Scalable Emulation of Heterogeneous Systems

Emilio Garcia Cota

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2019

ABSTRACT

Scalable Emulation of Heterogeneous Systems

Emilio Garcia Cota

The breakdown of Dennard's transistor scaling has driven computing systems toward application-specific accelerators, which can provide orders-of-magnitude improvements in performance and energy efficiency over general-purpose processors.

To enable the radical departures from conventional approaches that heterogeneous systems entail, research infrastructure must be able to model processors, memory and accelerators, as well as system-level changes—such as operating system or instruction set architecture (ISA) innovations—that might be needed to realize the accelerators' potential. Unfortunately, existing simulation tools that can support such system-level research are limited by the lack of fast, scalable machine emulators to drive execution.

To fill this need, in this dissertation we first present a novel machine emulator design based on dynamic binary translation that makes the following improvements over the state of the art: it scales on multicore hosts while remaining memory efficient, correctly handles cross-ISA differences in atomic instruction semantics, leverages the host floating point (FP) unit to speed up FP emulation without sacrificing correctness, and can be efficiently instrumented to—among other possible uses—drive the execution of a full-system, cross-ISA simulator with support for accelerators.

We then demonstrate the utility of machine emulation for studying heterogeneous systems by leveraging it to make two additional contributions. First, we quantify the trade-offs in different coupling models for on-chip accelerators. Second, we present a technique to reuse the private memories of on-chip accelerators when they are otherwise inactive to expand the system's last-level cache, thereby reducing the opportunity cost of the accelerators' integration.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

CAS *compare-and-swap*. 36–39, 48, 52

LL/SC *load-locked/store-conditional*. 37–41, 50, 52

ASIC application-specific integrated circuit. 5, 12, 13

CLB configurable logic block. 12

DBI dynamic binary instrumentation. 8, 24, 54, 55, 61, 64, 69, 77, 78, 80

DBT dynamic binary translation. 7, 8, 14–16, 18, 23, 25, 27, 29, 30, 37, 45, 127, 128, 131

DMA direct memory access. 9, 82, 85, 86, 96

DSP digital signal processing. 12

DVFS dynamic voltage and frequency scaling. 4

FFT fast Fourier transform. 10, 87, 88, 93, 95, 97

FP floating point. 8, 19, 20, 53–58, 70, 72–74, 77, 78, 80, 128, 131, 132

FPGA field-programmable gate array. 5, 12, 13, 96, 104

FPU floating point unit. 8–10, 20, 53–58, 73, 80, 131

GPU graphics processing unit. 5, 8, 11–13, 97

HDL hardware description languages. 12

HLS high-level synthesis. 6, 12, 88, 89

HTM hardware transactional memory. 30, 40, 49, 50

IoT Internet of things. 7

IR intermediate representation. 15, 16, 18, 19, 61, 62, 64, 68

ISA instruction set architecture. 6–10, 13–15, 17, 19–21, 23, 24, 27, 29–31, 36, 37, 46, 48, 51, 53–55, 59, 61, 63–65, 68, 69, 72, 75, 78, 80, 82, 83, 91, 117, 127, 128, 130, 131

JIT just-in-time. 19

LCA loosely-coupled accelerator. 84–86, 91, 93–95

LLC last-level cache. 9, 82, 88, 94–96, 99–101, 106, 110, 113, 114, 116, 117, 119, 120, 122, 125, 126

MMU memory management unit. 17, 21

MRU most recently used. 35

NoC network-on-chip. 100, 111, 113, 114, 116–118, 120, 125

NUCA non-uniform cache architecture. 100, 102, 116, 126

OS operating system. 25, 91, 95, 117

PC program counter. 59

PLM private local memory. 8, 85–89, 93, 96, 99–107, 110, 112, 113, 118, 119, 121, 125, 126

RCU read-copy-update. 22, 31, 32, 35, 39, 40, 51, 59, 63

RTL register-transfer level. 6, 12, 87, 89, 91

SIMD single instruction, multiple-data. 11, 20

SMT simultaneous multithreading. 42, 49

SoC system on chip. 5, 12, 82, 96

softMMU software memory management unit. 17, 18, 20, 21, 27, 66, 67, 70, 77, 128, 129

TB translation block. 16, 18, 32–34, 59–65, 68, 74, 75

TCA tightly-coupled accelerator. 83–86, 91, 93, 94

TDP thermal design power. 3

TLB translation lookaside buffer. 17, 20, 21, 31–33, 40, 66–68, 70, 72, 132

Acknowledgments

Confía en el tiempo, que suele dar dulces salidas a muchas amargas dificultades.[1]

- Miguel de Cervantes, *Novelas exemplares - La gitanilla (1613)*

My first words of gratitude go to my advisor, Luca Carloni. He took me as a student when I was a decidedly unproven candidate, as well as stubborn and impulsive. With his characteristic bonhomie he deftly guided my work, first exposing me to the grit that research requires, and then nurturing my ambition to eventually let me pursue my own line of research, which expanded to areas outside of his main expertise. Furthermore, he fully backed my (at times perilously quixotic) quest of not only publishing papers, but also merging the resulting source code into a mature open-source project. Through our shared moments of sadness, joy, as well as countless discussions about research, life and *calcio*, I have become a more effective researcher and a more discerning person, and now I am fortunate to call him a friend.

Another important pillar of support during my time at Columbia has come from other members of the System-Level Design group. Upon my arrival Marcin Szczodrak, Young Jin Yoon, Michele Petracca, Nicola Concer, Francesco Leonardi and Hung-Yi Liu provided essential guidance and mentoring. Later additions to the group (YoungHoon Jung, Paolo Mantovani, Christian Pilato, Giuseppe Di Guglielmo, Johnnie Chan, Davide Giri, Luca Piccolboni, Jihye Kwon, Guy Eichler, Kuan-Lin Chiu) were key contributors to the group's fun and productive work environment, from

[1] *Trust time; it usually provides a sweet way out of many bitter challenges.*

which I greatly benefited. Paolo Mantovani stands out as an enduring friend and collaborator from the SLD group. I am indebted to him for ability to quickly dissect and refine my ideas, all while not losing context of the 1,000 other projects he always had his hands on.

Other Columbia graduate students and post-docs provided support, guidance and friendship. Kanad Sinha, Melanie Kambadur, Lisa Wu, Eva Sitaridi, John Demme, Robert Martin, Christoffer Dall, Andrea Lottarini, Yipeng Huang, Jared Schmitz, Lianne Lairmore, Joël Porquet, Tom Repetti, David Williams-King, Hiroshi Sasaki, Richard Townsend, Martha Barker and all other friends who made it to our *1020* gatherings were fundamental to help me cope with hard times, both personal and research-induced. Apart from a great friend and mentor, John Demme was also key in making me realize the importance of clear, concise writing.

Outside of Columbia, the QEMU community has been an extraordinarily fertile source of collaboration. Paolo Bonzini, Alex Benné, Richard Henderson and many other members of the QEMU community have given me enormous amounts of support, helping me solve challenging problems, reviewing and improving my papers and/or source code submissions, and eventually merging many of my contributions into upstream QEMU.

My final words of gratitude go to my mother, sister, brothers and nephews in Spain as well as my family in Japan for their tireless patience and support over all these years. Here in the US, my wife has been instrumental in keeping my sanity, always being a source of comfort and encouragement. In no small part this work is also hers.

最愛の妻へ。　*To my beloved wife.*

In memory of Matthieu Cattin, a true friend and role model.

Chapter 1

Introduction

Over the last two decades, the world that we knew as the child of the Industrial Revolution has been upended by the onset of the Information Age. The pervasive adoption of computing has led to drastic changes to service sectors which have altered our daily lives, for instance transforming the way we shop, travel, learn, interact, eat and even mate, as well as rapid changes to manufacturing, supply chains and international affairs, e.g. by redefining warfare and making computing knowledge and the control of its supporting technologies a geopolitical asset.

To a large extent, computing's meteoric rise from business tool to a key component of the economy is due to computing's sustained exponential performance increase since the 1960's. This increase rode on the back of two powerful horses: transistor scaling and advances in computer architecture. In the mid 2000's transistor scaling began slowing down, and since then most of the onus of delivering performance has shifted to computer architects. As a result, computer architecture is today a very different field than it was only a decade ago; its focus has shifted away from general-purpose processors to *heterogeneous systems*, that is, systems that, by integrating general-purpose cores with specialized hardware, are capable of efficiently providing high performance for specific applications. In the remainder of this chapter we cover in more depth the motivation behind the trend towards specialization, as well as the resulting challenges that this dissertation addresses.

1.1 Transistor Scaling

In 1965, Gordon Moore predicted the in the decade to follow there would be a doubling of components on a chip every 12 months [182]. By 1975, Moore's prediction had proven so prescient that by then it was referred to as *Moore's law*. That year, Moore revised his prediction for the following decades, lowering the doubling of components to every two years [183].

Remarkably, for the next four decades the semiconductor industry delivered on the promise of Moore's law. This stunning rate of progress in transistor miniaturization and integration has been a story of engineering achievement, but ultimately it is a story about the economics that supported that achievement. Indeed, Moore's prediction was not just that this exponential growth of chip complexity was physically attainable; this future was also predicted to be economically feasible.

In the last few years, the economics behind transistor miniaturization have become increasingly uncertain. Given the astounding capital investment that is now required to bring up a new fabrication process, less foundries are capable of taking on the challenge [37, 156], which results in per-transistor prices that are not lowering with further miniaturization as they used to [120], as well as longer times between technology jumps in order to amortize the capital investments [151].

An additional trend works against demand for smaller transistors. Below 65nm, transistors do not follow anymore the scaling model predicted by Dennard in 1974 [82]. Contrary to Moore's observation, Dennard's work derives from physics; it modeled how device dimensions and voltage could both scale to yield smaller and faster transistors while—crucially—keeping power density constant. Unfortunately, Dennard's scaling model does not apply at sufficiently low voltages and gate lengths: leakage current and quantum tunneling effects, whose impact was reasonably ignored in 1974, are now limiting voltage scaling and speed improvements [34]. As a result, appetite for smaller transistors that are less power efficient or slower than those in previous generations becomes harder to justify, particularly for general-purpose applications that are power constrained such as microprocessors.

Is Moore's law dead or at least slowing down, then? Conflicting news reports and

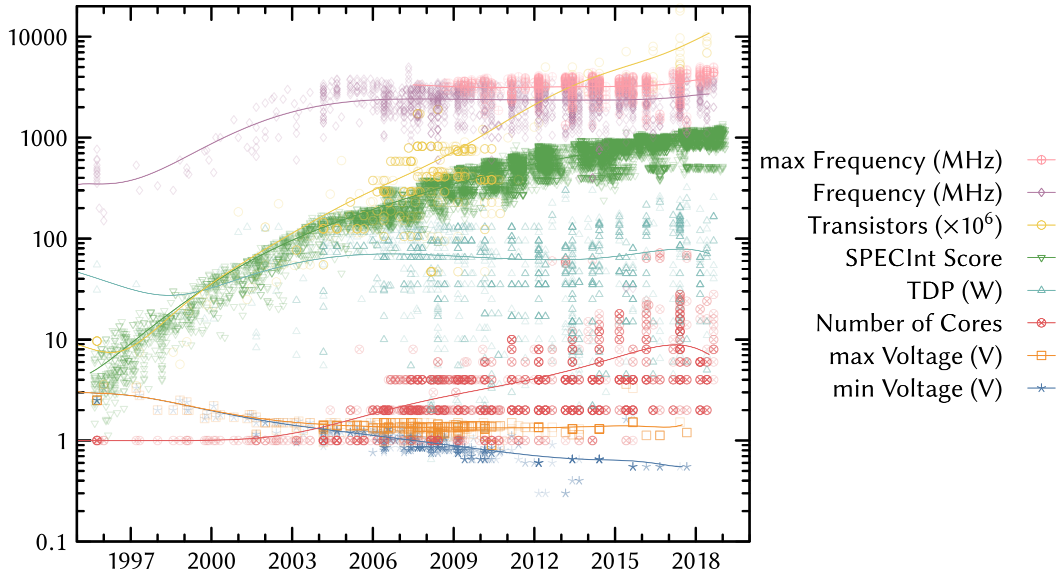


Figure 1.1: Microprocessor trends in the last two decades. For each set of data points, a smoothed yearly average is also shown. Scores from SPECInt 2000, 2006 and 2017 are converted to SPECInt95 scores using a conversion factor [74]. Data obtained from CPU DB [74], Intel ARK [2], SPEC [4] and Wikipedia.

speculation [219, 227] advise against answering this question with any certainty.[1] Consensus is, however, that "keeping up with Moore's law is harder than ever before" [92]. Nevertheless, as we discuss next it is the breakdown of Dennard scaling, and not the potential slowdown of Moore's law, what has brought a sea change to computer architecture.

