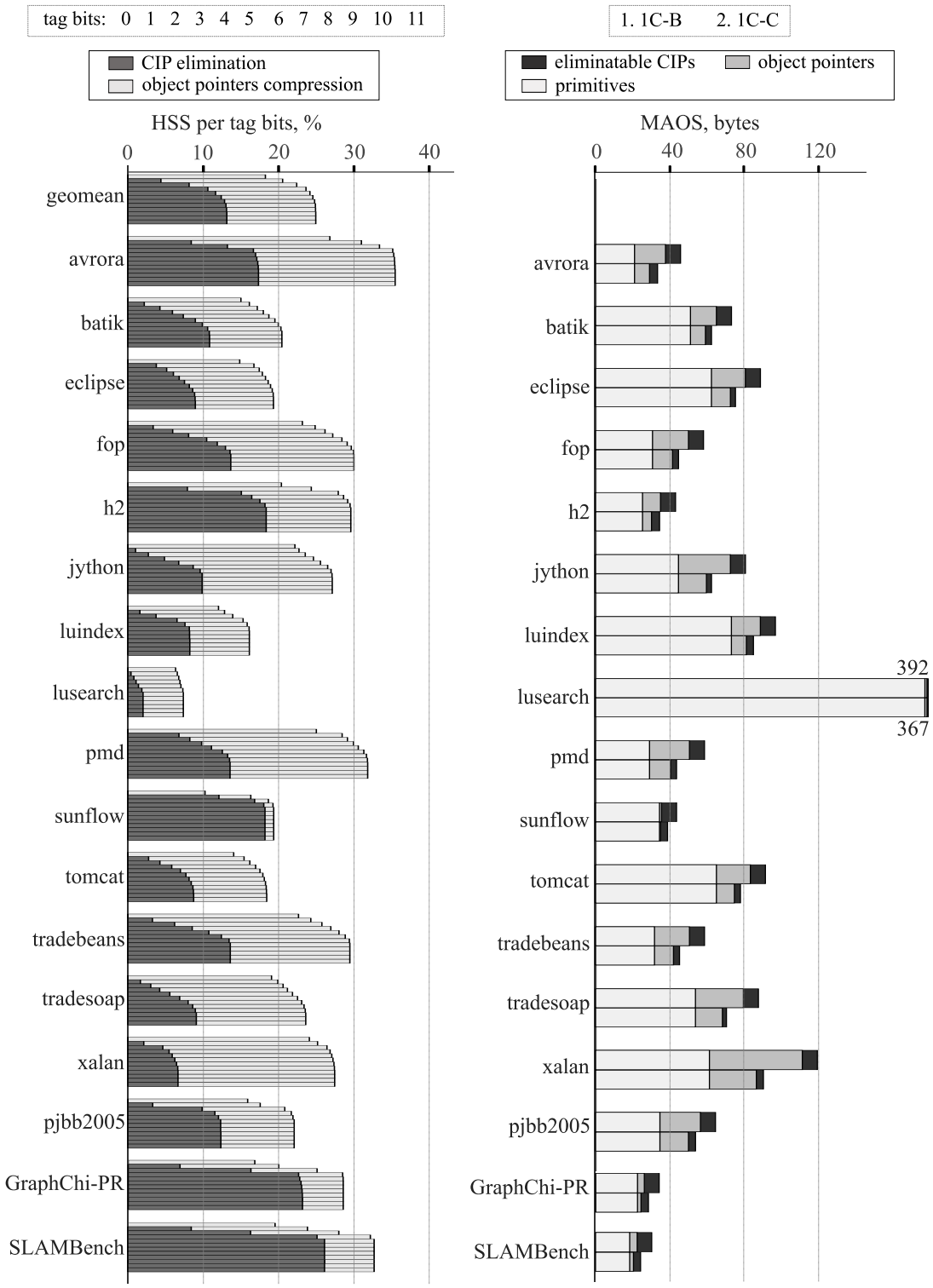
7.7 Experimental Results

7.7.1 Heap Space Savings

The main benefit of CIP elimination from object headers is heap space savings. Since the number of tag bits can vary on different architectures, and some of the bits can be utilised for other purposes, it is important to explore how much heap space savings can be achieved per available tag bits for CIDs storage. *Heap Space Savings* (HSS) per available tag bits are estimated on the two baseline configurations, `1C-B` and `1C-C`, by collecting profiling information on the number of all *Allocated Objects* (AO) for each class $AO(c)$, where c is class, and on the total allocated *Heap Space Volume* (HSV). $AO(c)$ is sorted in decreasing order and the *Sorted Allocated Objects* $SAO(i)$ sequence is obtained. Finally, it is estimated how much $HSS_{\mathtt{C}}(n)$ can be gained for configuration `C` using n tag bits available for CIDs storage by using the following formula:

$$HSS_{\mathtt{C}}(n) = \frac{HSV_{\mathtt{C}} - \mathtt{sizeof(CIP\_t)} * \sum_{i=0}^{2^n-2} SAO_{\mathtt{C}}(i)}{HSV_{\mathtt{1C-B}}} * 100\%.$$

In this estimation, it is assumed that the $AO(c)$ distribution can be perfectly predicted dynamically or it is known statically from previous runs. Estimated heap space savings



(a) Estimation of Heap Space Savings (HSS) per available tag bits for CIDs storage

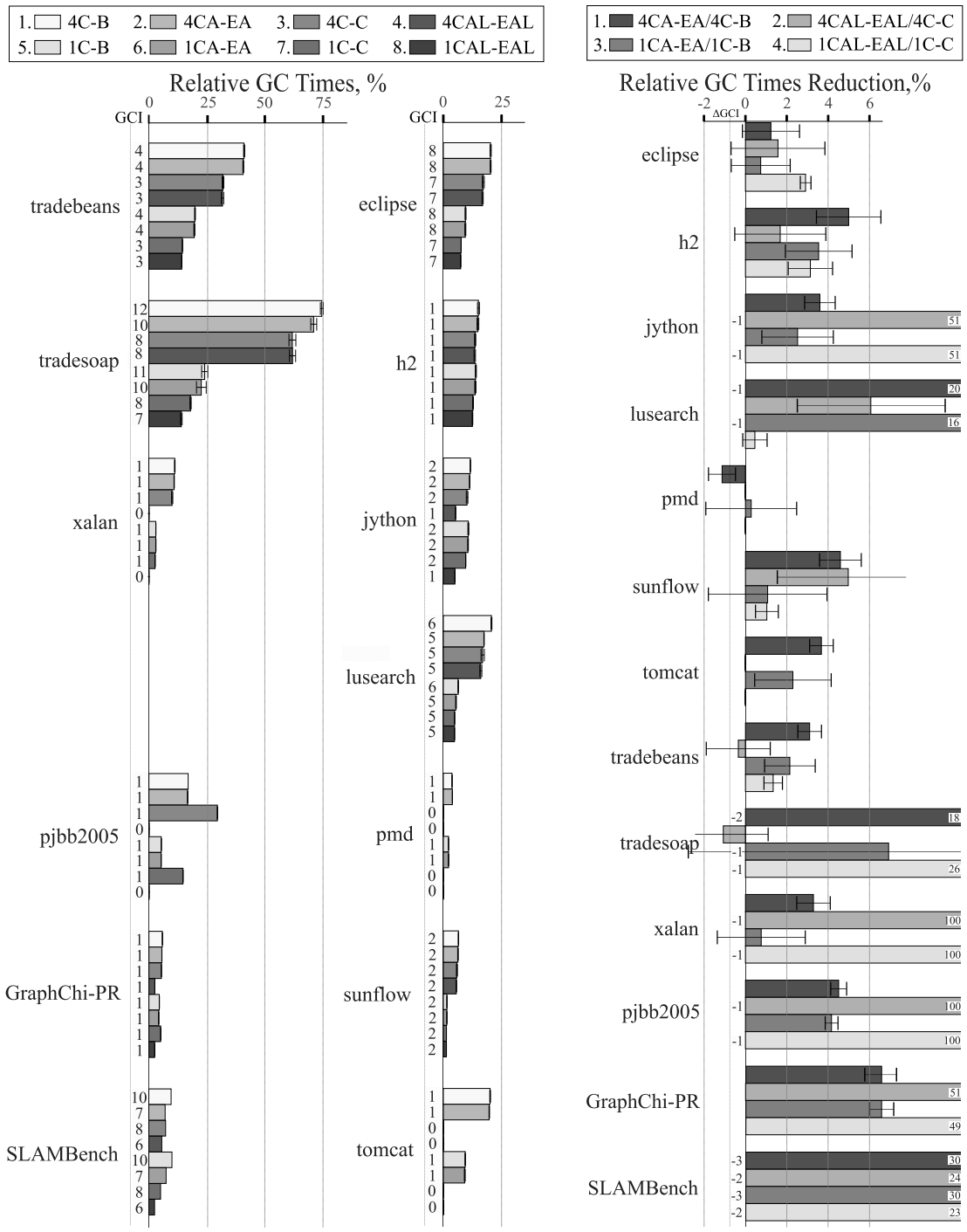
(b) Mean Allocated Object Size (MAOS)

Figure 7.7: Estimation of heap space savings and mean allocated object size for different configurations.

per tag bits for 1C-B (bottom dark grey bar segments) and 1C-C (full stacked bars) configurations are presented in Figure 7.7a. As depicted in the figure, the number of bits for CIP elimination required to reach heap space savings close to maximum can vary for different workloads, and 8 bits are sufficient to gain 99% of possible heap space savings on selected benchmarks, while 11 bits are enough to gain 100%. These savings can lead to a proportional reduction in GC times and cache misses due to a reduced memory footprint. Heap space savings inversely correlate with *mean allocated object size* as can be seen in Figure 7.7b. Mean allocated object size is shown for two configurations: 1C-B (left full stacked bar) and 1C-C (right full stacked bar). Thus, the bottom segments represent mean sizes of the primitive parts of the objects, the two upper segments represent mean sizes of the pointers in the objects, and the uppermost segments represent mean sizes of the CIPs that can be eliminated from the objects with the technique proposed in this chapter. On configurations with CIP elimination and proposed HW extensions, 1CA-EA and 1CAL-EAL, it is possible to achieve up to 26% (SLAMBench) and 10% geometric mean heap space savings. It can be also observed in Figure 7.7a that the effect of object pointers compression on heap space savings (the first bar for each test) is more significant than the effect of CIP elimination (the last dark grey bottom segment for each test) for all tests with the exception of sunflow, GraphChi-PR, and SLAMBench.