1.2 The Rise of Multicores

The end of Dennard scaling in the mid 00's was a defining moment for the microprocessor industry. As shown in Figure 1.1, single-core performance (represented via SPECint's performance scores) stopped growing exponentially around 2004, while at the same time thermal design power (TDP), clock frequency and operating volt-

[1]Avoiding potential market tremors is partly to blame for the lack of clarity from foundries on scaling progress. Intel, for instance, stopped publishing transistor counts for their chips in 2011, which explains the paucity of post-2011 transistor count numbers in Figure 1.1.

age plateaued. The number of transistors on a chip continued increasing, however. Where did those transistors go? Some of them were expended to improve single-core performance. The majority of them, however, were used to increase the number of cores, giving rise to the era of multicores.

If power consumption remained constant, how was it possible to increase single-threaded performance (even if a lower rate than before) while turning on more cores? This was achieved through the introduction of dynamic voltage and frequency scaling (DVFS) to control and expand operating ranges at run-time, e.g. by giving maximum frequency to a single core while clocking down or turning off other cores, or by lowering all cores' frequencies when executing parallel workloads.

Over the last decade, processors have relied on increasing core counts to provide aggregate performance gains. For how long can multicore scaling continue? Two fundamental limits stand in the way. First, given that power envelopes are fixed and that post-Dennard scaling provides increasing power densities, scaling down a design results in an exponentially larger portion of the chip having to remain turned off ("dark silicon") [233]. This curtails the utility of microarchitectural improvements, whose performance—as determined by Pollack's rule[2]—can only scale with the square root of the number of transistors used. Second, Amdahl's law [13] caps the speedup that can be achieved through parallelization. At best, e.g. for embarrassingly parallel problems, the maximum achievable speedup grows linearly with the number of cores. This improvement, while non-negligible, is far from the exponential performance increases that were delivered under Dennard scaling.

When accounting for these limits, models show that multicore scaling, i.e. the addition of cores to benefit from process scaling, is only a valid short term strategy for the post-Dennard scaling era [93, 122]. Thus, in order to sustain performance gains, a shift from the multicore paradigm was called for.

[2] *Pollack's rule* is due to Fred Pollack, who observed that performance scales as the square root of design complexity. The CPU DB paper [74] corroborates Pollack's rule by comparing the performance of several generations of microprocessors against their transistor counts and normalized area.

1.3 To Specialize or Perish

As a response to the end of multicore scaling, architects have turned to specialization, giving rise to *heterogeneous* systems that integrate general-purpose cores with specialized *accelerators*. Accelerators are specialized hardware that can deliver superior energy efficiency and performance over general purpose processors. Among accelerators, application-specific integrated circuits (ASICs) offer the highest degree of specialization, which results in higher cost, complexity and—crucially—performance and energy efficiency over the alternatives, e.g. graphics processing units (GPUs) or field-programmable gate arrays (FPGAs) [56, 147]. The use of ASICs for acceleration was initially adopted at scale by designs for high-volume systems on chip (SoCs), such as those destined for the mobile market. However, the recent mobile/cloud computing bifurcation has also brought accelerators to the server room, with numerous examples in diverse applications, e.g. convolution [198], databases [141, 242], graph analytics [107, 191], neural networks [52, 126], speech recognition [245] and video encoding [108].

The shift towards greater heterogeneity via ASIC accelerators is at least due to three reasons. First, compared to the alternatives such as abandoning silicon, giving up on further miniaturization, or restricting the use of additional transistors to under-clocked components [228], specialization can deliver order-of-magnitude improvements in both energy efficiency and performance and does not require traumatic changes from the semiconductor industry. Second, in a future where chips are dominated by dark silicon, specialization is a natural fit: increasing dark silicon should lead to commensurate increases in the number of integrated accelerators, which can be turned on and off at run-time to suit the workload's needs. Third, economic factors also play in specialization's favor: even if Moore's law slows down or stops altogether, accelerators can be implemented using older technology nodes and still yield significant efficiency and performance, at a fraction of the cost in both fabrication masks and design complexity (e.g., through simpler design rules) [134].

1.4 Challenges in Heterogeneous System Emulation

The staggering rate of progress in computer systems' performance over time is largely owed to quantitative methods, which are used to motivate as well as evaluate research ideas [80]. In particular, simulation is the most prevalent quantitative method, since it can capture a degree of complexity that analytical models cannot.

High-performance simulators are typically functional-first, that is, they employ an instrumented emulator whose execution of a workload feeds a timing model [91]. Emulators are therefore a key component of simulators, since they determine what workloads can be modeled and provide a lower bound to simulation time.

The emulation of heterogeneous systems poses a new set of challenges that existing emulators cannot fulfill. These challenges are related to accelerator modeling, full-system emulation, instruction set architecture (ISA) diversity, and the delivery of high performance without sacrificing correctness.

Accelerator Modeling. In heterogeneous systems, most of the performance and energy efficiency gains come from the use of accelerators. Thus, emulators must be capable of modeling accelerators that interact with other components of the system, for instance other accelerators or general-purpose cores. The modeling of accelerators can occur at different points in the accuracy vs. performance spectrum, e.g. from a simple back-of-the-envelope latency model, to interacting with external engines that simulate designs at register-transfer level (RTL) or written in languages suited for high-level synthesis (HLS).

Full-system Emulation. How to efficiently and securely integrate accelerators within the rest of the system is an open research question. Solutions are likely to cut across the hardware-software interface, for instance by modifying the system's virtual memory mechanism. Further, emulators are also likely to accelerate applications that largely execute within the operating system's kernel, such as memory allocation and I/O-intensive applications (e.g., storage, network). Evaluating solutions to these issues will therefore require full-system emulation.

ISA diversity. Emulators typically run on server machines, where the x86 ISA dominates today. The heterogeneous systems to be emulated, however, are not only server-like. For instance, they could be mobile systems (which typically implement the ARM ISA) or Internet of things (IoT) systems (dominated by ARM and other RISC ISA's such as MIPS and, increasingly, RISC-V). Thus, simulating these systems will require cross-ISA emulation and instrumentation, since the host and target ISAs are likely to be different. Further, the open-source nature of ISAs such as RISC-V is likely to spark ISA innovations [16] whose prototyping and evaluation will also require cross-ISA support.

Performance and correctness. To enable productive analysis of workloads that span across multi-cores and accelerators, emulators must provide the above features with high performance and correctness. Thus, emulators must be able to (1) reduce the existing single-threaded performance gap between cross-ISA and same-ISA emulators, and (2) leverage multi-core hosts to perform parallel emulation of accelerators and multi-core guests, while reconciling guest and host differences in memory consistency model and atomic instruction semantics to maintain correctness.

1.5 Thesis

> *Fast, scalable machine emulation is feasible and useful for evaluating the design and integration of heterogeneous systems.* □

1.6 Contributions

To support the above thesis, we present contributions that improve the speed and scalability of machine emulation and reduce the cost of accelerator integration.

On the emulation side, we make the following contributions:

- The design of a machine emulator based on dynamic binary translation (DBT) that scales for multi-core guests while remaining memory efficient via the use of a shared code cache. [66, 67]

- An approach to reconciling the difference between guest and host atomic instruction semantics to enable correct and scalable cross-ISA emulation of multi-core guests. [66]

- A technique to improve the performance of cross-ISA DBT of floating point (FP) instructions by leveraging the host's floating point unit (FPU). [67]

- An ISA-agnostic instrumentation layer that converts a cross-ISA DBT engine into a low-overhead cross-ISA dynamic binary instrumentation (DBI) tool. This cross-ISA tool supports state-of-the-art DBI features such as instrumentation injection at the granularity of individual instructions, and has comparable performance to state-of-the-art, same-ISA tools when used for complex instrumentation workloads such as cache simulation. [67]

The feasibility of these contributions is supported by their implementation and evaluation. The implementation of all the above intellectual contributions has been integrated into version 4.0 of the open-source QEMU emulator[3], except that of the instrumentation layer, which at the time of writing remains under review by the QEMU community.

On the heterogeneous system design and integration side, we make the following contributions:

- A quantitative comparison of different approaches to the design and integration of accelerators for high-throughput applications that are irregular, i.e. not amenable to acceleration via GPUs or vector processors. A salient conclusion of this study is that working sets of non-trivial size are best served by loosely-coupled accelerators that integrate private local memories (PLMs) tailored to their needs. [69]

- A technique to exploit accelerator PLMs to reduce the opportunity cost of integrating on-chip accelerators by transparently exposing accelerator PLMs to

[3]http://www.qemu.org/

the cache substrate, thereby extending the system's last-level cache capacity when accelerators are not in use. [68, 70]

These two contributions have been evaluated through simulation based on scalable machine emulation, which highlights the latter's usefulness as a vehicle for heterogeneous systems' research.

1.7 Outline

We begin the rest of this dissertation by discussing background and related work in Chapter 2.

We then describe our work on achieving fast and scalable cross-ISA emulation, which we cover in two parts. First, Chapter 3 presents Pico, a cross-ISA emulator based on QEMU that (1) allows us to explore the performance vs. correctness tradeoffs in cross-ISA emulation of atomic instructions, and (2) scales for most parallel guest workloads, i.e. those that do not show high rates of parallel code translation. Second, Chapter 4 presents Qelt, a design that improves over Pico's by (1) also scaling for multi-core guests that show high rates of code translation, (2) improving cross-ISA floating point emulation speed by leveraging the host FPU, and (3) adding instrumentation support via an ISA-agnostic layer.

Next we study the design and integration of heterogeneous systems through simulation. Chapter 5 describes a comparison of three models of accelerator coupling: tight coupling behind a CPU, loose out-of-core coupling with direct memory access (DMA) to the last-level cache (LLC), and loose out-of-core coupling with DMA to DRAM. Chapter 6 presents Roca, a technique to lower the opportunity cost of accelerator integration by reusing their local memories to expand a non-uniform LLC when they are otherwise unused.

Chapter 7 considers future directions for this work. Chapter 8 concludes.

Chapter 2

Background and Related Work

In this chapter we provide a foundation for the rest of this dissertation. We first cover background material on accelerators, to then discuss the state of the art in machine emulation and instrumentation.

2.1 Accelerators

An accelerator is "a specialized hardware unit that performs a set of tasks with higher performance or better energy efficiency than a general-purpose processor" [217]. Given this broad definition, confusion is perhaps inevitable when discussing accelerators, since very disparate hardware implementations share the same name.

Shao and Brooks [217] provide a useful taxonomy of accelerators. They classify accelerators across two dimensions: coupling and granularity.

- *Granularity* refers to the size of the computation that is assigned to accelerators, which determines their generality. Thus, *instruction-level* accelerators (e.g., an FPU or vector unit) merely extend an ISA and therefore are the most general; *kernel-level* accelerators are more specific, since they speed up a particular algorithm or kernel such as a fast Fourier transform (FFT); *application-level* accelerators, whose aim is to speed up entire applications (e.g., neural networks [50, 52] or key-value stores [78]), are the most specialized.

- *Coupling* refers to the distance between the accelerator and the general-purpose processor. Thus, *loosely* coupled accelerators are connected to the processor via the on-chip interconnect or off-chip memory bus, and *tightly* coupled accelerators are either part of the processor or attached to its cache. Chapter 5 presents a quantitative comparison of the two coupling models.