7.7.2 Effects of CIP Elimination on GC

First, it is examined how much time is spent in GC invocations relative to the execution time for configurations with and without CIP elimination. Relative GC times to execution times are presented in Figure 7.8a. Since GC never happens on avrora, batik, fop, and luindex on the tested configurations, the results for these tests are not shown in the figure. When no space is left in the "from-space", a *Garbage Collection Invocation* (GCI) occurs that copies live objects to the "to-space". The number of GCIs during execution of each benchmark for each configuration is shown left of each bar. Configurations with (*-E*) and without (*-B, *-C) CIP elimination are depicted adjacently. The evaluation is done pairwise 4CA-EA against 4C-B, 4CAL-EAL against 4C-C, etc. The following notation will be used hereafter: 4CA-EA/4C-B means 4CA-EA compared against 4C-B. It can be seen that CIP elimination leads to reduction in the number of GC invocations and relative GC times to execution times for jython (*CAL-EAL/*C-C), lusearch (*CA-EA/*C-B), tradesoap (*CA-EA/*C-B, 1CAL-EAL/1C-C), xalan (*CAL-EAL/*C-C), pjbb2005 (*CAL-EAL/*C-C), SLAMBench (*CA-EA/*C-B, *CAL-EAL/*C-C).



(a) Relative Garbage Collection Times (GCT) to Execution Times (ET) and numbers of Garbage Collection Invocations (GCI) (left of the bars)

(b) Relative reductions in Garbage Collection Times (GCT)

Figure 7.8: Changes in GC times and numbers of GC invocations for various configurations.

For all subsequent figures, the deltas in garbage collection invocations for compared configurations (△GCI) not equal to zeros will be shown left of the bars.

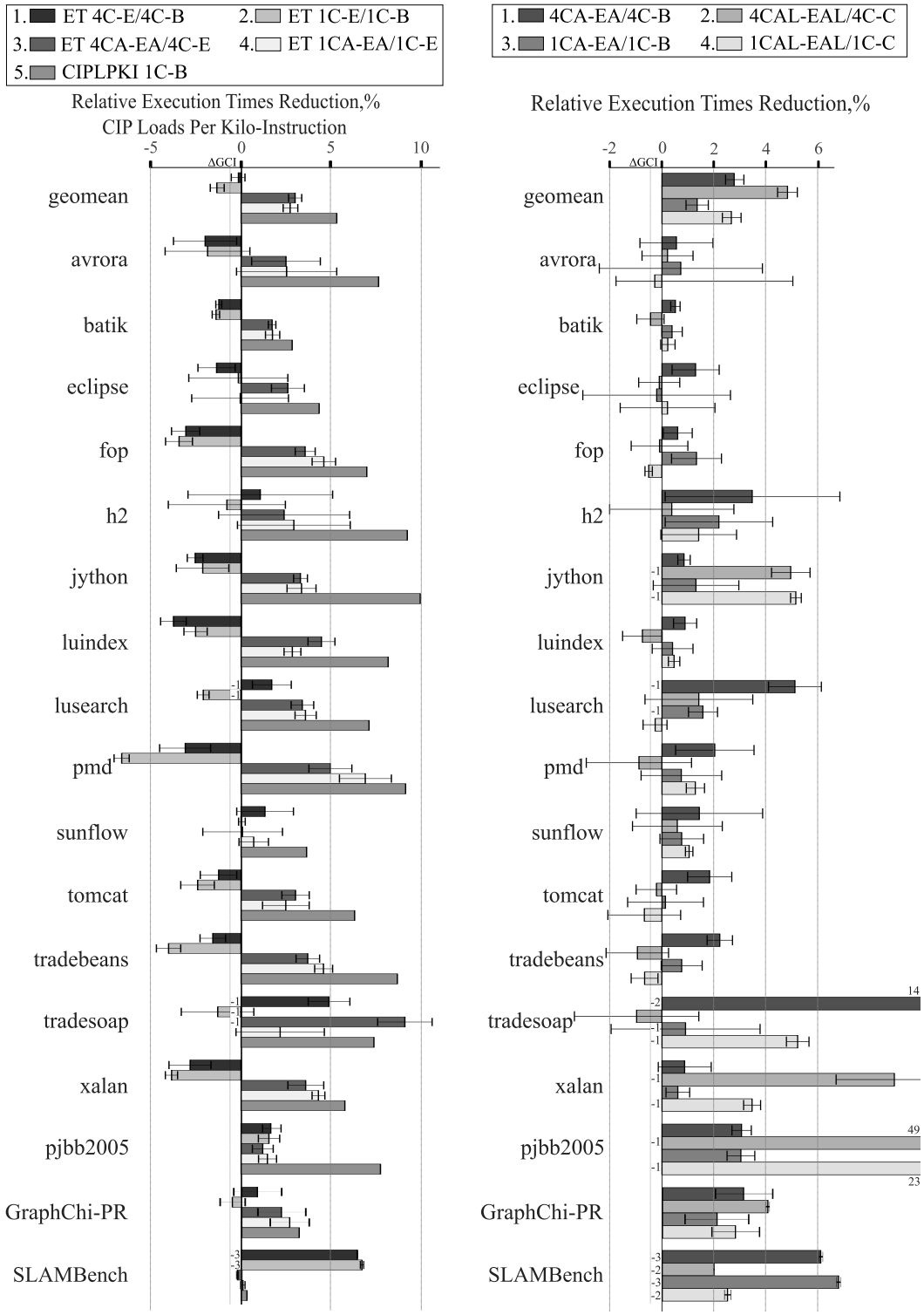
Secondly, it is investigated how GC time is affected as a result of CIP elimination. The relative reductions in GC times are presented in Figure 7.8b. When the number of garbage collection invocations is reduced as a result of CIPs elimination, the GC times reductions are above 16% (lusearch 1CA-EA/1C-B). When △GCI is zero, the GC times are reduced for the majority of tests with no increases above 3% (tradesoap 4CAL-EAL/4C-C). In two cases (GraphChi-PR *CAL-EAL/*C-C), the reduction in GC time is approximately 50%, which is explained by the high dynamism in the number of live objects, as was shown by Nguyen *et al.* in Figure 2 [NFX+16].

The presented data provide evidence that the maintenance and traversal of the list of CIDs during copying GC (described in Section 7.4) do not introduce significant overhead and do not overweigh the performance gains. Provided that the set of live objects is the same during copying GC, the gains are due to less memory footprint of the copied objects.

7.7.3 Effect of CIP Elimination on Execution Time for Configurations Without HW Extensions

In this experiment, the effect of the SW-only CIP elimination, without the proposed HW extensions, on execution time is investigated. The following configurations are evaluated: 4C-E/4C-B, and 1C-E/1C-B. 0% and 1% geometric mean execution time degradations are observed for the aforementioned configurations respectively, which are shown in Figure 7.9a (the first two series). Execution time reductions when △GCI is zero are observed for pjbb2005 and SLAMBench. On pjbb2005 reductions are 1.7% for 4C-E/4C-B and 1.6% for 1C-E/1C-B, and on SLAMBench reductions are 6.5% for 4C-E/4C-B and 6.8% for 1C-E/1C-B. These observations correspond with the fact, that pjbb2005 was classified as "cache-miss-intensive", while DaCapo-9.12 as "non-cache-miss-intensive" by Inoue and Nakatani [IN12]. The performance degradation for DaCapo-9.12 is observed because the performance overhead during the SW-only CIP retrieval from tagged pointers is not covered by the gains due to cache miss reductions.

This hypothesis is validated by measuring the percentage of *CIP Loads Per Kilo-Instruction* (CIPLPKI) on the 1C-B configuration, which is shown as the fifth (the last) series in Figure 7.9a with a geometric mean value of 5.4 CIPLPKI. As discussed



(a) Relative reductions in Execution Times (ET) for various configurations and CIP Loads Per Kilo-Instr. (CIPLPKI) for 1C-B

(b) Relative reductions in Execution Times (ET) for various configurations

Figure 7.9: Relative changes in Execution Times (ET) for various configurations.

in Section 7.4, SW-only CIP retrieval from tagged pointers has the dynamic execution height of 5.5 instructions in the proposed implementation. Thus, the estimated overhead of CIP access on the `1C-E` configuration will be 24.3 extra instructions per kilo-instruction. This estimation is tested by evaluating configurations with enabled CIP elimination with the proposed AGU extensions against the SW-only CIP elimination: `4CA-EA`/`4C-E`, and `1CA-EA`/`1C-E`, which are the third and the fourth series in Figure 7.9a. The relative geometric mean execution time reductions for these configurations are 3.1% and 2.8% respectively, which are consistent with the estimation and motivate the usage of the AGU extension.

The data for the SW-only CIP elimination for configurations with compressed pointers is not presented in this chapter. The reason behind that is the high estimated performance overhead, since loads of compressed object pointers happen approximately 6 times more frequently than CIP loads in geometric mean for `1C-C` configuration, with the geometric mean rate of 32.4 loads of compressed object pointers per kilo-instruction. A few modern architectures, like Intel Haswell and AMD Excavator, provide advanced instructions for bit manipulation, such as PDEP and PEXT [Int11], that can be used for tagged pointers compression and decompression in one instruction. However, these instructions have a high latency of 3 cycles [Fog16]. It is estimated that the overhead of compressed pointers decompression using these instructions is 32.4 extra instructions and 97.2 extra execution cycles per kilo-instruction. This significant overhead motivates the usage of the LSU extension.

7.7.4 Effect of CIP Elimination on Execution Time for Configurations with HW Extensions

In this experiment, the effect of CIP elimination is investigated for configurations with the proposed AGU and LSU extensions. The following configurations are compared: `4CA-EA`/`4C-B`, `4CAL-EAL`/`4C-C`, `1CA-EA`/`1C-B`, and `1CAL-EAL`/`1C-C`. The relative reductions in execution time for these configurations are shown in Figure 7.9b, with geometric mean values of 2.9%, 5.0%, 1.4%, and 2.8% respectively. The maximum reductions are 13.6%, 49.1%, 6.9%, and 22.6% respectively, with no degradations on single tests below 4%.

7.7.5 Reduction in Cache Misses

When the CIPs are eliminated from the object headers, they are densely located in `CIPArray`, and the memory footprint of the allocated objects in the heap is smaller. Both factors lead to a decrease in cache misses. This hypothesis is tested by comparing the following configurations: `4CA-EA`/`4C-B`, `4CAL-EAL`/`4C-C`, `1CA-EA`/`1C-B`, and `1CAL-EAL`/`1C-C`. Significant relative reductions in *L3 Cache Misses Per Kilo-Instruction* (L3CMPKI) were observed which are shown in Figure 7.10b, with geometric mean values of 12.7%, 10.0%, 12.4%, and 9.2% respectively. These values are correlated with the heap space savings estimations presented in Figure 7.7a, which are 13.3% for `1C-B`(11 bits)/`1C-B`(0 bits) and 7.2% for `1C-C`(4 bits)/`1C-C`(0 bits).

Relative reductions in *L2 Cache Misses Per Kilo-Instruction* (L2CMPKI) are more moderate compared to L3CMPKI. As shown in Figure 7.10a, the geometric mean values are 8.2%, 4.4%, 8.5%, and 5.8% for respective configurations. The relative increase in L2CMPKI in `pjbb2005` for `4CAL-EAL`/`4C-C` and `1CAL-EAL`/`1C-C` configurations is related to the complete elimination of GC invocations on configurations with CIP elimination, `4CAL-EAL` and `1CAL-EAL`. Furthermore, GC can have different cache miss characteristics from the workload and can have a positive effect on the locality of copied objects.

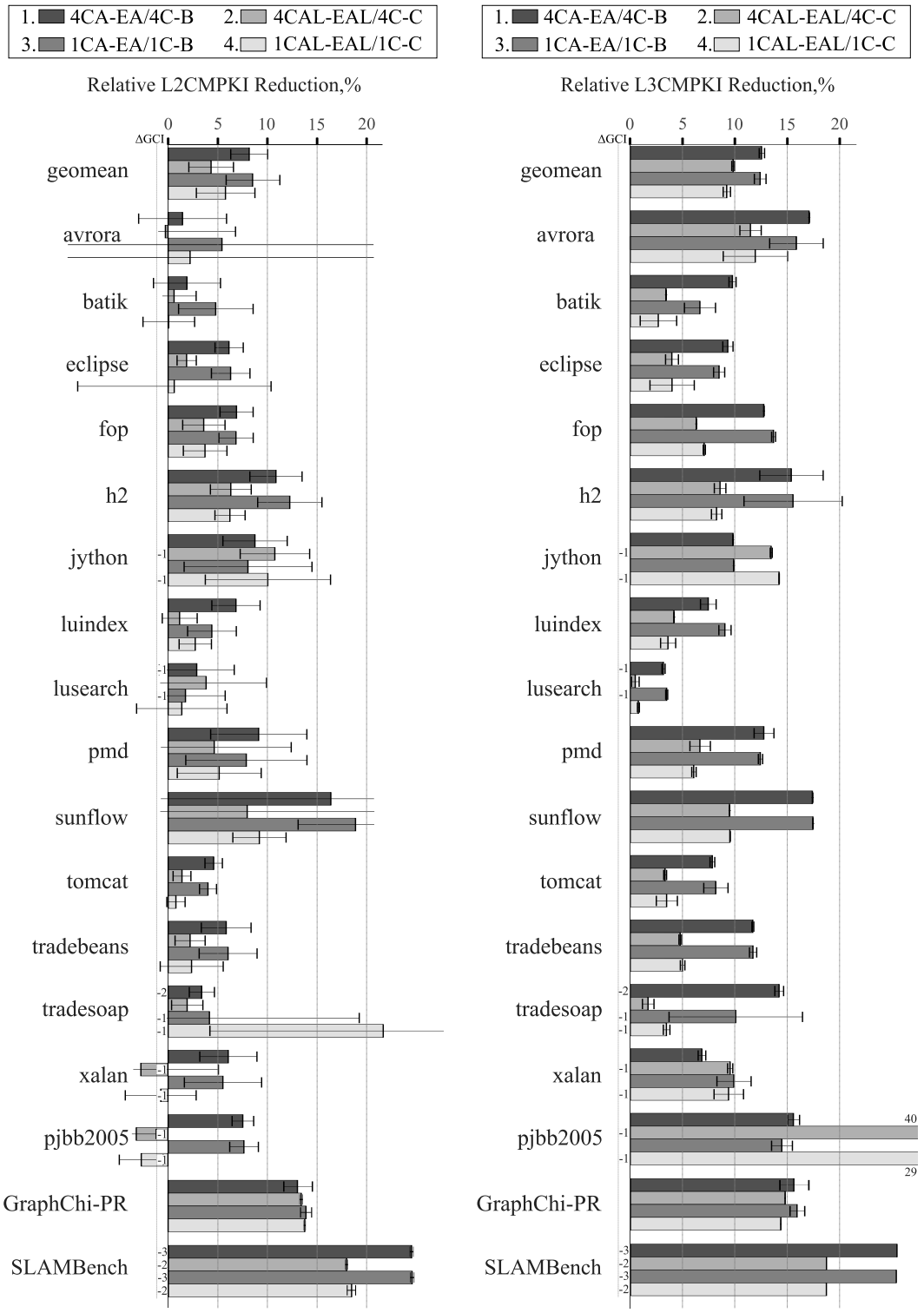
7.7.6 Reduction in Dynamic Energy

Reduction in cache misses leads to less DRAM and L3 cache traffic which is consequently translated to *Dynamic Energy* (DE) reductions in these components. This hypothesis is tested on the same configurations as before. The relative reductions in *DRAM Dynamic Energy* (DRAMDE) are shown in Figure 7.11b. The geometric mean values of 12.9%, 11.1%, 12.3%, and 10.6% are correlated with L3CMPKI reductions for the respective configurations. Maximum relative DRAMDE reductions reach up to 27.6%, 50.1%, 27.0%, 41.5% respectively.

Finally, relative reductions in *L3 Dynamic Energy* (L3DE) are shown in Figure 7.11a, with geometric mean values of 8.6%, 5.3%, 9.3%, and 7.6% respectively.

7.8 Related Work

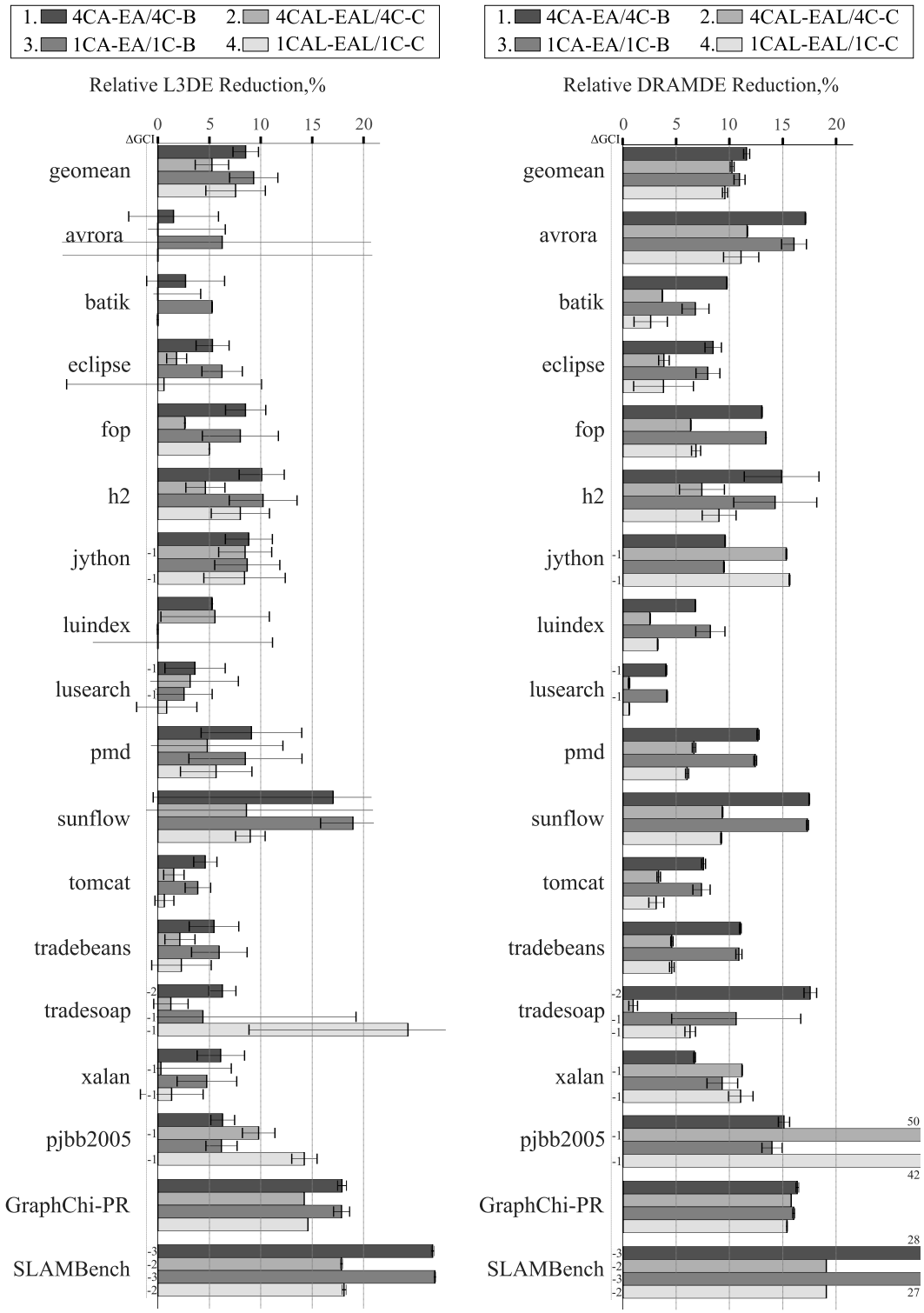
From a historical perspective, Steele [Ste77] proposed using contiguous memory regions for LISP, so that only variables of a certain data type can be in a given region.



(a) Relative reductions in L2 Cache Misses Per Kilo-Instruction (L2CMPKI) for various configurations

(b) Relative reductions in L3 Cache Misses Per Kilo-Instruction (L3CMPKI) for various configurations

Figure 7.10: Relative reductions in cache misses per kilo-instruction for various configurations.



(a) Relative reductions in L3 Dynamic Energy (L3DE) for various configurations

(b) Relative reductions in DRAM Dynamic Energy (DRAMDE) for various configurations

Figure 7.11: Relative reductions in dynamic energy for various configurations.