Ultimately, the choice of a particular accelerator platform results from considerations regarding flexibility, performance, area, energy efficiency and cost. Based on those parameters, we now discuss popular accelerator platforms, listed from more to less flexible.

Vector processors. Processors that support the execution of instructions that operate on one-dimensional arrays (*vectors*). Modern CPUs implement single instruction, multiple-data (SIMD) registers that are 128-bit, 256-bit, or 512-bit wide. SIMD can be explicitly generated via compiler intrinsics, or under certain conditions emitted automatically by the compiler. Algorithms that are heavy on control flow are not good candidates for vectorization. In some cases, however, control flow can be transformed into data flow to then execute on vector units. For example, Polychroniou et al. [194] demonstrate this approach for database operators to deliver up to an order-of-magnitude performance improvement over previous scalar and vectorized implementations.

GPUs. Processors optimized for arithmetic throughput, memory bandwidth and parallelism. Originally developed for graphics applications, GPUs are now a common accelerator for compute-intensive workloads, such as linear algebra, signal processing, and image/video processing. GPUs offer programming flexibility through the abstraction of GPU internals via mature software runtimes such as OpenCL or CUDA. GPUs can significantly outperform general-purpose processors for highly-parallel applications given GPU's superior parallelism, memory bandwidth and ability to hide memory latency [119, 128].

FPGAs. Integrated circuits that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects, thereby allowing reconfiguration. This feature makes FPGAs eminently general-purpose. Flexibility-wise, however, FPGAs are inferior to CPUs and GPUs: reprogramming an FPGA is more complex and time-consuming than updating software, and the development process of FPGA is less mature, both in the user-friendliness of hardware description languagess (HDLs) and in the necessary tooling. Fortunately, recent developments on the productivity side are promising. First, methodologies for rapid, HLS-based design space exploration of accelerators have emerged [62, 168]. Second, languages and runtimes specifically tailored to FPGAs can deliver simpler design and deployment processes than RTL currently does [17, 127, 142]. Third, the availability of FPGAs as part of affordable SoCs [71] or via cloud platforms has considerably lowered the barrier of entry for users, which is likely to result in stronger demand and therefore faster progress on the productivity front.
Compared to GPUs, FPGAs typically have lower performance for tasks that are a good match for GPUs due to the FPGA's lower clock rate and lower computation density. FPGAs, however, have higher energy efficiency, which has made them find a place in today's power-constrained data centers [197]. Furthermore, future FPGA designs might be able narrow the performance gap with GPUs for popular applications such as neural networks, for instance by increasing the amount of on-chip memory and digital signal processing (DSP) units [190].

ASICs. Integrated circuits customized to specific applications. ASICs can deliver the highest performance and energy efficiency among accelerators, albeit at the expense of flexibility (unlike FPGAs, ASICs cannot be reconfigured after fabrication) and—crucially—cost. The latter downside is the biggest hurdle against wider ASICs adoption. Although cheaper alternatives to the current ASICs fabrication process have been proposed [137], cost of development and fabrication remains today the dominant factor in determining the viability of ASIC-based accelerators [134]. As a result, ASICs have traditionally been restricted to high-volume markets such as mobile SoCs.

Of all the factors that guide the choice of an accelerator platform, cost is arguably the most important one. Thus, as demand for a particular application increases, higher (and more costly) specialization is to be expected. An illustrative example of this progression is described by Taylor [229], who summarizes the evolution of Bitcoin-mining hardware. Initially, adoption was low and therefore CPUs were enough to *mine* coins profitably. As the price of bitcoin raised, however, more players entered the scene which led to an accelerator "arms race" towards higher energy efficiency. Predictably, *miners* quickly moved from CPUs to GPUs, then to FPGAs, and finally to ASIC accelerators, with an order-of-magnitude energy efficiency improvement at each transition.

With the rising importance of energy-efficient computing, we are likely to see similar trajectories in highly competitive markets. For instance, the recent explosion in demand for machine learning (ML) computation has fueled an "ML arms race", and consequently ML accelerators have started populating the data center [126, 164, 212].

2.2 Machine Emulation

A machine emulator is a hardware or software artifact that enables one system (the *host*) to behave like another system (the *guest*). The term "emulator" originated in 1964 at IBM, where engineers extended the ISA of the then upcoming IBM "System/360" computer to support the ISA of the "1401", an older IBM machine. The ISA extension allowed 1401 code to be *emulated* directly on the 360, while running at least four times faster than on the 1401 itself [196].

Machine translators have three core components: a translator, a dispatcher and a device emulator. The translator translates guest instructions into host instructions. The device emulator models guest system peripherals (e.g., timers, interrupt controllers, I/O devices), or executes system calls directly on the host in *user-mode* emulation. The dispatcher controls the guest's execution state and mediates between the device emulator and the translator, e.g. by deciding when to execute translated code or whether to exit execution due to a guest exception.

In the remainder of this section we first cover dynamic binary translation, a key technique for high-performance translators. We then examine the main sources of inefficiency in machine emulators, discuss the challenges in scaling parallel cross-ISA emulation, review the state of instrumentation tools, and conclude by describing how instrumentation is used for computer architecture simulation.

Most of our discussion is based on QEMU [24], a popular machine emulator and virtualizer. QEMU's popularity is partly due to being (1) cross-ISA and (2) portable. That is, (1) it supports many guest-host ISA combinations, and (2) can be ported to support additional guest/host ISAs with reasonable engineering effort. Although centered on QEMU[1], our discussion applies to portable, cross-ISA emulators at large.

2.2.1 Cross-ISA Emulation via Dynamic Binary Translation

Dynamic binary translation (DBT) is a binary recompilation technique by which sequences of *target* (i.e. guest) code are translated and cached at run-time[2] for subsequent execution on the host. The typical granularity of DBT applies to a basic block, i.e. a linear sequence of instructions with a single entry point and a single exit point. The Shade translator [60] was first to incorporate key optimizations for DBT engines, such as caching, indexing and chaining of blocks of translated code. By leveraging these optimizations DBT amortizes the cost of translation, thereby outperforming alternatives such as using an interpreter [111].

The applications of DBT are numerous. For instance, DBT has been used in software systems for performance optimization [18, 51, 136, 159, 223, 248], code manipulation [38, 187, 214], energy efficiency optimization [243], parallelization [144, 244], cross-ISA execution [20, 53, 76, 79, 90], control flow monitoring [138, 199] and virtualization without hardware support [41, 42]. In Section 2.5 we cover in more detail the application of DBT for machine emulation/simulation and instrumentation.

[1]Henceforth we refer to QEMU in the present tense to describe the state of QEMU prior to the work presented in this dissertation, most of which is now part of QEMU.

[2]The run-time aspect is an important one for emulation. An alternative is to perform static binary translation. However, compile-time analysis is insufficient to support dynamic linking or self-modifying code, and makes discovering code accessed via indirect jumps problematic [86]. For this reason, static binary translators usually rely on additional run-time analysis, typically via interpreta-

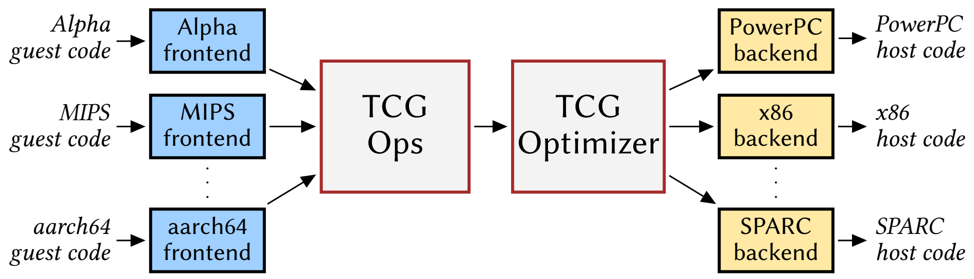


Figure 2.1: Cross-ISA portability is achieved in QEMU by leveraging the TCG IR.

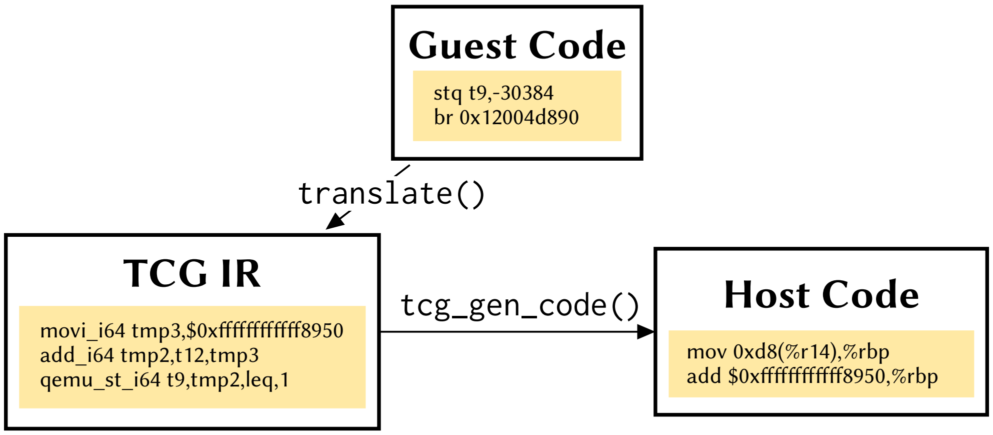
Cross-ISA DBT engines differ from same-ISA ones in that the host and guest ISAs are different. Within cross-ISA engines, *portable* ones differ from *fixed* cross-ISA engines (e.g., EL [20], FX!32 [53]) in that they support more than one host ISA. This is typically achieved by using an intermediate representation (IR) to mediate between target and host ISAs. QEMU [24] and LLVM [152] represent the state of the art in portable cross-ISA engines. LLVM is a language-independent optimizer and code generator, and therefore can generate high-quality code. The downside of this high-quality code generation is the cost it incurs, which complicates the use of LLVM in dynamic binary translators, as we discuss in Section 2.2.3.2. In contrast, QEMU only performs simple optimizations, which results in faster translation at the expense of code quality. Figure 2.1 depicts the use of TCG, QEMU's intermediate representation (IR), to achieve cross-ISA portability.

2.2.2 Key Emulator Structures and Concepts

To better frame our discussion in the remainder of this chapter, we now define some key emulator concepts.

User-mode and full-system emulation. User-mode emulation applies DBT techniques to target code, but executes system calls on the host. While user-mode emulation can employ a system call translation layer, for example to run 32-bit programs on 64-bit hosts or vice versa, it is generally limited to programs that are compiled for

tion [57].



```
stq t9,-30384
br 0x12004d890
```

```
movi_i64 tmp3,$0xffffffffffff8950
add_i64 tmp2,t12,tmp3
qemu_st_i64 t9,tmp2,leq,1
```

```
mov 0xd8(%r14),%rbp
add $0xffffffffffff8950,%rbp
```

Figure 2.2: An example of portable cross-ISA DBT. An Alpha basic block is translated into an x86_64 TB using QEMU's TCG IR.

the same operating system as the host. Full-system emulation refers to the emulation of an entire system: target hardware is emulated, which allows DBT-based virtualization of an entire target guest. The guest's ISA and operating system are independent from the host's.

Virtual CPUs (vCPUs). In *user-mode*, a vCPU corresponds to a guest's thread of execution. In *full-system* mode, a vCPU represents a core of the guest CPU.

Translation block (TB). As shown in Figure 2.2, a target basic block is translated to form one or more translation blocks (TBs). Guest execution is thus emulated as a traversal of connected TBs. Note that while most basic blocks are translated into a single TB, the emulator can also break a basic block into several TBs, for example to force an exit to the device emulator in order to update hardware state.

Helpers. Sometimes the IR is not rich enough to represent complex guest behavior, or it could only do so by overly complicating target code. In such cases, *helper* functions are used to implement this complex behavior outside of the IR, leveraging the expressiveness of a full programming language. Helpers are oftentimes used to emulate instructions that interact with hardware state, such as the memory management unit's translation lookaside buffer (TLB).