Using segmentation and enforcing data structures alignment, Dybvig *et al.* [DEB94] investigated a hybrid technique utilising also the least significant bits of the address for implicit typing for the Scheme language. Continuing taking advantage of the contiguous virtual addresses for implicit typing, Bacon *et al.* [BFG02] used this technique for compression of object headers in the context of 32-bit Jikes RVM. They gained most of the space savings by eliminating the thin lock word. The implicit typing scheme via virtual addressing was later explored in the context of the 64-bit Jikes RVM on PowerPC by Venstermans *et al.* [VEDB07a] to remove the object header completely.

In contrast to these related systems, the presented approach using object typing via tagged pointers does not bind objects to contiguous memory regions. Thus, the proposed technique can facilitate additional optimisations such as data transformations and object fusing [WM10] and objects alignment and collocation as proposed by Inoue and Nakatani [IN12].

Using tags for storing type information has been used in many computer systems [Org73, Bab00, HSH81]. However, the main use case of tagged pointers is capability-based security which also requires tagged memory. A generalised hardware support for tag processing has been recently proposed by Dhawan *et al.* [DHR+15], which can be extended for performance purposes by the HW extensions introduced in this work. The other use case of tagged memory is the association of application-specific metadata with specific memory locations. Hardware support for tag processing related to software transactional memory was proposed by Stipic *et al.* [STZ+12].

7.9 Conclusions

In this chapter, the MaxSim platform for HW/SW co-design research, introduced in the previous chapter, has been demonstrated. By using this platform, a proposed technique of CIP elimination by encoding CIDs in tagged pointers has been thoroughly evaluated. Although this technique significantly reduces memory footprint, it has been shown that retrieving CIPs from object pointer tags without extra HW support can degrade performance. To address performance issues, novel hardware extensions have been proposed. These extensions are related to the AGU removing the performance degradations associated with CIP retrieval from tagged pointers and to the LSU for efficient load-decompression and store-compression of tagged object pointers.

In addition, the experimental results for the proposed HW/SW co-designed technique achieve significant heap space savings, dynamic energy reductions, and performance improvements without significant regressions. Although the proposed technique has been researched in the context of a research JVM implementation, it can be widely applicable to other object-oriented languages and managed runtimes.

Chapter 8

Conclusions and Future Work

8.1 Summary and Conclusions

As discussed in Chapter 1, progress in microprocessors has moved towards increasing the number of cores, heterogeneity and bitness of computing. Performance, programmability and energy efficiency of next generations of microprocessors will be highly dependent on efficient synchronisation, hardware virtualisation, memory subsystem utilisation and closer synergy between hardware and software. Multidisciplinary work presented in this thesis addressed these challenges by advancing the state-of-the-art in the following three major fields of computer science: shared-memory synchronisation, computer architecture simulation, and high-level language computer architecture. Chapters 2, 4, and 5 provided the background in these areas for the three chapters presenting the contributions of this work. The contributions of this work to the state-of-the-art are summarised in the subsequent three paragraphs.

Chapter 3 explored specialisation of the five state-of-the-art barrier synchronisation algorithms for the Intel Xeon Phi coprocessor. A novel hybrid specialised variant was presented based on different algorithms to synchronise at intra-core and inter-core levels. Comparing the hybrid algorithm with previously proposed algorithms, lower overheads have been observed in the experiments on EPCC barrier microbenchmark, and an improved performance has been observed on direct N-body simulation kernel and two NAS Parallel Benchmarks, CG and MG. In other words, the fastest known barrier implementation for Intel Xeon Phi was presented, achieving a 3.28× lower overhead than the Intel OpenMP barrier implementation (ICC 14.0.0). These optimised barriers are available at `https://github.com/arodchen/cbarriers` released as free software. In addition, the analysis of the impact of the ring interconnect and distributed

tag directories of the Xeon Phi system on barrier synchronisation has been provided. It was found that the inability to have explicit control over tag directories leads to missed optimisation opportunities when specialising SW for the Intel Xeon Phi coprocessor.

Chapter 6 presented the MaxSim platform. It is a novel and open-source experimental platform for HW/SW co-design research and characterisation of managed workloads. MaxSim is based on the state-of-the-art Maxine VM, the ZSim microarchitectural simulator, and the McPAT power, area, and timing modelling framework. MaxSim features the simulation of 16-bit-tagged pointers, which can be utilised for: (1) low-intrusive memory access profiling, (2) tagged pointers modelling on x86-64 architectures, and (3) experimenting with novel HW/SW co-designed optimisations by extending the semantics of memory access operations via pointer tagging. In addition, the address-space morphing technique was presented. This technique allows modelling of complex software changes, such as compressed object pointers optimisation and other data layout transformations. MaxSim's capabilities were showcased by: (1) performing an up-to-date microarchitectural characterisation of the full set of the DaCapo benchmarks in less than a day, and (2) presenting a novel HW/SW co-designed optimisation that performs dynamic load elimination for array length retrieval achieving up to 14% L1 data cache loads reduction and up to 4% dynamic energy reduction. MaxSim is available at `https://github.com/arodchen/MaxSim` released as free software.

In Chapter 7, the MaxSim platform was demonstrated by exploring opportunities for type information elimination from objects on architectures with tagged pointers support. Although this technique significantly reduces memory footprint, it has been shown that retrieving type information from object pointer tags without extra HW support can degrade performance. To address performance issues, novel hardware extensions have been proposed. These extensions are related to the address generation unit removing the performance degradations associated with CIP retrieval from tagged pointers, and to the load-store unit for efficient load-decompression and store-compression of tagged object pointers. The experimental results for the proposed HW/SW co-designed technique, across all the DaCapo benchmark suite, SLAMBench, pseudo-SPECjbb2005 and GraphChi-PR executed to completion, show up to 26% and 10% geometric mean heap space savings, up to 50% and 12% geometric mean dynamic DRAM energy reduction, and up to 49% and 3% geometric mean execution time reduction with no significant performance regressions. Although the proposed technique has been researched in the context of a research Java virtual machine implementation, it can be widely applicable to other object-oriented languages and managed runtimes.

8.2 Future Work

8.2.1 Specialisation of Barrier Synchronisation

Exploration on Other Architectures

The extensible parameterised cbarriers framework for empirical evaluation of barrier synchronisation algorithms used in Chapter 3 can be further exploited to research barrier synchronisation on other multi- and many-core systems. One of the possible topics in this research directions is research of NUMA-aware barrier synchronisation. The cbarriers framework was parameterised so that locations of the shared variables of the barrier synchronisation algorithms can be in either the nodes which mainly write or read them. Some preliminary experiments in this direction were carried out on the platform presented in Table 8.1 featuring two NUMA nodes. On this platform, it was found that the overhead of the dissemination barrier synchronisation algorithm is 22% less when synchronisation variables are located in the writing NUMA nodes as shown in Figure 8.1a compared to the case when synchronisation variables are located in the reading NUMA nodes as shown in Figure 8.1b. Similar experiments and observations

Name	Intel Xeon CPU E5-2620
Microarchitecture	Sandy Bridge
Number of sockets	2
Number of cores	12
Number of SMT threads	24
Number of NUMA nodes	2

Table 8.1: Experimental platform with two NUMA nodes.

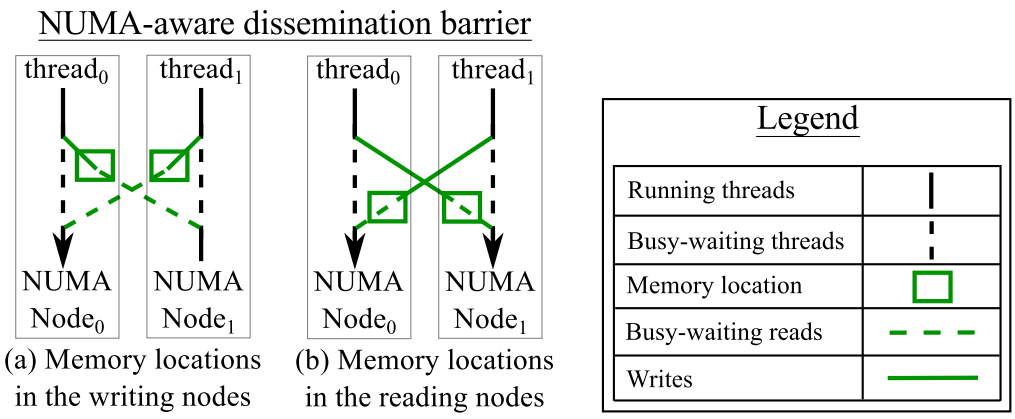


Figure 8.1: Diagram of the NUMA-aware dissemination barrier synchronisation algorithm via shared memory.

were conducted and explained by Dice [Dic12].

Furthermore, the hybrid barrier approach presented in Chapter 3 has applications outside the scope of the Intel Xeon Phi 5110P coprocessor. In fact, it can be applied to any many-core system, where threads can be grouped is such a way, so that the latency of inter-group communication is significantly higher than the latency of intra-group communication between two threads. So, for instance, on the platform presented in Table 8.1, it may be beneficial to use the hybrid barrier which uses one of the tree barriers for synchronisation inside the NUMA node and the dissemination barrier for the inter-node synchronisation.

Dynamic Adaptation

Figures 3.7b and 3.8b, presented in Chapter 3, show that the best performance barrier synchronisation algorithm is dependent on the level of parallelism and the workload. Thus dynamic adaptation of barrier synchronisation algorithm can be researched to improve performance and energy efficiency. The area of the adaptive barrier synchronisation appears to be an underinvestigated research topic. One of the relevant works in this area concerns adaptive backoff barrier synchronisation to reduce memory traffic via reducing the access rate to synchronisation variables [AC89]. However, the problem of adaptive switching to a more optimal synchronisation algorithm at runtime remains open.

8.2.2 HW/SW Co-Designed General-Purpose CPUs and MREs

Object-Relative Addressing of Final References via Tagged Pointers

Venstermans *et al.* [VEDB07b] showed that object-relative addressing is an efficient way of improving memory utilisation in the context of 64-bit JVMs. The general idea of this technique is that references inside objects can be encoded as offsets from the base addresses of the referencing objects to the base addresses of the referenced objects. In case the address distance between the referencing and referenced objects meets certain requirements, the reference can be encoded as an offset in 32 bits instead of 64 bits of the ordinary addressing scheme.