Guest RAM. Fast RAM accesses are a requirement for performance in a DBT; memory accesses are exceedingly common. Synchronization between accesses to the guest RAM is generally handled by the program running in the guest; the emulator only needs to provide a correct implementation of the target architecture's memory model (e.g., memory ordering, atomic instruction semantics). This problem is covered in detail in Section 3.3.

Devices. Emulated devices, which are only present in full-system mode, are managed by the device emulator. Device emulation is less of a performance concern than binary translation, since device accesses are (1) significantly more rare than RAM accesses and (2) often serialized, resulting in access times orders-of-magnitude slower than RAM accesses. Performance aside, the goal of the device emulator is to support the modeling of as many hardware devices as possible. QEMU is arguably the leading tool in this aspect. For this reason, QEMU's device emulator is used for both cross-ISA emulation as well as full-system, same-ISA virtualization with hardware support, e.g. by leveraging KVM [139, 73].

Software memory management unit (softMMU). The softMMU emulates the guest's memory management unit (MMU) in software. Among other functions, the softMMU provides guest virtual-to-physical address mappings as well as guest-to-host virtual address mappings. To achieve this efficiently, a softMMU typically generates code to emulate guest memory accesses in two steps: first, the corresponding host virtual address is looked up from an array that acts as a software translation lookaside buffer (TLB) and is therefore indexed by the guest virtual address; second, assuming the host address was found, the access to guest RAM is performed on it.

2.2.3 Sources of Machine Emulation Overhead

In this section we discuss the main sources of inefficiency in portable cross-ISA machine emulators. These are: indirect branch handling, quality of the generated code, floating point emulation, and the overhead of the softMMU.

2.2.3.1 Indirect branch handling

The handling of indirect branches is a key component to the performance of a DBT engine [116]. Indirect branches cannot be optimized via regular TB chaining, since the destination TB, unlike for regular branches, is not known at translation time. The best known solution to minimize indirect branch overhead is the use of *traces* as introduced in Dynamo [18]. A trace is a sequence of hot TBs that are grouped together to form a single-entry, multi-exit unit of execution.

Unfortunately, the applicability of trace compilation to full-system emulators is limited; even for direct jumps, the optimization is constrained to work only across same-page TBs, for otherwise the validity of the jump target's virtual address cannot be guaranteed without querying at run-time the softMMU. An approach better suited for full-system emulators is the use of caching [213], although adapted to full-system emulation like Embra's *speculative chaining* [241]. This approach is demonstrated on QEMU by Hong et al. [117]. They first add a small cache to each vCPU thread to track cross-page and indirect branch targets, and then modify the target code to inline cache lookups and avoid most costly exits to the dispatcher. In Section 4.2.4.2 we present a similar approach that extends QEMU's IR to abstract the necessary cache lookups.

2.2.3.2 Code Quality

Some DBT engines such as QEMU prioritize the speed of code generation over the quality of the generated code. This can result in significant emulation overhead, particularly for workloads that have regions of *hot* (i.e. frequently-executed) code. A common solution is therefore to apply aggressive optimization to those hot regions, which are detected via a small amount of profiling [53]. HQEMU [118] implements this approach for QEMU by leveraging LLVM to compile not just hot TBs, but entire traces of blocks (i.e. sequences of TBs, as described above). The use of traces instead of TBs gives the compiler greater potential for optimization, while also increasing the number of compilation jobs and size of the generated code. Thus, to minimize the performance impact, HQEMU offloads the compilation jobs to other helper threads in the host. A similar approach is followed by Böhm et al. [33] in an LLVM-based emulator.

An alternative for generating higher-quality code is to use techniques that do not rely on IR manipulations. HERMES [249] is a QEMU-based system that adds an additional host-specific optimization pass *after* the TCG backend's stage. Bansal and Aiken [19] use peephole superoptimizers to generate translation rules, which are used for inexpensive yet high-quality code generation. They exhaustively search all possible rules of translation for up to several instructions in length, which for complex ISAs can be time consuming. Wang et al. [235] propose a different rule-based approach that is also automated yet can generate rules at a faster rate. Their system works by first analyzing source code and the corresponding binaries compiled for both ISAs. Then, potential rules are produced and validated using equivalence checking.

The use of expensive translation techniques can be amortized with persistent code caching [39, 112, 202]. Wang et al. [236] present such a system based on QEMU. Their framework handles complicated use cases for persistent code caching, such as guest code generated by just-in-time (JIT) engines and relocatable guest binaries.

2.2.3.3 Floating Point Emulation

Faithful emulation of floating point (FP) instructions is more complex than just generating the correct FP result. Correct emulation requires emulating the entire *floating point environment.* Apart from generating the right result, this includes the modeling of hardware state that configures the behavior of the FP operations (e.g., rounding mode) and keeps track of FP flags (e.g., invalid, divide-by-zero, under/overflow, inexact), optionally raising exceptions in the guest as the flags are set.

The FP environment is defined in the specification of each ISA. Despite the compliance of many commercial ISAs with the IEEE-754 standard [7], emulation remains non-trivial for several reasons: (1) the standard leaves details to be decided by the implementation (e.g., underflow tininess detection or the raising of flags in certain scenarios), (2) some features are not part of the standard, even though they might have wide adoption (e.g., flush-to-zero and denormals-are-zeros), and (3) some widely used implementations are not compliant with the standard (e.g., ARM NEON [15]).

Due to this diversity, IRs (such as LLVM and TCG) do not represent FP-environment-related features. Thus, portable cross-ISA emulators trade performance (e.g., 2× slow-down for QEMU [24]) for portability by invoking soft-float[3] emulation code via helpers.

We present in Section 4.2.1 an approach whose goal is to recover most—if not all—of this performance loss by leveraging the host FPU for the vast majority of guest FP instructions. Similarly to our work, Guo et al. [104] leverage the host FPU to emulate guest FP instructions. They, however, employ a considerable amount of soft-float operations in order to handle all possible corner cases (e.g. due to different operands, flags and rounding modes). Our approach puts greater emphasis on performance: it identifies a *fast path* (or *common case*) that can be accelerated with a minimum amount of auxiliary code, and defers all other (unlikely) cases to a *slow path* entirely implemented in soft-float.

An orthogonal issue that affects FP emulation performance in cross-ISA translators is the translation of vectorized (i.e. SIMD) code, which is frequently found in FP-heavy workloads. Recent work [99, 157, 180] has tackled this problem for portable cross-ISA emulators, although in an incomplete fashion due to a focus on performance and not on emulating the entirety of the guest FP environment. Only Guo et al. [104] do perform faithful emulation of the FP environment when dynamically translating SIMD code, which they demonstrate by emulating ARM Neon and VFP extensions in an LLVM-based cross-ISA emulator. Most of the remaining work in this area tackles the problem of dynamically rewriting of same-ISA vector code, for instance to convert scalar loops to SIMD instructions [131, 244], or to allow vector code compiled for older ISA specifications to take advantage of subsequent SIMD additions to the ISA [106, 175].

[3]Soft-float implementations, e.g. Hauser's softfloat/testfloat packages [110], faithfully emulate the guest's FP environment by exclusively using integer instructions.

2.2.3.4 Memory Management Unit (MMU) Emulation

Efficient, portable implementations of a softMMU were pioneered by the Embra [241] and Simics [165] simulators. The authors of these systems also identified that the size of the TLB in the softMMU does not have to match that of the simulated system; it can be made larger to gain a performance improvement for memory-hungry workloads. QEMU [24] implements a similar softMMU. Due to being open source and supporting many different target and host ISAs, since its release QEMU has served as the main evaluation platform for further research on MMU emulation.

Tong [231] et al. present an extensive study on how QEMU's softMMU can be optimized. Among several enhancements, they propose dynamic resizing of the softMMU, an idea that we extend in Section 4.2.4.1 with the consideration of TLB use rates in the recent past, which yields significant speedups for memory-hungry workloads (Section 4.3.2).

Same-ISA virtualization has benefited from hardware support for low-overhead MMU emulation [9, 72, 176]. Several approaches that leverage these architectural extensions for virtualization have recently emerged to support cross-ISA MMU emulation. They range from keeping shadow page tables using the host's virtual memory support [238], to virtualized page tables [48, 96] to deploying a hypervisor under which to run the emulator [222]. Unfortunately, all of these approaches require the host's virtual address length to be greater than the guest's. It is unclear whether this limitation can be overcome without sacrificing performance.

2.3 Parallel Cross-ISA Machine Emulation

The increasing core counts in machines with diverse ISAs—from embedded systems to servers—calls for efficient, parallel cross-ISA emulators. Unfortunately, QEMU is not well-equipped for the task: QEMU-user spends large amounts of time sleeping on locks and stops all CPUs every time an atomic operation executes; QEMU-system is only parallel when used in conjunction with KVM as a device emulator, otherwise it executes the emulated processors in a round-robin fashion in a single host thread.

The main design choice when developing a scalable emulator is whether to equip each guest CPU with a private translation code buffer, or to share cached translated code among all executing CPUs. A comparison of shared vs. private code caches was conducted by Bruening et al. [40] using DynamoRIO as the underlying translation system. According to the authors, a private code cache is simpler to manage due to the absence of synchronization for most common operations. However, private code caches have a major downside in their potentially egregious memory consumption: while desktop workloads typically share very little code among threads, typical server workloads (e.g. web servers, databases) spawn hundreds of threads/processes that execute large amounts of shared code, either due to heavy interactions with the kernel (e.g. processing of heavy I/O or network traffic) or due to the high-level languages these workloads are written in (e.g., Java) [97].

Pico's shared code cache design (Section 3.2) is made possible by read-copy-update (RCU), a pattern for efficiently accessing *read-mostly* data structures [174]. RCU improves scalability by allowing reads to occur concurrently with updates. Whenever a writer wants to free data, it has to wait until all readers exit their current critical section. This way, RCU provides *lifetime guarantees* during a read-side critical section. On the other hand, reads are *non-repeatable*. To limit the impact of non-repeatable reads, writers create new copies of the data structures they update. As long as readers do not repeat *pointer* reads, they will always see consistent data. The data, however, may potentially be stale. This may limit the applicability of RCU, but this technique has nevertheless seen extremely wide usage in the Linux kernel [173]. Readers, in addition to running concurrently with writers and other readers, only incur a very small overhead. User-space implementation of RCU typically require readers to note whether they are running inside an RCU-protected region and, possibly, wake up writers that are waiting for concurrent readers. However, in the common case when no writer is waiting, an RCU reader will only access thread-private data, thereby not causing any cache line bouncing. The implementation we used for Pico/Qelt (Chapters 3 and 4) also needs a memory barrier at each end of the RCU critical sections, though it is possible to use more complex schemes that eliminate the barrier. [84]

Two works in the literature address the problem of adding multicore support to QEMU: COREMU [237] and PQEMU [87]. COREMU uses private code caches, whereas PQEMU's authors experimented with both private and shared code caches. As an optimization, PQEMU supports a complex state machine to manage the shared code cache.

Neither PQEMU nor COREMU (including a later port for MIPS hosts [125]) address correctness issues specific to cross-ISA emulation. These issues arise due to differences between guest and host in their (1) memory consistency models and (2) atomic operation semantics. The first issue was recently addressed by Lustig et al. [161]. The authors model the insertion of memory barriers between memory accesses as a state machine; their ArMOR framework accepts a description of memory consistency[4] models for the guest and the host, and produces such a state machine for use in an interpreter or dynamic binary translator. The second challenge, which to our knowledge has no prior work in the literature, has to do with the correct and scalable emulation of *load-locked*/*store-conditional* pairs (*LL/SC*) on hosts that only provide a *compare-and-swap* (*CAS*) instruction which, unlike *LL/SC*, is subject to the *ABA problem* [177]. Section 3.3 describes our solution to these issues.