Similarly, as future work, it is suggested to research an opportunity to encode final references inside objects via object-relative addressing in tagged pointers; a final reference in Java terminology is such a reference inside an object that is set during object initialisation and cannot be changed during the lifetime of an object. As a first step, it

is suggested that final references, encoded in the tags, are not completely eliminated from objects as in the technique of type information elimination from object headers described in Chapter 7. Rather, it is proposed to apply object-relative addressing via tagged pointers in the way similar to array length encoding assisted by HW extensions described in Chapter 6 to benefit from eliminated loads but not space savings. However, in contrast to array length tagging, object-relative addressing via tagged pointers should account for two additional challenges: (1) different possible offsets of final references inside objects, and (2) updates of the tags encoding final references via object-relative addressing during garbage collection.

Continuous HW-Assisted Data Layout Transformation

The MaxSim platform, presented in Chapter 6, features the address space morphing technique. One of the capabilities of this technique is the ability to perform fields reordering inside objects depending on object pointer tags and provided fields reordering maps. One of the possible future research directions is an exploration of the ability to perform such a remapping in HW to facilitate continuous data layout transformation in MREs. For continuous (non-stop-the-world) data transformation, objects of transformed and original layouts can be distinguished by different tags assigned to object pointers. In the same fashion, emitted memory access instructions in generated code can be labelled with corresponding tags. In case tags of a memory access instruction and an accessed object pointer do not match during memory access operation execution, a remapping HW mechanism can recalculate an offset so that a correct field is accessed using a dedicated HW buffer, which stores a mapping between original and transformed object layouts. When at some point of execution (which is projected to be garbage collection) an MRE can prove that all the live objects of the same type have a single layout, tagging can be cleared for both memory access instructions in generated code and object pointers. In contrast to applying data layout transformation to all live objects at once, which incurs significant pause time, the proposed technique is projected to increase utilisation of memory subsystem by applying object layout transformation gradually.

Bibliography

[5-L16] 5-level paging and 5-level EPT white paper. https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, 2016. [Online; last accessed 27 June 2017].

[AAC+99] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 314–324, 1999.

[AC89] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA)*, pages 396–406, 1989.

[ACU15] Adria Armejach, Adrian Cristal, and Osman S. Unsal. Tidy cache: Improving data placement in die-stacked DRAM caches. In *Proceedings of the 2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 65–73, 2015.

[Ada92] Don Adams. CRAY T3D system architecture overview manual. ftp://ftp.cray.com/product-info/mpp/T3D_Architecture_Over/T3D.overview.html, 1992. [Online; last accessed 27 June 2017].

[ADE+01] Vishal Aslot, Max J. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In *Proceedings of the International Workshop on OpenMP Applications and Tools:*

OpenMP Shared Memory Parallel Programming (WOMPAT), pages 1–10, 2001.

[AJK+15] K. Aingaran, S. Jairath, G. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, and T. Wicki. M7: Oracle's next-generation Sparc processor. *IEEE Micro*, 35(2):36–45, March 2015.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS)*, pages 483–485, 1967.

[ARM15] ARM Cortex-A series programmer's guide for ARMv8-A. `http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf`, 2015. [Online; last accessed 27 June 2017].

[AS16] Ayaz Akram and Lina Sawalha. A comparison of x86 computer architecture simulators. `http://sc16.supercomputing.org/sc-archive/tech_poster/tech_poster_pages/post233.html`, 2016. [Online; last accessed 27 June 2017].

[ATBC+04] A. R. Adl-Tabatabai, Jay Bharadwaj, M. Cierniak, M. Eng, J. Fang, B. T. Lewis, B. R. Murphy, and J. M. Stichnoth. Improving 64-bit Java IPF performance by compressing heap references. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 100–110, 2004.

[Bab00] Boris Babayan. E2K technology and implementation. In *Proceedings of the 6th International Euro-Par Conference on Parallel Processing (Euro-Par)*, pages 18–21, 2000.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, August 2011.

[BFG02] David F. Bacon, Stephen J. Fink, and David Grove. Space- and time-efficient implementation of the Java object model. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, pages 111–132, 2002.

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924, July 2003.

[BGH+06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[BH09] Luiz Andre Barroso and Urs Hoelzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 1st edition, 2009.

[BJK+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–268, 1998.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[Bor07] Shekhar Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference (DAC)*, pages 746–749, 2007.

[Bra16] David Brash. ARMv8-A architecture – 2016 additions. `https://community.arm.com/groups/processors/blog/2016/10/27/armv8-a-architecture-2016-additions`, 2016. [Online; last accessed 27 June 2017].

[Bro86] Eugene D. Brooks, III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[Bul99] J. M. Bull. Measuring synchronisation and scheduling overheads in OpenMP. In *Proceedings of the First European Workshop on OpenMP (EWOMP)*, pages 99–105, 1999.

[Cab] Diego Caballero. SIMD@OpenMP: A programming model approach to leverage SIMD features. *PhD Dissertation.* `https://upcommons.upc.edu/bitstream/handle/2117/96011/TDLCdG1de1.pdf`. [Online; last accessed 27 June 2017].

[CC04] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 221–232, 2004.

[CDL99] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24, 1999.

[CDM13] Diego Caballero, Alejandro Duran, and Xavier Martorell. An OpenMP barrier using SIMD instructions for Intel® Xeon Phi coprocessor. In *Proceedings of the 9th International Workshop on OpenMP (IWOMP)*, pages 99–113. 2013.

[CFS14] CFS scheduler. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`, 2014. [Online; last accessed 27 June 2017].

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[CHE11] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multicore simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12, 2011.

[CHE+14] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):28:1–28:25, August 2014.

[Cli10] Cliff Click. Cliff Click's blog: Biased locking. `http://www.cliffc.org/blog/2010/01/09/biased-locking/`, 2010. [Online; last accessed 27 June 2017].

[CMB+13] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 308–319, 2013.

[Cow13] James Cownie. Fastest possible barrier (Intel developer zone forum discussion). `http://software.intel.com/en-us/forums/topic/392587`, 2013. [Online; last accessed 27 June 2017].

[CPN+09] Eric S. Chung, Michael K. Papamichael, Eriko Nurvitadhi, James C. Hoe, Ken Mai, and Babak Falsafi. ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(2):15:1–15:32, June 2009.

[CPST15] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. Memento mori: Dynamic allocation-site-based optimizations. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM)*, pages 105–117, 2015.

[CSK+07] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 249–261, 2007.

[DEB94] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Technical report, 1994.

[DFF+13] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 381–394, 2013.

[DGR+74] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[DHR+15] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 487–502, 2015.

[Dic12] Dave Dice. NUMA-aware placement of communication variables. `https://blogs.oracle.com/dave/numa-aware-placement-of-communication-variables`, 2012. [Online; last accessed 27 June 2017].

[DKK07] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 482–493, 2007.

[Dol14] Romain Dolbeau. Address selection for efficient barriers on the Intel Xeon Phi. `http://www.dolbeau.name/dolbeau/publications/barrierphi.pdf`, 2014. [Online; last accessed 27 June 2017].

[EBS17a] OpenCL overview. `https://www.khronos.org/opencl/`, 2017. [Online; last accessed 27 June 2017].

[EBS17b] The OpenMP API specification for parallel programming. `http://openmp.org`, 2017. [Online; last accessed 27 June 2017].

[EBSA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multi-core scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[Eec10] Lieven Eeckhout. *Computer Architecture Performance Evaluation Methods*. Morgan & Claypool Publishers, 1st edition, 2010.

[Fab74] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[Fog16] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. `http://www.agner.org/optimize/instruction_tables.pdf`, 2016. [Online; last accessed 27 June 2017].

[GBEDB04] Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Method-level phase behavior in Java workloads. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 270–287, 2004.

[GdL16] Cosmin Gorgovan, Amanieu d'Antras, and Mikel Luján. MAMBO: A low-overhead dynamic binary modification tool for ARM. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):14:1–14:26, April 2016.

[Goo15] Google. ART and Dalvik. `https://source.android.com/devices/tech/dalvik/index.html`, 2015. [Online; last accessed 27 June 2017].

[Gra16] OpenJDK: Graal project. `http://openjdk.java.net/projects/graal/`, 2016. [Online; last accessed 27 June 2017].

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 2001 IEEE International Workshop on Workload Characterization (WWC)*, pages 3–14, 2001.

[GV94] D. Grunwald and S. Vajracharya. Efficient barriers for distributed shared memory computers. In *Proceedings of the 8th International Parallel Processing Symposium (IPPS)*, pages 604–608, 1994.

[Haz11] Kim Hazelwood. *Dynamic Binary Modification*. Morgan & Claypool Publishers, 1st edition, 2011.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, 1992.

[HFM88] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[HMMR04] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. A survey of barrier algorithms for coarse grained supercomputers. *Chemnitzer Informatik Berichte*, 2004.

[HMMR06] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. Fast barrier synchronization for InfiniBand. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS), CAC'06 Workshop*, 2006.

[HMN09] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 413–422, 2009.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 4th edition, 2006.

[HQW+10] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 37–47, 2010.

[HRSS11] Wei Huang, K. Rajamani, M.R. Stan, and K. Skadron. Scaling with design constraints: Predicting the future of big chips. *IEEE Micro*, 31(4):16–29, July 2011.

[HSC+12] Wim Heirman, Souradip Sarkar, Trevor E. Carlson, Ibrahim Hur, and Lieven Eeckhout. Power-aware multi-core simulation for early design stage hardware/software co-optimization. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 3–12, 2012.

[HSDH04] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 251–269, 2004.

[HSH81] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA)*, pages 341–348, 1981.

[HTCU10] Tim Harris, Sasa Tomic, Adrian Cristal, and Osman S. Unsal. Dynamic filtering: multi-purpose architecture support for language runtime systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 39–52, 2010.

[IN12] Hiroshi Inoue and Toshio Nakatani. Identifying the sources of cache misses in Java programs without relying on hardware counters. In *Proceedings of the 2012 International Symposium on Memory Management (ISMM)*, pages 133–142, 2012.

[Int08] 754-2008 - IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[Int11] Intel® 64 and IA-32 architectures software developer's manual. volume 1: Basic architecture. `http://download.intel.com/design/processor/manuals/253665.pdf`, 2011. [Online; last accessed 27 June 2017].

[Int12] Intel® Xeon Phi™ coprocessor instruction set architecture reference manual. `https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf`, 2012. [Online; last accessed 27 June 2017].

[Int14] Intel® Xeon Phi™ coprocessor system software developers guide. `https://software.intel.com/sites/default/files/managed/09/07/xeon-phi-coprocessor-system-software-developers-guide.pdf`, 2014. [Online; last accessed 27 June 2017].

[Int16] Floating point unit demonstration on STM32 microcontrollers. `www.st.com/resource/en/application_note/dm00047230.pdf`, 2016. [Online; last accessed 27 June 2017].