Pico's shared-code-cache design scales for full-system multicore guests during code *execution*. However, if the multicore guest performs parallel code *translation* (e.g., by executing large amounts of previously untranslated code, which is common when running parallel compilation jobs), scalability suffers because translation is serialized with a coarse-grained lock as done in PQEMU [87]. Full-system translators with thread-private caches (e.g., QSim [132], Parallel Embra [150], COREMU [237]) can trivially scale during parallel code translation, yet as mentioned above this can result in prohibitive memory usage [40]. Qelt, which we present in Section 4.2.2, is, to our knowledge, the first DBT engine for full-system emulators to scale during parallel code generation and chaining while maintaining a shared code cache.

[4]Our understanding of cross-ISA issues is shaped by work in the area of modeling and validation of memory barriers and atomic instructions, in particular Sarkar et al.'s work on modeling POWER and ARM multiprocessors [210] and Alglave et al.'s work on validating those models on real hardware [11].

2.4 Instrumentation

DBT is the basis for dynamic binary instrumentation (DBI) tools, such as DynamoRIO [38], Pin [160] and Valgrind [187]. DBT is well-suited for code instrumentation: it enables the handling of unmodified binaries (thus removing the need for recompilation/re-linking) while covering all executed user-space code without requiring access to the original sources. The overhead of DBI tools is usually low, most of it coming from the analysis performed and not the instrumentation. For instance, Valgrind is significantly slower than Pin or DynamoRIO, in part due to the maintenance of a heavy-weight shadow memory.

A major limitation of the above tools is the lack of support for cross-ISA analysis, therefore requiring the instrumented binary's ISA to match that of the host machine. QEMU does not yet provide instrumentation capabilities, which has sparked the development of multiple QEMU-based instrumentation tools.

A popular QEMU fork is the Unicorn framework [188], which adds an instrumentation layer around QEMU's DBT engine. Unfortunately, Unicorn is not an instrumentation tool: it cannot emulate binaries or work as a machine emulator, since those features were removed when forking QEMU. Unicorn is therefore suitable for being embedded into other applications, or for reverse engineering purposes. PEMU [247] and QTrace [230] implement a rich instrumentation and introspection interface for x86 guests, although they incur high overhead. TEMU [220], PANDA [88] and DECAF [113], of which the latter two are cross-ISA, implement security-oriented features such as taint propagation at the expense of large performance overhead. Of the above QEMU-based tools, only QTrace allows for instruction-level instrumentation injection. QTrace achieves this by shadowing guest memory and registers, which might be a contributor to its low performance. In contrast, Qelt's injection model is simpler, works entirely at translation time, can instrument helpers and has negligible performance overhead.

Similarly to Qelt, DBILL [163], PIRATE [240] and PANDA [88] can instrument helpers, although by leveraging the LLVM compiler to convert the original helper code into an instrumentable IR [54]. This is a flexible approach, but incurs high translation overhead and complexity. QEMU-based instrumentation was combined with debugging and introspection capabilities in QVMII [89], which optionally provides deterministic record-and-replay execution at the expense of performance.

2.5 Simulation via Instrumented Emulation

Instrumented emulation has been part of simulation almost since the introduction of emulators. At SOSP'79, Canon et al. [45] presented an emulator used for simulation purposes. They established two simple requirements in order to make emulation suitable for performance evaluation: *"(1) a consistent virtual time base, independent of real time, must be established, and (2) all virtual clock values and the timing of asynchronous virtual events must be derived from the virtual time."* Execution-driven simulators are thus implemented in a two-pronged approach: the emulator (sometimes called *functional simulator*) drives execution, and the timing model (which interacts with the emulator via the instrumentation layer) controls virtual time.

We now review the state of the art in user-mode and full-system simulation.

2.5.1 User-mode Simulation

Whether to support user-mode or full-system simulation is a decision that is based on the type of workloads to be run. If the guest workload spends most of the runtime in user-mode, a user-mode emulator might be accurate enough to drive simulation. Given the complexity of computer systems, however, it is hard to provide accuracy guarantees for user-mode simulators, since the operating system (OS) can still have an impact even if it executes rarely. For instance, the OS' handling of virtual memory and its use of cache-related instructions can have significant impact in performance [44]. These effects are only exacerbated in multicore architectures, which can result in virtual memory scalability limits [59] and scheduling decisions that greatly affect performance [36, 158].

Despite these shortcomings, user-mode simulators are a popular research tool. Table 2.1 shows a comparison of user-mode emulators in the literature, sorted chrono-

Name	Pub. date	Emulator	Speed	Scalability	ISAs
SimpleScalar [43]	1997	Interpreter	Low	None	several
FastSim [211]	1998	Memoization	High	None	MIPS
SESC [204]	2004	Interpreter	Low	None	
Graphite [181]	2010	Pin (DBT)	High	High	x86-on-x86
SlackSim [49]	2010	Interpreter	Med	Low	PISA
Sniper [47]	2011	Pin (DBT)	High	High	x86-on-x86
HORNET [203]	2012	Interpreter	Med	Low	MIPS
Zsim [209]	2013	Pin (DBT)	High	High	x86-on-x86
ESESC [14]	2013	QEMU (DBT)	High	High	ARM guests
PriME [100]	2014	Pin (DBT)	High	High	x86-on-x86

Table 2.1: Comparison of user-mode architectural simulators: emulation engine, speed, scalability on multicore hosts, and supported ISAs.

logically. Over time, higher simulation speeds have been achieved by (1) adopting DBT for faster emulation and (2) leveraging multicore hosts to emulate multicore guests. These changes have only been possible by raising the *level of abstraction*, that is, by phasing out cycle-level simulation (as in SimpleScalar [43] or SESC [204]) for coarser abstractions (e.g., instructions) that can still yield acceptable accuracy. Note that cycle-level simulation is not a guarantee of accuracy; only a validated model is, regardless of its level of abstraction [83, 239].

The adoption of the multicore-on-multicore model was led by SlackSim [49] with the introduction of both pessimistic and optimistic synchronization across simulated cores. HORNET [203] implements pessimistic synchronization, which in architectural simulators does not offer much room for scalability. Optimistic synchronization can potentially exploit more parallelism via speculation, yet imposes great complexity due to the necessary ability to roll back changes [101]. These difficulties motivated the emergence of approximate synchronization schemes that forgo event causality to achieve higher scalability. Graphite [181] pioneered this model. It uses Pin [160] as the emulator to implement fast, scalable simulation of multicores that can also be distributed across machines. It does not, however, model cache coherence. Later simulators such as Sniper [47], Zsim[209], ESESC [14] implement cache coherence and

Name	Pub. date	Emulator	Speed	Scalability	ISAs
SimOS [205]	1995	Embra [241] (DBT)	Med	None	MIPS-on-MIPS
Simics [165, 166]	1995	Interpreter	Low	None	several
gem5 [28]	2011	Interpreter	Low	None	several
PTLSim [246]	2007	QEMU (DBT)	Med	None	x86 guests
MARSS [192]	2011	QEMU (DBT)	Med	Low	x86 guests

Table 2.2: Comparison of full-system architectural simulators: emulation engine, speed, scalability on multicore hosts, and supported ISAs.

exploit approximate synchronization model to achieve some scalability on multicore machines. PriME [100] follows a similar approach, and it also scales across simulation hosts.

2.5.2 Full-System Simulation

Table 2.2 shows a comparison of full-system simulators in the literature, sorted chronologically. SimOS [205] pioneered key ideas in full-system emulation, such as the use of DBT for execution and the adoption of a softMMU to portably emulate virtual memory. Simics [166], which was developed at the same time as SimOS, has a lesser focus on speed, perhaps due to its stronger focus on accuracy. Today, arguably the most popular full-system simulator is gem5 [28], an open-source project that is a descendant of M5 [29] and GEMS [172]. gem5 performs event-driven simulation, which makes it a slow yet useful tool for simulating microarchitectural detail. gem5 models several ISAs, and has been validated for a subset of them against real hardware [105]. PTLSim [246] is an x86 simulator that uses QEMU to drive execution, which gives it higher speed than gem5 and access to the many devices emulated in QEMU. MARSS [192] builds on PTLSim to, among other improvements, leverage multicore hosts to scale the timing model, which is cycle-level.

By interacting with external simulation engines, full-system architectural simulators can model heterogeneous systems that integrate accelerators. Simics supports the attachment of externally-simulated SystemC devices [133]. gem5 derivatives have added support for accelerator models in PARADE [61] and gem5-aladdin [218]. Rabbits [103] integrates QEMU with SystemC accelerators by running QEMU inside a SystemC simulation thread.

None of the full-system simulators in the literature can simulate heterogeneous systems (i.e., multicores with accelerators) at high speed while leveraging multicore hosts. This goal has not been achieved due to the lack of fast, scalable full-system emulators, while user-mode simulators show that timing models can scale. Our work with Pico (Chapter 3) and Qelt (Chapter 4) fills this gap by turning QEMU into a fast, scalable machine emulator.

Chapter 3

Pico: Cross-ISA Machine Emulation

3.1 Introduction

Efficient, scalable cross-ISA DBT poses two main challenges. First, concurrent access to key data structures should avoid contention on the memory hierarchy. Second, guest and host ISAs may differ in the implementation of atomic operations, as well as in the memory consistency model; such mismatches impose additional work on the DBT engine, beyond simply performing instruction-by-instruction translation.

This chapter[1] proposes a design for a scalable cross-ISA dynamic binary translator that is simple, memory-efficient and correct. Our design, which we call Pico, is implemented on QEMU [24] due to its wide use and support of many different guest and host ISA combinations. The two main contributions of this chapter are:

- A memory-efficient design of a shared code cache for DBT engines. Based on the observation that *code translation* is rare, and that runtime is mostly spent on code execution, we keep the core logic of Pico simple, and achieve scalability through careful tuning of the emulator's data structures. In particular, the code cache is backed by a novel, highly concurrent hash table design that enables fast and scalable lookups (Section 3.2).

[1]This chapter incorporates and extends work previously published in the proceedings of the 2017 CGO conference [66]. Our implementation was successfully evaluated by the conference's Artifact Evaluation (AE) committee.

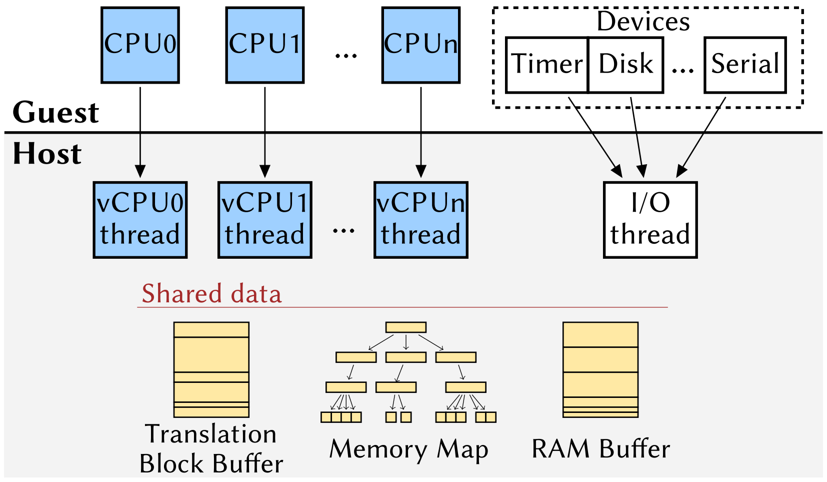


Figure 3.1: Pico's full-system architecture. Each guest CPU is emulated by a corresponding host "vCPU" thread. Guest devices are emulated by a single "I/O" thread.

- A scalable, fully correct cross-ISA approach to emulating atomic instructions and reconciling guest-host differences in memory consistency models. We emulate strongly-ordered architectures on top of weaker ones by leveraging the work of Lustig et al. [161]. When possible, atomics are translated to the equivalent operation on the host. Otherwise, they are emulated faithfully either by instrumenting stores or, as a high-performance alternative, by leveraging hardware transactional memory (HTM) on hosts that support it (Section 3.3).

We evaluate Pico on a 64-core x86_64 host running x86_64 code, and on a 12-core, 96-thread POWER8 host running x86_64 and Aarch64 code. Our results show that scalability of DBT with a shared code cache is comparable with native execution and hardware-assisted virtualization. Further, we quantify the implementation overhead of the different options to handle architectural mismatches between guest and host, exploring correctness vs. performance trade-offs for atomic instruction emulation.