[Int17] Vega 3 compute appliances. `https://www.azul.com/products/vega/vega-3-compute-appliances/`, 2017. [Online; last accessed 27 June 2017].

[JCL17] Java™ platform, standard edition 7 API specification. `https://docs.oracle.com/javase/7/docs/api/overview-summary.html`, 2017. [Online; last accessed 27 June 2017].

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[Jik14] Jikes–Sniper page in Sniper online documentation. `http://snipersim.org/w/Jikes`, 2014. [Online; last accessed 27 June 2017].

[KBG12] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[KCR+17] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. Heterogeneous managed runtime systems: A computer vision case study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 74–82, 2017.

[KKC+13] R. Krishnaiyer, E. Kultursay, P. Chawla, S. Preis, A. Zvezdin, and H. Saito. Compiler-based data prefetching and streaming non-temporal store generation for the Intel® Xeon Phi™ coprocessor. In *Proceedings of the 2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pages 1575–1586, 2013.

[Kli81] Paul Klint. Interpretation techniques. *Software Practice and Experience*, 11(9):963–973, 1981.

[KRB+16] Christos Kotselidis, Andrey Rodchenko, Colin Barrett, Andy Nisbet, John Mawer, Will Toms, James Clarkson, Cosmin Gorgovan, Amanieu d'Antras, Yaman Cakmakci, Thanos Stratikopoulos, Sebastian Werner, Jim D. Garside, Javier Navaridas, Antoniu Pop, John Goodacre, and Mikel Luján. Project Beehive: A hardware/software co-designed stack for runtime and architectural research. In *Proceedings of the 10th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG)*, 2016.

[KS04] Tejas S. Karkhanis and James E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 338–349, 2004.

[KVBWA12] Asif Khan, Muralidaran Vijayaraghavan, Silas Boyd-Wickizer, and Arvind. Fast and cycle-accurate modeling of a multicore processor. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 178–187, 2012.

[KWM+08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):7:1–7:32, May 2008.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LAS+13] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1):5:1–5:29, April 2013.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, 1984.

[LH10] James Larus and Galen Hunt. The Singularity system. *Communications of the ACM*, 53(8):72–79, August 2010.

[LYBB14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[MAKB03] S. Mathew, M. Anders, R. K. Krishnamurthy, and S. Borkar. A 4-GHz 130-nm address generation unit with 32-bit sparse-tree adder core. *IEEE Journal of Solid-State Circuits*, 38(5):689–695, May 2003.

[MCE+02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[MCS91] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, February 1991.

[MCS14a] Altera SoC FPGA. `http://www.arm.com/products/tools/altera-soc-fpga.php`, 2014. [Online; last accessed 27 June 2017].

[MCS14b] Zynq-7000 all programmable SoC. `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/`, 2014. [Online; last accessed 27 June 2017].

[MG11] Zoltan Majo and Thomas R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR)*, pages 12:1–12:10, 2011.

[MJCM] Niall Murphy, Timothy Jones, Simone Campanoni, and Robert Mullins. Limits of dependence analysis for automatic parallelization.

[MKK+10] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[MPG+17] John Mawer, Oscar Palomar, Cosmin Gorgovan, Andrey Rodchenko, Andy Nisbet, Will Toms, and Mikel Luján. The potential of dynamic binary modification and CPU-FPGA SoCs for simulation. In

Proceedings of the 25th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017.

[MSB+05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, November 2005.

[MWG00] Erik Meijer, Redmond Wa, and John Gough. Technical overview of the Common Language Runtime, 2000.

[NAS] NAS parallel benchmarks. `http://www.nas.nasa.gov/publications/npb.html`. [Online; last accessed 27 June 2017].

[NBZ+15] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly, Andrew J. Davison, Mikel Luján, Michael F. P. O'Boyle, Graham Riley, Nigel Topham, and Steve Furber. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *Proceedings of the 2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5783–5790, 2015.

[.NE17] .NET framework class library. `https://msdn.microsoft.com/en-us/library/gg145045(v=vs.110).aspx`, 2017. [Online; last accessed 27 June 2017].

[NFX+16] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 349–365, 2016.

[NIH+11] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 127–136, 2011.

[NNJA74] Kesav V. Nori, H.H. Naegeli, Kurt Jensen, and Urs Ammann. The PASCAL (P) compiler : implementation notes. Technical report, 1974.

[OOK+10] Rei Odaira, Kazunori Ogata, Kiyokuni Kawachiya, Tamiya Onodera, and Toshio Nakatani. Efficient runtime tracking of allocation sites in Java. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 109–120, 2010.

[Ope17] OpenJDK. `http://openjdk.java.net`, 2017. [Online; last accessed 27 June 2017].

[Org73] Elliott Irving Organick. *Computer System Organization: The B5700/B6700 Series (ACM Monograph Series)*. Academic Press, Inc., 1973.

[PACG11] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSS: A full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference (DAC)*, pages 1050–1055, 2011.

[PAK+11] Michael Pellauer, Michael Adler, Michel A. Kinsy, Angshuman Parashar, and Joel S. Emer. HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–417, 2011.

[PBMW98] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. In *Proceedings of the 7th International World Wide Web Conference (WWW)*, pages 161–172, 1998.

[Phi14] Stephen Phillips. M7: Next generation SPARC. `http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/migration/m7-next-gen-sparc-presentation-2326292.html`, 2014. [Online; last accessed 27 June 2017].

[PJB05] pjbb2005. `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`, 2005. [Online; last accessed 27 June 2017].

[Poi17] Pointer authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017. [Online; last accessed 27 June 2017].

[Pow08] J. R. Powell. The quantum limit to Moore's Law. *Proceedings of the IEEE*, 96(8):1247–1248, August 2008.

[PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA)*, pages 348–354, 1984.

[Pro14] Protocol Buffers - Google's data interchange format (ver. 2.6.1). https://developers.google.com/protocol-buffers/, 2014. [Online; last accessed 27 June 2017].

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot™ server compiler. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.

[RDS15] Ravi Rajwar, Martin Dixon, and Ronak Singhal. Specialized evolution of the general purpose CPU. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[Ree13] Robert Reed. Intel® Xeon Phi™ coprocessor February developer webinar Q&A responses. https://software.intel.com/en-us/articles/intelr-xeon-phitm-coprocessor-february-developer-webinar-qa-responses, 2013. [Online; last accessed 27 June 2017].

[RH13] Sabela Ramos and Torsten Hoefler. Modeling communication in cache-coherent SMP systems: A case-study with Xeon Phi. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 97–108, 2013.

[Ric69] Martin Richards. BCPL: A tool for compiler writing and system programming. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 557–566, 1969.

[RKN+17] Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, and Mikel Luján. MaxSim: A simulation platform for managed applications. In *Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 141–151, 2017.

[RKN+18] Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, and Mikel Luján. Type information elimination from objects on architectures with tagged pointers support. *IEEE Transactions on Computers*, 67(1):130–143, Jan 2018.

[RNPL15] Andrey Rodchenko, Andy Nisbet, Antoniu Pop, and Mikel Luján. Effective barrier synchronization on Intel Xeon Phi coprocessor. In *Proceedings of the 21st International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 588–600, 2015.

[SGC+06] J. Sampson, R. Gonzalez, J. F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 235–246, 2006.

[SHB+14] Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S. McKinley. Cooperative cache scrubbing. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 15–26, 2014.

[SHC+04] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium*, page 5, 2004.

[Shi15] Boris Shingarov. Live introspection of target-agnostic JIT in simulation. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST)*, pages 5:1–5:9, 2015.

[SHW11] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[SJL11] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the NAS parallel benchmarks in OpenCL. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 137–148, 2011.

[SK10] John Sartori and Rakesh Kumar. Low-overhead, high-speed multicore barrier synchronization. In *Proceedings of the 5th International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*, pages 18–34, 2010.

[SK13] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 475–486, 2013.

[SKK+15] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter. Tejas: A java based versatile micro-architectural simulator. In *25th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 47–54, Sept 2015.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., 2005.

[SPE05] SPECjbb2005. `http://www.spec.org/jbb2005/`, 2005. [Online; last accessed 27 June 2017].

[SPE06] SPEC CPU® 2006. `http://www.spec.org/cpu2006/`, 2006. [Online; last accessed 27 June 2017].

[SPE08] SPECjvm2008 benchmarks. `http://www.spec.org/jvm2008`, 2008. [Online; last accessed 27 June 2017].

[SPSS08] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd Annual*

International Conference on Supercomputing (ICS), pages 277–288, 2008.

[SS86] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA)*, pages 414–423, 1986.

[SSO+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[Sta12] Standard ECMA-335, Common Language Infrastructure (CLI). `https://www.ecma-international.org/publications/standards/Ecma-335.htm`, 2012. [Online; last accessed 27 June 2017].

[Ste77] G.L. Steele. *Data Representation in PDP-10 MacLISP*. Artificial intelligence memo. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1977.

[STZ+12] S. Stipic, S. Tomic, F. Zyulkyarov, A. Cristal, O. Unsal, and M. Valero. TagTM - accelerating STMs with hardware tags for fast meta-data access. In *Proceedings of the 2012 Design, Automation, and Test in Europe Conference and Exhibition (DATE)*, pages 39–44, March 2012.

[Tei12] J. Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, May 2012.

[THC+14] Chia-Heng Tu, Hui-Hsin Hsu, Jen-Hao Chen, Chun-Han Chen, and Shih-Hao Hung. Performance and power profiling for emulated Android systems. *ACM Transactions on Design Automation of Electronic Systems*, 19(2):10:1–10:25, March 2014.

[TIO] TIOBE programming community index. `https://www.tiobe.com/tiobe-index/`. [Online; last accessed 27 June 2017].

[Und04] Keith Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 171–180, 2004.

[VEDB07a] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. Java object header elimination for reduced memory consumption in 64-bit virtual machines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(3):17:1–17:30, September 2007.

[VEDB07b] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. Object-relative addressing: Compressed pointers in 64-bit java virtual machines. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*, pages 79–100, 2007.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC)*, pages 1–27, 1998.

[Whi12] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[WHVDV+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):30:1–20:24, January 2013.

[WM10] Christian Wimmer and Hanspeter Mössenbösck. Automatic feedback-directed object fusing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(2):7:1–7:35, October 2010.

[WMGW06] Greg Wright, Phil McGachey, Erika Gunadi, and Mario Wolczko. Introspection of a Java™ virtual machine under simulation. Technical report, 2006.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 24–36, 1995.

[WW12] Christian Wimmer and Thomas Würthinger. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, pages 13–14, 2012.

[YTL87] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, 1987.

Appendix A

cbarriers Framework Manual

The cbarriers framework used in Chapter 3 for evaluation of different barrier synchronisation algorithms is located at `https://github.com/arodchen/cbarriers` under the Apache v2.0 free software license. This appendix provides a brief description how to use this framework and what other packages it relies on. Furthermore, it contains the command that was used to run the `EPCC` and `NBODY` benchmarks described in Chapter 3 on Intel Xeon Phi 5110P. Part of this description was published at `https://github.com/arodchen/cbarriers/blob/master/README.md`.

A.1 Dependencies

The framework requires the following packages to be installed:
`make, gcc, r-base, r-cran-gplots, data.table package in R`

A.2 Usage

The cbarriers framework operation is described in the `makefile` and is carried out by the `make` utility.

A.2.1 Building, Running Benchmarks, Plotting Results

`make build` - builds barrier evaluation framework.

`make test` - runs tests.

`make db` - creates data base needed for plotting.

`make plot` - plots charts in pdf.

`make all` - does all mentioned above.

`make help` - prints help message.

A.2.2 Help Message

```
Usage: make [target] [arguments]

Available targets: help
                   build    cleanbuild
                   test     cleantest
                   db       cleandb
                   plot     cleanplot
                   all      cleanall


Available arguments [name={valid values} (default value) - description]:

 EXTRACFLAGS={...} ( ) - extra CFLAGS
 EXTRALDFLAGS={...} ( ) - extra LDFLAGS

 TEST_RUNNER={...} ( ) - test runner
 TEST_RUNNER_PASS_ARG_BEG={...} ( ) - option to begin passing arguments to
     test runner
 TEST_RUNNER_PASS_ARG_END={...} ( ) - option to end passing arguments to
    test runner
 TEST_RUNNER_INIT={...} ( ) - option to initialise test runner
 TEST_RUNNER_FINI={...} ( ) - option to finalise test runner

 CPUS_NUM={1..N} (shell nproc) - number of available CPUs
 THREADS_SURPLUS={(1-CPUS_NUM)..N} (0) - difference between the number
     of threads and CPUs
 THREADS_MAX_NUM={1..N} (CPUS_NUM + THREADS_SURPLUS) -
    maximum number of threads (THREADS_SURPLUS has priority over
    THREADS_MAX_NUM if both are defined)
```

THREADS_MIN_NUM={1..THREADS_MAX_NUM} (1) - minimum number of threads
BARRIERS_NUM={1..N} (1000000) - number of barriers in the experiment
EXPERIMENTS_NUM={1..N} (10) - number of experiments
CCL_SIZE={1..N} (128) - cache coherency line size
ARCH={x86_64|mic|armv7l} (shell arch) - target architecture
HOST_NAME={...} (shell hostname) - target hostname
CPU_MAP_PRIORITY_DELTA={1..CPUS_NUM} (1) - priority delta for mapping threads to CPUs
THREADS_INC={+=1,*=2,...} (+= 1) - compound assignment operator for the increment of the number of threads
RADIX_INC={+=1,*=2,...} (+= 1) - compound assignment operator for the increment of radix
RADIX_MAX={2..THREADS_MAX_NUM} (THREADS_MAX_NUM) - maximum radix
OMP_IMPL={gnu,intel} (gnu) - OMP library implementation
INTERPOLATE_RADIX={yes,no} (no) - when number of threads participating in barrier is less than radix then the result is taken from experiment when number of threads participating in a barrier is equal to radix
PRINT_SYNCH_UNSYNCH_PHASE_TIME={yes,no} (yes) - printing synchronised and unsynchronised phase execution time, where the phase is an interval by two adjacent barriers
TOPOLOGY_AWARE_MAPPING={yes,no} (yes) - use topology information while mapping threads to nodes of the barrier
TOPOLOGY_NUMA_AWARE_ALLOC={yes,no} (yes) - use topology information while allocating memory for barriers on NUMA machines
USER_DEFINED_ACTIVE_PUS_SELECTION={yes,no} (no) - user defined active PUs selection controlled by CPU_MAP_PRIORITY_DELTA
CHARTS_IGNORE_BARRIERS={sr,...,pthread} () - comma-separated list of barriers to ignore on charts
CHARTS_SUR_ONLY_SPINNINGS={ptyield,...,hwyield} () - comma-separated list of spinning to be used on surplused charts

A.3 Recipes

The following command runs tests on Intel Xeon Phi 5110P:

```
make test TEST_RUNNER='ssh mic0 "ulimit -s unlimited; export KMP_LIBRARY=
    turnaround; export KMP_BLOCKTIME=infinite;'
    TEST_RUNNER_PASS_ARG_END='"' HOST_NAME=mic0
    TEST_RUNNER_INIT='scp -r bin mic0:' TEST_RUNNER_FINI='ssh mic0 "rm
    -rf bin"' CC=icc CPUS_NUM=232 EXTRACFLAGS="-mmic"
    BARRIERS_NUM=10000 ARCH=mic
    TOPOLOGY_NUMA_AWARE_ALLOC=no THREADS_MIN_NUM=8
    THREADS_INC=+=8 RADIX_INC=*=2 EXPERIMENTS_NUM=10
```

Appendix B

MaxSim Platform Manual

The MaxSim platform for simulation of managed applications presented in Chapter 6 is located at `https://github.com/arodchen/MaxSim` under the GPLv2 free software license. This appendix provides a brief description how to use this platform and what other packages it relies on. This description was published at `https://github.com/arodchen/MaxSim/blob/master/README.md`.

B.1 Dependencies

The platform requires the following dependencies to be satisfied and the following packages to be installed:

```
Maxine VM 2.0.5 dependencies, Oracle JDK 7u25 for Linux x64,
ZSim dependencies, McPAT 1.0 dependencies, timelimit, protobuf-2.6.1
```

B.2 Required Environment Variables

Setting of the following environmental variables is required to find the tools necessary for the MaxSim platform operation:

```
Maxine VM 2.0.5, ZSim, McPAT 1.0 required environment variables

PROTOBUFPATH=<protobuf-2.6.1 install path>

MCPATPATH=<McPAT 1.0 bin path>
```

B.3 Usage

B.3.1 Building, Cleaning, Style Checking, and Setting Kernel Parameters

`./scripts/generateMaxSimInterface.sh` - generates MaxSim interface.

`./scripts/setZSimKernelParameters.sh` - sets ZSim kernel parameters (requires sudo).

`./scripts/buildMaxine<Debug|Product>.sh` - builds Maxine VM (and re-generates MaxSim interface).

`./scripts/buildImageC1X<Debug|Product>.sh` - builds Maxine VM image.

`./scripts/buildZSim<Debug|Product>.sh` - builds ZSim (and re-generates MaxSim interface).

`./scripts/buildMaxSim<Debug|Product>.sh` - builds MaxSim (does all mentioned above).

`./scripts/cleanMaxine.sh` - cleans Maxine VM.

`./scripts/cleanZSim.sh` - cleans ZSim.

`./scripts/cleanMaxSim.sh` - cleans Maxine VM and ZSim.

`./scripts/checkStyle.sh` - checks style in Maxine VM.

B.3.2 Running DaCapo-9.12-bach Benchmarks

Command:

```
./scripts/runMaxSimDacapo.sh <output directory> <ZSim template configuration>
    <number of runs>
```

Arguments:

```
<output directory> - existing output directory where results are stored (overwrites
    existing results in the directory)
<ZSim template configuration> - ZSim template configuration in which
    COMMAND_TEMPLATE is replaced by actual command to be executed by
    ZSim (e.g. ./zsim/tests/*.tmpl)
<number of runs> - number of runs of each benchmark
EXTRA_MAXINE_FLAGS - environment variable used to pass extra flags to
    Maxine VM
```

B.3.3 MaxSim Interface and Configuration

MaxSim interface is defined using Protocol Buffers 2.6.1 in the following file:

```
./maxine/com.oracle.max.vm/src/com/sun/max/vm/maxsim/MaxSimInterface.proto
```

Default values of `message MaxSimConfig` define a build-time MaxSim configuration.

`isMaxSimEnabled` indicates whether Maxine and ZSim are configured to work in tandem (`true`) or separately (`false`).

`pointerTaggingType` indicates a type of active pointer tagging. Three types of pointer tagging are available: `NO_TAGGING` - native x86-64 tagging (sign-extension of the 47th bit), `CLASS_ID_TAGGING` - objects tagging by their class IDs, `ALLOC_SITE_ID_TAGGING` - objects tagging by IDs of allocation sites.

`layoutScaleFactor` and `layoutScaleRefFactor` are parameters of two bijections of the address space morphing scheme described in Chapter 6. `layoutScaleFactor` is the second parameter of f_e and the first parameter of f_c bijection. `layoutScaleRefFactor` is the first parameter of f_e bijection.

B.3.4 ZSim MaxSim-Related Configuration Parameters

The `pointerTagging` simulation parameter indicates whether pointer tagging simulation is enabled in ZSim.

The `MAProfCacheGroupId` compact ID can be assigned to a cache. When MaxSim profiling is active, the event related to a specific cache will be aggregated in the corresponding MAProfCacheGroup. `MAProfCacheGroupNames` parameter is associated with caches, and it defines names of MAProfCacheGroups delimited by `|` symbol (*e.g.* ./zsim/tests/*.tmpl).

Note. When working in tandem with Maxine VM `startFastForwarded` Maxine VM process parameter should be set to `true`. Exiting fast forwarding should be performed explicitly in Maxine VM.

B.3.5 MaxineVM MaxSim-Related Flags

`-XX:-MaxSimEnterFFOnVMExit` - makes MaxSim enter fast forwarding mode on VM exit (default: `false`).

`-XX:-MaxSimExitFFOnVMEnter` - makes MaxSim exit fast forwarding mode on VM enter (default: `false`).

`-XX:MaxSimMaxineInfoFileName=<value>` - MaxSim Maxine information file name (default: maxine-info.db).

`-XX:-MaxSimPrintProfileOnVMExit` - makes MaxSim print profiling information on VM exit (default: `false`).

`-XX:-MaxSimProfiling` - enables MaxSim profiling (default: `false`).

`-XX:MaxSimZSimProfileFileName=<value>` - MaxSim ZSim profile file name (default: zsim-prof.db).

`-XX:-TraceMaxSimTagging` - traces MaxSim tagging.