3.2 Emulator Design

Pico's architecture, shown in Figure 3.1, has four main components: the state of CPUs, the memory map, the translation block cache and the guest RAM. In this section we cover the first three components, deferring the discussion of cross-ISA memory access emulation to Section 3.3.

We do not describe in detail the host "I/O" thread for device emulation, since it has been present in QEMU for several years, supporting concurrent emulation of devices and guest code execution with adequate performance.

3.2.1 CPU Execution

Pico allocates one host thread per emulated CPU; threads can then access the CPU registers without need for synchronization. However, CPUs do communicate with each other, even if sparingly; for example, the ARM architecture has instructions to flush the TLB on *all* cores in a *shareability domain*.

For this purpose Pico uses a two-pronged approach that scales for the common case, in which no communication is ongoing. First, every CPU loop is wrapped in an RCU read-critical section. Second, messages are delivered by setting a "flag" on the *consumer* CPU's state. The use of RCU allows *producer* CPUs to establish when messages have been consumed by waiting for a grace period to elapse. To ensure that the receipt of messages is bounded in time, CPUs check the request flag at the beginning of every translated basic block. Despite the simplicity, the cost of the checks is low: each basic block only needs two or three more instructions, depending on the host architecture—e.g., a load, a compare and a well-predicted branch.[2] Note that QEMU does not attempt to compile multiple basic blocks into a single compiled trace, otherwise the amortized cost could be made even lower.

3.2.2 Memory Map

QEMU-system organizes the guest's memory map as a tree of RAM, I/O and "container" memory regions, from which a radix tree is built for efficient lookups. Changes in the memory map are rare once the kernel has booted, and typically happen only

[2]QEMU originally relied on asynchronous signals to exit emulation, instead of checking a flag before executing every translation block. [24] This was changed in QEMU 1.5 (2013) to improve portability and improve thread-safety.

in response to device hot-plug and hot-unplug. Given how infrequent these changes are, the radix tree is simply rebuilt from scratch whenever a change occurs.

The memory map radix tree is read on every TLB miss[3] and on every interaction with an emulated memory-mapped device; it is therefore crucial for Pico to provide cheap access to it. The low update frequency and the "rebuild from scratch" approach make the memory map data structure an excellent candidate for RCU. Thus, once the tree is rebuilt, all CPUs are "kicked" out of their execution loop, thereby concluding their RCU read-side critical sections. This guarantees that they will all read the updated state upon resuming execution.

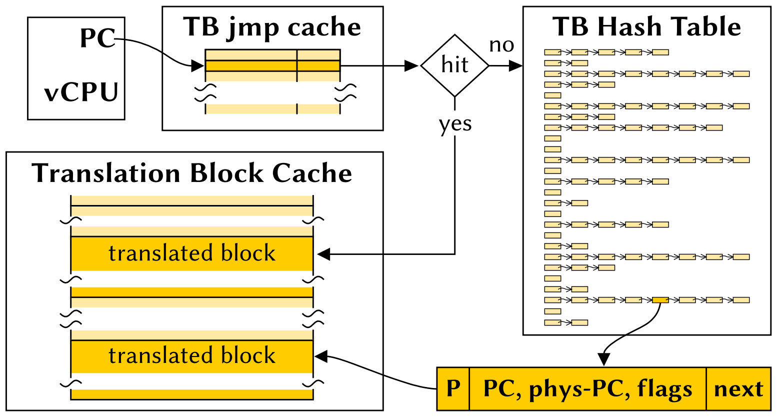
3.2.3 Translation Block Cache

The translation block cache minimizes code retranslation by buffering already translated code. It consists of translation blocks (TBs), i.e., guest basic blocks translated to the host architecture, that are indexed by an associated hash table.

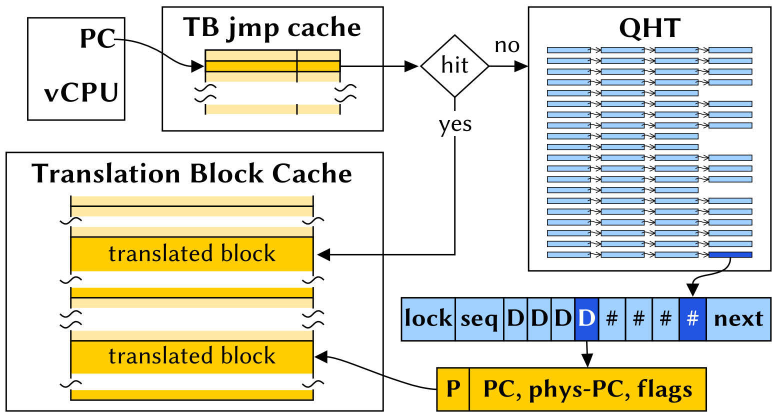
Even though TBs can be invalidated, memory in the block cache is never reused. When the block cache is full, the emulator stops all CPUs and starts over with an empty buffer. Such flushing of the block cache is done for simplicity and performance. For most workloads and with an appropriately sized code buffer, the hit rate is very high and the buffer is flushed rarely—on the order of 2-3 flushes/minute in system emulation and practically never for user mode. Thus, maintaining an eviction policy (such as "least recently used") would likely result in a net loss of performance: the associated bookkeeping would negate the gains of avoiding already rare flushes.

The translation block cache and associated lookup mechanism in QEMU is shown in Figure 3.2a. The hash table points to TB descriptors, and is indexed by guest physical address. The hash table uses separate chaining with a fixed number of buckets. From the CPU execution loop, TB lookups are performed in two steps. First, threads access a direct-mapped, CPU-private cache. On a hit, a pointer to the corresponding TB is returned. On a miss, the shared hash table is accessed after acquiring a global

[3] The memory map is not used in QEMU-user, since guest addresses are trivially translated to host addresses by adding a constant value.



(a)



(b)

Figure 3.2: Translation Block lookup mechanisms in (a) QEMU and (b) Pico. Pico's improved hashing results in a more uniform bucket distribution. Further, QHT has higher performance due to its dynamic resizing and concurrent lookup support.

lock. The CPU-private caches are tuned for latency; they are small and are invalidated relatively often (e.g., after every TLB flush). Thus, contention on this lock can be high.

Hash table design. Pico increases performance and scalability of the shared hash table in two ways. First, we improve the hashing function used to place TB descriptors in the hash table buckets. Second, we adopt a new hash table design that enables correct, concurrent lookups.

Pico uses *xxhash* [6], a high-performance non-cryptographic hash algorithm, to mix all three parts of the key: the virtual program counter (*PC*), the physical program counter (*phys-PC*) and a set of flags representing the active CPU mode (*flags*). Using all this information leads to a significantly more uniform distribution of TBs into buckets: for instance, the longest observed chain after fully booting Debian "jessie" on ARM is brought down from 550 to 40 TBs.

Our hash table design, called QHT, is highlighted in Figure 3.2b. Its main feature is support for concurrent reads with optimal scalability. In addition, even though Pico does not need it for the translation block cache, QHT also allows concurrent writes to separate buckets. In QHT, bucket chains are composed of cache line–sized nodes. Each node has a head spinlock for serializing writers, and stores multiple pointer-sized objects ("D" in the figure) along with their precomputed hashes (shown as "#").

QHT is similar to CLHT-LB [77]; however, CLHT imposes a restriction on the memory allocator that can be used with the hash table: the same address cannot be returned twice by the allocator while reads are occurring. To remove this restriction, QHT uses a per-bucket sequence number and the *seqlock* algorithm [149] to synchronize readers against writers. Readers of a *seqlock* need not perform any write, avoiding cache line bouncing and preserving scalability; they only need to retrieve the sequence number with a regular load (plus, on non-TSO hosts, a read fence) before and after traversing each bucket.

Writers update the sequence number before and after a concurrent write; therefore, if the low bit is set, readers know a writer is currently active. Readers wait until the low bit is clear, then access the bucket. If the sequence number changes during the access, the reader might have seen an inconsistent state and retries the access. Retries are highly unlikely due to (1) the size of the hash table (it is not uncommon to have several hundred thousand elements), and (2) its low update rate—about 6% when booting Debian "jessie" on ARM, arguably a worst-case workload since most translated code is executed only once.

Figure 3.3 presents, for several hash table configurations, the time it takes to fully boot Debian "jessie" on ARM and immediately shut down. QEMU uses a fixed-size

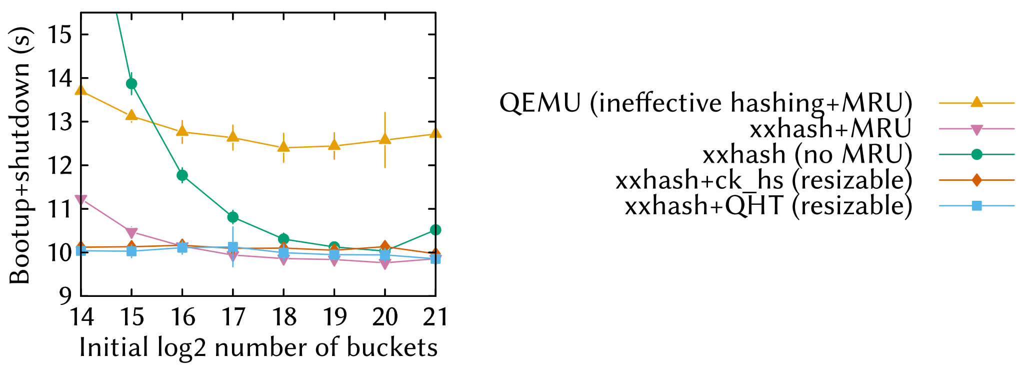


Figure 3.3: Bootup+shutdown time of Debian "jessie" in an ARM guest running on an Intel Haswell i7-4790K host.

hash table together with a most recently used (MRU) promotion policy, which moves items to the front of the bucket after every successful lookup. The plot shows that using MRU along with effective hashing (xxhash) *and* appropriate sizing gives optimal performance. On the other hand, MRU writes to memory on every lookup, which hurts scalability due to excessive cache line bouncing.

Furthermore, a fixed-size hash without MRU has inferior performance due to excessive bucket chain lengths and increased number of cache misses. QHT virtually matches the performance of an ideally-sized hash table with MRU promotion by supporting resizes concurrent with lookups. Since resizes are rare, concurrent writes spin on bucket locks while a resize takes place. The freeing of pre-resize bucket chains is deferred by using RCU.

We benchmarked QHT against other hash table designs featuring resizing and concurrent lookup. While we did not apply CLHT in QEMU due to the aforementioned memory allocation requirements, Figure 3.3 shows that when booting a full system QHT has performance on a par with that of *ck_hs*, the hash set implementation from concurrencykit [5].

However, QHT and *ck_hs* show great performance differences in write-heavy scenarios. Figure 3.4 plots the scalability to 64 cores of QHT, CLHT and *ck_hs*, driven from a hash table microbenchmark operating on 200K elements at different update rates. Due to its use of seqlocks, QHT achieves performance comparable to CLHT's

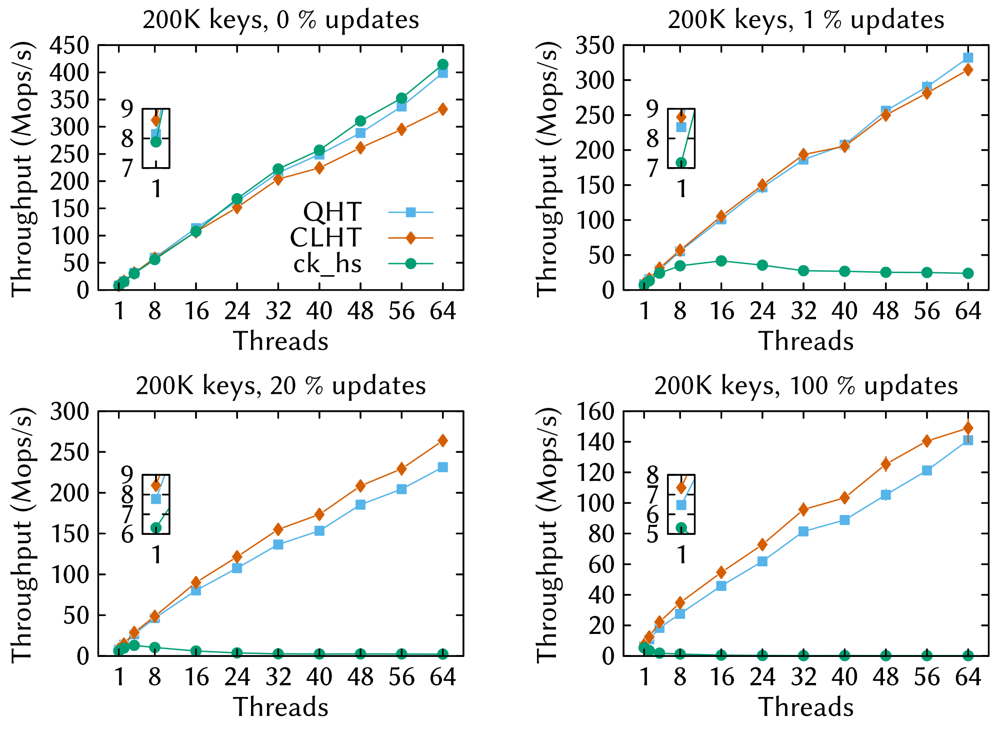


Figure 3.4: QHT, CLHT and *ck_hs* performance comparison.

while not imposing restrictions on the memory allocator. On the other hand, *ck_hs* is an open-addressed hash set and therefore takes the same lock around every insert; as a result, it scales poorly even for modest update rates, which limits its general applicability. This limitation is shared with similar hash table implementations, such as the one proposed by Bruening et al. in [40].

3.3 Correct, Cross-ISA Memory Accesses

Two cross-ISA issues are specific to multi-threaded emulators. First, the emulator has to handle mismatches in the memory consistency models of the source and target ISAs. Second, it has to correctly emulate the semantics of atomic operations. The latter problem, in turn, has to be attacked differently for the two families of atomic operations: *compare-and-swap* (*CAS*) and other read-modify-write operations on one hand, and *load-locked/store-conditional* (*LL/SC*) on the other.

3.3.1 Mismatches in the Memory Consistency Model

The memory consistency model of an ISA is made of a set of implicit ordering guarantees that the ISA promises to respect. These could be, for example, which memory accesses (loads, stores, atomic read-modify-write sequences) can be reordered in front of older accesses, or whether memory ordering obeys causality (also known as *transitive visibility*).

Performing DBT on a host that only allows reordering loads before older stores (such as x86_64) is trivial, because the emulated code cannot have an implicit ordering guarantee that the host does not respect. Unfortunately, the opposite case is problematic: performing DBT of guest code on a host whose ISA is *weaker* than that of the guest means that certain reorderings need to be explicitly forbidden by the translator.

Recent work by Lustig et al. [161] deals with exactly this problem: given two memory consistency models, they provide a framework, called ArMOR, that generates a state machine for the translator that guarantees correct execution of code from stronger ISAs on weaker ones.

We evaluate the performance impact of ArMOR's state machines in Section 3.4.5.

3.3.2 Compare-and-Swap (CAS)

Single-word *CAS* in the guest can be easily mapped to *CAS* in the host. Pico implements this similarly to how COREMU [237] does[4].

Multi-word *CAS* is required to emulate processors with 64-bit words on top of 32-bit ones. Fortunately, most 32-bit processors do provide 64-bit *CAS* or *load-locked/store-conditional* (*LL/SC*) operations; the only exception is PowerPC processors (until POWER8, which can implement it using hardware transactional memory).

This approach cannot portably implement the x86 `cmpxchg16b` instruction, which performs a *CAS* operation on a 128-bit quantity. This operation is not available natively on 32-bit hosts, as well as on 64-bit PowerPC processors until POWER8. Fortu-

[4]Source code available at `http://coremu.sf.net/`

nately, this instruction is rarely used by operating systems (e.g., Linux), and the x86 `cpuid` instruction marks its presence with a separate feature bit. We therefore either hide this feature bit from the guest, or fall back to the strategies used for emulating load-locked/store-conditional instructions, which are described below.

Bus-locked atomics. Architectures such as x86 support the atomic execution of certain instructions via a prefix (e.g., *LOCK xadd*). The same infrastructure used for *CAS* can also be used to implement other instructions such as atomic fetch-and-add or test-and-set; they can be trivially reduced to a *CAS* loop, as done in COREMU [237]. However, for increased efficiency, Pico leverages the equivalent bus-locked instruction on the host whenever possible.

3.3.3 Load-Locked/Store-Conditional (*LL/SC*)

Rather than providing *CAS*, most RISC processors implement atomic read-modify-write operations through two instructions, *load-locked* (also known as *load-link*) and *store-conditional*. The first returns the current value of a memory location; the second stores a new value only if no updates have occurred to the location since the load. Unlike *CAS*, these instructions detect the case where a location has been changed to a different value and then back to the original.

Implementing *LL/SC* primitives is trivial in a sequential emulator. Whenever a *store-conditional* instruction is concurrent with one or more regular stores, however, a parallel emulator has to order the conditional store against each regular store. This is a *consensus problem* of order two, whose solution requires an atomic instruction (such as a test-and-set instruction) in both regular and conditional stores [114]. A trivial, non-scalable solution is to stop all other CPUs while executing *store-conditional* instructions. This is exactly what QEMU does; performance however drops very quickly even with very few concurrent threads. A solution, in order to scale, should thus avoid atomic instructions whenever the store does not conflict with *LL/SC* operations. We now present three different solutions, exploring the trade-offs between correctness, scalability and portability.

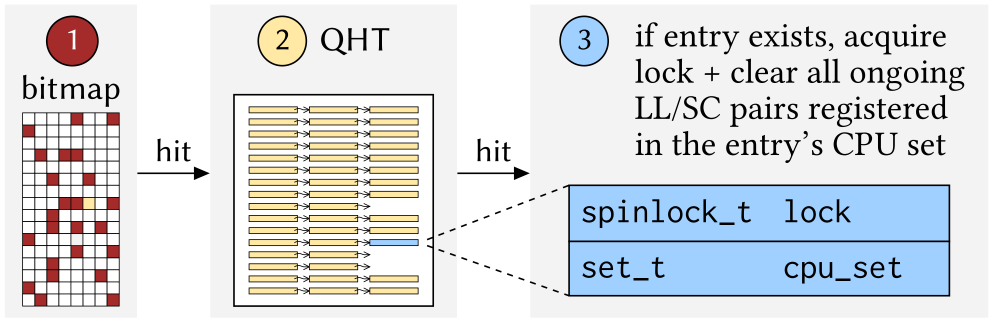

1
bitmap
hit
2 QHT
hit
3 if entry exists, acquire lock + clear all ongoing LL/SC pairs registered in the entry's CPU set
spinlock_t lock
set_t cpu_set

Figure 3.5: Instrumentation of stores in *Pico-ST*. Stores execute while holding the appropriate lock iff an atomic instruction has previously been performed on their target cache line.

***Pico-CAS:* a (slightly) incorrect and scalable solution.** The simplest, but nonetheless practical solution is to not implement exact *LL/SC* semantics; instead, a *store-conditional* operation can simply perform a *CAS* from the value fetched by *load-locked* to the argument of *store-conditional*.

Of course, this suffers from the *ABA problem* [177]: if the location were modified twice between *load-locked* and *store-conditional*, and the second write restored the original value, then the *store-conditional* would incorrectly succeed.

At the same time, in practice this is rarely problematic. Portable code using C/C++11 atomics only has access to *CAS*, and the entire Linux kernel does not employ *LL/SC* in ways that would break when emulated with *CAS*. Therefore, synchronization algorithms and lock-free data structures avoid the ABA problem using techniques such as reference counting, RCU or hazard pointers [178]. Nevertheless, it is worth looking further for fully correct solutions.

***Pico-ST:* a correct, scalable and portable solution.** It is possible to avoid the ABA problem if one accepts a slowdown in single-threaded performance. This requires instrumenting stores to check whether the physical address to store to has *ever* been accessed atomically. If so, ongoing *LL/SC* pairs to the same cache line are canceled.

For each cache line that an *LL/SC* operates on, a corresponding entry is kept. An entry has two fields: a lock and a set of CPUs to track ongoing *LL/SC* operations.

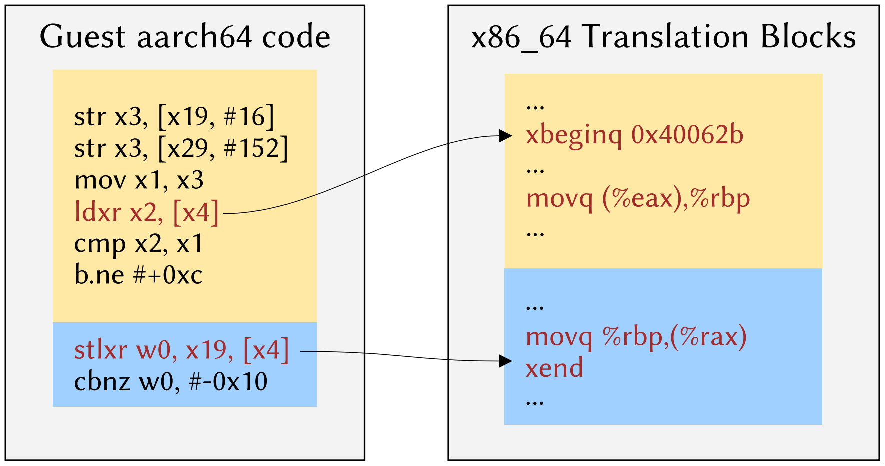
Entries are looked up by coupling a scalable hash table (we use QHT, see Section 3.2.3) and a bitmap. While each entry in the hash table represents a single cache line, each bit in the bitmap can represent a configurable number of cache lines, thus keeping its storage overhead practical. A bit set in the bitmap means that it *is possible* but not guaranteed that a lock might have to be taken for a given address; to dispel the uncertainty, the hash table is checked with the full address of the cache line. The instrumentation preceding emulated stores is depicted in Figure 3.5.

An additional measure is necessary to avoid races between the first *load-locked* operation on a cache line and a concurrent store to it by another CPU. This has to be done in three steps: (1) insert the appropriate entry in the hash table and bitmap, (2) kick all other CPUs out of the execution loop, and wait for them to actually exit; and (3) proceed with the *load-locked* emulation. Step 2 can be implemented easily by waiting for an RCU grace period (see Section 3.2.1). After Step 2, all regular stores to the newly-added cache line will be protected by the spinlock; this makes Step 3 safe.

Finally, we relax the requirement of keeping an entry for each cache line that an *LL/SC ever* operated on, which could potentially degenerate into acquiring a lock on every emulated store. Thus, we periodically reset both bitmap and hash table, thereby guaranteeing at negligible cost that the bitmap will remain sparsely populated.

***Pico-HTM:* leveraging hardware transactional memory.** If the host also supports *LL/SC*, one may think of using them for emulating the target's *LL/SC*. This is dangerous, however, because most processors constrain the instructions that can appear between an *LL/SC* pair. If these restrictions are not respected, the store might fail spuriously. The extra overhead of dynamic translation, such as TLB lookups and register spills, may thus cause the store to fail forever.

Fortunately, some of the newest members of the POWER, s390 and x86_64 families feature HTM, which does superficially resemble *LL/SC*. HTM is however more flexible than *LL/SC* and places fewer constraints on the instructions that can appear between the *LL/SC* pair. POWER processors, for example, can write to several hundred cache lines in a single transaction [153].



```
str x3, [x19, #16]
str x3, [x29, #152]
mov x1, x3
ldxr x2, [x4]
cmp x2, x1
b.ne #+0xc
stlxr w0, x19, [x4]
cbnz w0, #-0x10
```

```
...
xbeginq 0x40062b
...
movq (%eax),%rbp
...
...
movq %rbp,(%rax)
xend
...
```

Figure 3.6: Example *LL/SC* pair translated with *Pico-HTM*.