`-XX:MaxSimDataTransDB=<value>` - MaxSim data transformation database for address space morphing.

Note. All the flags, related to collecting and printing profiling information, have effect only when either `pointerTaggingType [default = ALLOC_SITE_ID_TAGGING]` or `pointerTaggingType [default = CLASS_ID_TAGGING]`. `-XX:MaxSimDataTransDB=` accepts `DataTransDB` message with `DataTransInfo`s having `FieldOffsetRemapPair`s representing m_e reordering map described in Chapter 6. `-XX:MaxSimDataTransDB=` has effect only when `pointerTaggingType [default = CLASS_ID_TAGGING]`.

B.3.6 Controlling Simulation by Managed Applications

MaxSim simulation can be controlled by applications by setting `MaxSim.Command` property (via a call to `System.setProperty("MaxSim.Command", <value>)`) the following values:

`"ROI_BEGIN()"` - exits fast-forwarding mode and starts simulation of a region of interest.

`"ROI_END()"` - enters fast-forwarding mode and stops simulation of a region of interest.

`"PRINT_PROFILE_TO_FILE(<file name>)"` - prints profile to a file with a specified name.

`"RESET_PROFILE_COLLECTION()"` - resets profile collection.

Note. All the commands, related to collecting and printing profiling information, have effect only when either `pointerTaggingType [default = CLASS_ID_TAGGING]` or `pointerTaggingType [default = ALLOC_SITE_ID_TAGGING]`, and when, at the same time, `-XX:+MaxSimProfiling` flag is passed to Maxine VM.

B.3.7 Printing Profiling Information in the Textual Format

Command:

```
cd maxine
../graal/mxtool/mx maxsimprofprint <flags>
```

Flags:

```
-MaxineInfoDB=<arg>
    Location of the file containing Maxine information database.
-ZSimProfileDB=<arg>
    Location of the file containing ZSim profile database.
-help[=true|false, t|f, y|n] (default: false)
    Show help message and exit.
-o=<arg> (default: maxsim-prof.txt)
    Output file name.
```

B.3.8 Retrieving Statistics Collected by ZSim

Command:

```
./scripts/retrieveZSimStat.py <ZSim stat dir> <Maxine VM oper modes>
    <characteristic> (<cahe name>)
```

Arguments:

```
<ZSim stat dir> - directory containing ZSim stat files (zsim-ev.h5)
<Maxine VM oper modes> - comma-separated numerical list of Maxine VM
    operation modes for which statistics is retrieved. Operation modes are listed in
    MaxineVMOperationMode in MaxSimInterface.proto
<characteristic> - retrieved characteristic. Supported characteristics are:
    C - cycles
    I - instructions
    IPC - instructions per clock
    C[H|M|A][LD|ST|LDST](PKI) - cache characteristics
        [..|..] - required alternatives
        (..|..) - optional alternatives
        H - hits
        M - misses
        A - accesses
        LD - loads
        ST - stores
        LDST - loads and stores
        PKI - per kilo instruction
<cahe name> - cache name required only for cache characteristics listed above
```

Note. The parts of this script were obtained from the ZSim-NVMain [ACU15] simulator.

B.3.9 Modelling Power and Energy Using McPAT

Command:

```
./scripts/runMcPAT.py <flags>
```

Flags:

```
  [-h (help)]
  [-z <zsim-stat-dir>] - directory containing ZSim stat files (zsim-ev.h5)
  [-e <maxine-op-modes>] - comma-separated numerical list of Maxine VM
      operation modes for which statistics is retrieved. Operation modes are listed in
      MaxineVMOperationMode in MaxSimInterface.proto
  [-d <resultsdir (default: .)>]
  [-t <type: total|dynamic|static|peak|peakdynamic|area>]
  [-o <output-file (power{.png,.txt,.py})>]
```

Note. The parts of this script were obtained from the Sniper [HSC+12] simulator under the MIT License.

B.4 Recipes

Profiles simple `./maxine/com.oracle.max.tests/src/test/output/HelloWorld.java` application using `4C` ZSim configuration described in Chapter 6:

```
# Changes pointerTaggingType default type to CLASS_ID_TAGGING
sed -i 's/pointerTaggingType = 2 \[default = NO_TAGGING/
    pointerTaggingType = 2 \[default = CLASS_ID_TAGGING/' ./maxine/com.
    oracle.max.vm/src/com/sun/max/vm/maxsim/MaxSimInterface.proto

# Builds MaxSim
./scripts/buildMaxSimProduct.sh

# Simulates HelloWorld application and produces ZSim profile and Maxine
    information files (zsim-prof.db and maxine-info.db)
./zsim/build/release/zsim ./zsim/tests/Nehalem-4C_MaxineHelloWorld.cfg
```

```
# Prints profile to maxsim-prof.txt
pushd maxine
../graal/mxtool/mx maxsimprofprint -MaxineInfoDB=../maxine-info.db
    -ZSimProfileDB=../zsim-prof.db -o=../maxsim-prof.txt
popd

# Changes back pointerTaggingType to NO_TAGGING
sed -i 's/pointerTaggingType = 2 \[default = CLASS_ID_TAGGING/
    pointerTaggingType = 2 \[default = NO_TAGGING/' ./maxine/com.oracle.max.
    vm/src/com/sun/max/vm/maxsim/MaxSimInterface.proto
```

Profiles a simple application performing various operations with a singly linked list `./maxine/com.oracle.max.tests/src/test/output/MaxSimSingleLinkedList.java` using `1CQ` ZSim configuration described in Chapter 6:

```
# Changes pointerTaggingType default type to CLASS_ID_TAGGING
sed -i 's/pointerTaggingType = 2 \[default = NO_TAGGING/
    pointerTaggingType = 2 \[default = CLASS_ID_TAGGING/' ./maxine/com.
    oracle.max.vm/src/com/sun/max/vm/maxsim/MaxSimInterface.proto

# Builds MaxSim
./scripts/buildMaxSimProduct.sh

# Simulates MaxSimSingleLinkedList application and produces three ZSim profile
    and one Maxine information files
./zsim/build/release/zsim ./zsim/tests/Nehalem-1CQ_MaxSimSingleLinkedList.cfg

# Changes back pointerTaggingType to NO_TAGGING
sed -i 's/pointerTaggingType = 2 \[default = CLASS_ID_TAGGING/
    pointerTaggingType = 2 \[default = NO_TAGGING/' ./maxine/com.oracle.max.
    vm/src/com/sun/max/vm/maxsim/MaxSimInterface.proto
```

Characterises and profiles DaCapo-9.12-bach using `1CQ` ZSim configuration described in Chapter 6:

```
# Changes pointerTaggingType default type to ALLOC_SITE_ID_TAGGING
sed -i 's/pointerTaggingType = 2 \[default = NO_TAGGING/
    pointerTaggingType = 2 \[default = ALLOC_SITE_ID_TAGGING/' ./maxine/
    com.oracle.max.vm/src/com/sun/max/vm/maxsim/MaxSimInterface.proto


# Simulates DaCapo-9.12-bach benchmarks and produces ZSim profile files
mkdir dacapo_characterization
EXTRA_MAXINE_FLAGS="-XX:+MaxSimProfiling
    -XX:+MaxSimPrintProfileOnVMExit" ./scripts/runMaxSimDacapo.sh
    dacapo_characterization ./zsim/tests/Nehalem-1CQ.tmpl 1

# Changes back pointerTaggingType to NO_TAGGING
sed -i 's/pointerTaggingType = 2 \[default = ALLOC_SITE_ID_TAGGING/
    pointerTaggingType = 2 \[default = NO_TAGGING/' ./maxine/com.oracle.max.
    vm/src/com/sun/max/vm/maxsim/MaxSimInterface.proto
```

Retrieves L3 Cache Misses per Kilo Instruction:

```
./scripts/retrieveZSimStat.py
    ./dacapo_characterization/zsim/DaCapo-9.12-bach_eclipse_product_0 1,2
    CMLDPKI l3
```

Models energy spent in the Garbage Collection (GC) part of the workload:

```
./scripts/runMcPAT.py
    -z ./dacapo_characterization/zsim/DaCapo-9.12-bach_eclipse_product_0 -e 2
```

Models energy spent in the non-GC part of the workload:

```
./scripts/runMcPAT.py
    -z ./dacapo_characterization/zsim/DaCapo-9.12-bach_eclipse_product_0 -e 1
```

Profiles simple `./maxine/com.oracle.max.tests/src/test/output/HelloWorld.java` application using `1CQ` ZSim configuration and models compressed object pointers optimisation and `String` objects' fields reordering as depicted in Figure 6.3 in Chapter 6 using reordering map `./misc/DataTrans/HelloWorldCompPointDataTrans.db`:

```
# Changes pointerTaggingType default type to CLASS_ID_TAGGING and
    layoutScaleFactor to 2
sed -i 's/pointerTaggingType = 2 \[default = NO_TAGGING/
    pointerTaggingType = 2 \[default = CLASS_ID_TAGGING/' ./maxine/com.
    oracle.max.vm/src/com/sun/max/vm/maxsim/MaxSimInterface.proto
sed -i 's/layoutScaleFactor = 3 \[default = 1/
    layoutScaleFactor = 3 \[default = 2/' maxine/com.oracle.max.vm/src/com/sun/
    max/vm/maxsim/MaxSimInterface.proto

# Builds MaxSim
./scripts/buildMaxSimProduct.sh

# Simulates HelloWorld application and produces ZSim profile and Maxine
    information files (zsim-prof.db and maxine-info.db)
./zsim/build/release/zsim
    ./zsim/tests/Nehalem-1CQ_MaxineHelloWorldCompPointDataTrans.cfg

# Prints profile to maxsim-prof.txt
pushd maxine
../graal/mxtool/mx maxsimprofprint -MaxineInfoDB=../maxine-info.db
    -ZSimProfileDB=../zsim-prof.db -o=../maxsim-prof.txt
popd

# Changes back pointerTaggingType to NO_TAGGING and layoutScaleFactor to 1
sed -i 's/pointerTaggingType = 2 \[default = CLASS_ID_TAGGING/
    pointerTaggingType = 2 \[default = NO_TAGGING/' ./maxine/com.oracle.max.
    vm/src/com/sun/max/vm/maxsim/MaxSimInterface.proto
sed -i 's/layoutScaleFactor = 3 \[default = 2/layoutScaleFactor = 3 \[default = 1/'
    maxine/com.oracle.max.vm/src/com/sun/max/vm/maxsim/MaxSimInterface.
    proto
```