As depicted in Figure 3.6, Pico compiles *load-locked* to a "begin transaction" instruction followed by a regular load, and a *store-conditional* to a regular store followed by a "commit transaction" instruction. This works because all commercial HTM implementations provide *strong atomicity* [31]. Under strong atomicity, a store conflicting with a transaction will force the transaction to abort; the emulator can then check for conflicting regular stores just by testing whether the transaction completed successfully.

There is one important difference between HTM and *LL/SC*. Regular stores between the *load-locked* and *store-conditional* instructions persist after a failed conditional store; with HTM, instead, an abort rolls back all stores in the transaction. We cannot therefore map a transaction abort directly to a *store-conditional* failure. Instead, we retry the transaction until it succeeds, so that the conditional store actually never reports a failure. Because the semantics of *store-conditional* is respected, the difference is not visible to the emulated program.

HTM also requires a fallback for repeated aborts. After a few failed attempts, or if one of the blocks between *load-locked* and *store-conditional* is not translated, Pico executes the *LL/SC* sequence with all other CPUs stopped.

3.4 Evaluation

3.4.1 Setup

Machines. We perform measurements on the following machines, which all have 64G of RAM and run Linux v4.4:

SKL has a 3.4GHz 4-core Intel Skylake i7-6700 processor with 2-way simultaneous multithreading (SMT), for a total of 8 hardware threads.

P8 has two 3.3 GHz 6-core IBM POWER8 processors with 8-way SMT, for a total of 96 hardware threads.

AMD has four 2.3GHz, 16-core AMD Opteron 6376 processors, for a total of 64 cores.

The guest kernel is always Linux v4.4. We use QEMU v2.4.0-rc3 as the baseline emulator, since newer versions already include some of the improvements in Pico.

Workloads. We use SPEC CPU2006 benchmarks for single-threaded performance measurements. For measuring scalability we use the PARSEC suite [26], as well two server workloads: the *pgbench* tool in PostgreSQL v9.5 [3] and the *Masstree* in-memory key-value store [169]. We compare Pico-user to native execution, and Pico-system to KVM.

For evaluating the overhead of atomic instruction emulation we wrote a simple microbenchmark called *atomic_add*. Each thread in *atomic_add* executes a loop that atomically increments a random element of an array of integers, which is appropriately padded to avoid false cache line sharing. By varying the size of the array, we can observe different levels of contention in the memory hierarchy.

All experiments are run five times; we show the resulting mean and corrected sample standard deviation. For each experiment we choose the thread pinning policy that exhibits higher performance from either scattering threads evenly across NUMA nodes, or favoring same-node pinnings.

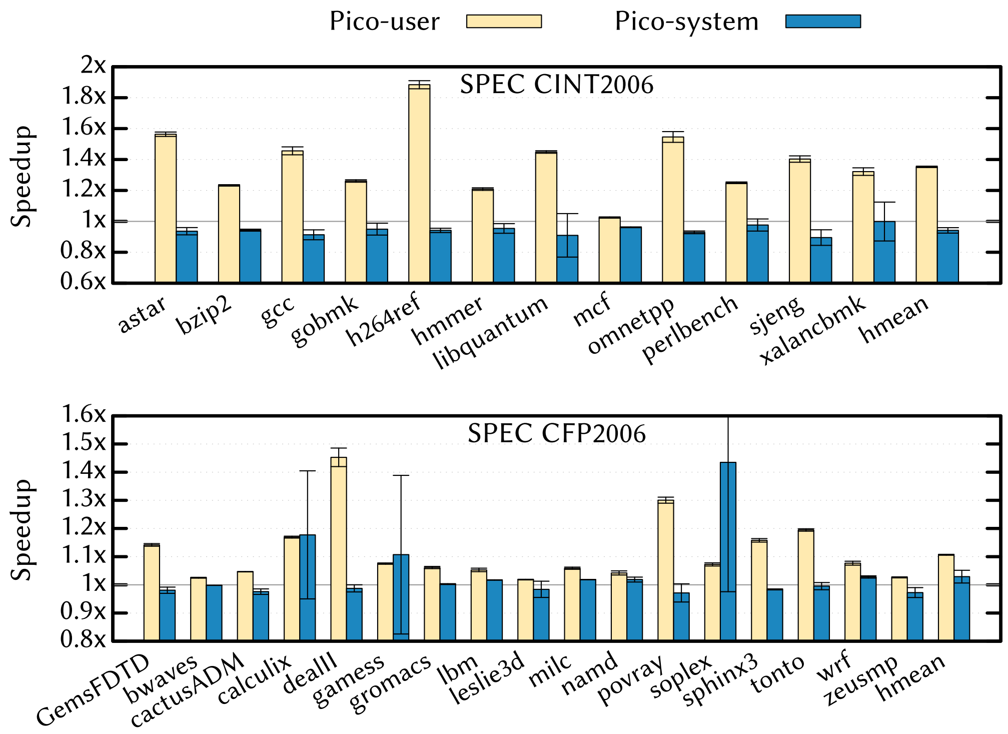


Figure 3.7: Speedup on *SKL* of Pico over QEMU for single-threaded x86_64 SPEC06 workloads.

3.4.2 Single-Threaded Performance

This experiment (Figure 3.7) compares the performance of Pico for single-threaded execution over the baseline QEMU implementation on *SKL*. QEMU-user already supports multiple threads of execution, but this introduces overhead even for single-threaded programs; for example, translation block lookups (Section 3.2.3) are serialized with a lock. Pico-user is thus 20–90% faster than QEMU-user, mostly due to its better translation block hashing and to QHT's cache efficiency.

Pico-system introduces locks and fences to cover previously unprotected data structures, and converts some memory accesses to atomic operations. This balances the advantages of improved hashing, making performance virtually identical to QEMU-system—on average slightly worse for integer benchmarks, and slightly better for floating-point.

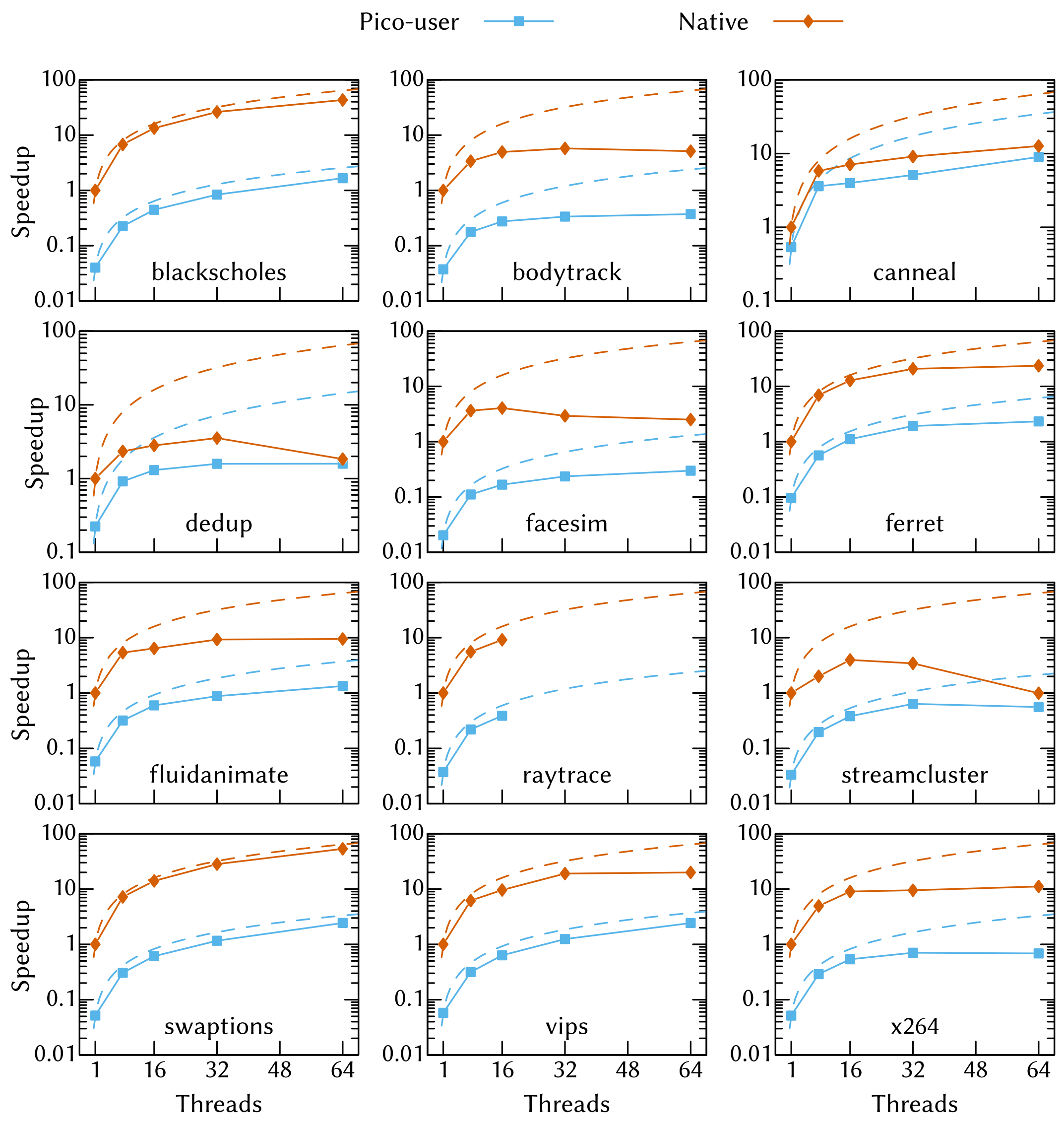


Figure 3.8: Speedup over native on *AMD* for PARSEC under Pico-user.

3.4.3 Parallel Workloads

In this experiment we evaluate the scalability of Pico-user by running PARSEC with the `native` input set[5]. Figure 3.8 compares the scalability of Pico against that of a native run on *AMD*. Speedups are shown normalized over the native single-threaded

[5]We had issues running two PARSEC benchmarks on *AMD*: *freqmine* crashed frequently, and so did *raytrace* beyond 16 threads. The crashes happened for both native execution and Pico.

execution times; ideal, linear scaling is shown with a dashed line for comparison.

Most PARSEC benchmarks do not scale well to dozens of cores [221], showing a performance cliff that is indicative of excessive cache line contention. Nevertheless, on average, the scalability of Pico is better than that of native execution. This is because the slowdown caused by DBT reduces the rate of accesses to shared memory in the workload, thereby delaying the onset of the contention-related performance cliff (e.g., *streamcluster*) and even avoiding it altogether (e.g., *facesim*, *fluidanimate*). This effect is present for most workloads we tested, but it is particularly visible here due to the slow floating point emulation in the DBT engine.

Benchmarks that do not have particular contention show similar scalability under both Native and Pico, as it can be seen, for instance, in *swaptions*, *blackscholes* or *canneal*.

3.4.4 Server Workloads

We now show the results for full-system emulation of a guest with 32G of RAM running multi-threaded (Masstree) and multi-process (PostgreSQL) server workloads on *AMD*. These workloads are representative of large virtual machines and stress two potential sources of performance degradation: code translation and device emulation.

Code translation is stressed by multi-process server workloads due to their code size [40], and because the virtual address of the program counter is part of the translation block hash key; even if the text of the program is loaded only once in memory by the operating system, techniques such as address space layout randomization (ASLR) can cause the emulator to translate it multiple times. Device emulation is stressed simply because QEMU runs all emulation under a single global mutex, and we did not change this in Pico.

For PostgreSQL, we use *pgbench* to create and populate a database of scale factor 150, running each test for 120s and spawning two connections per thread. Both PostgreSQL server and *pgbench* run in the same guest, with the server configured with a buffer of 8GB. For Masstree, we run 10s get/put tests on a database initialized with 140M “1-to-10-byte decimal” [169] keys.

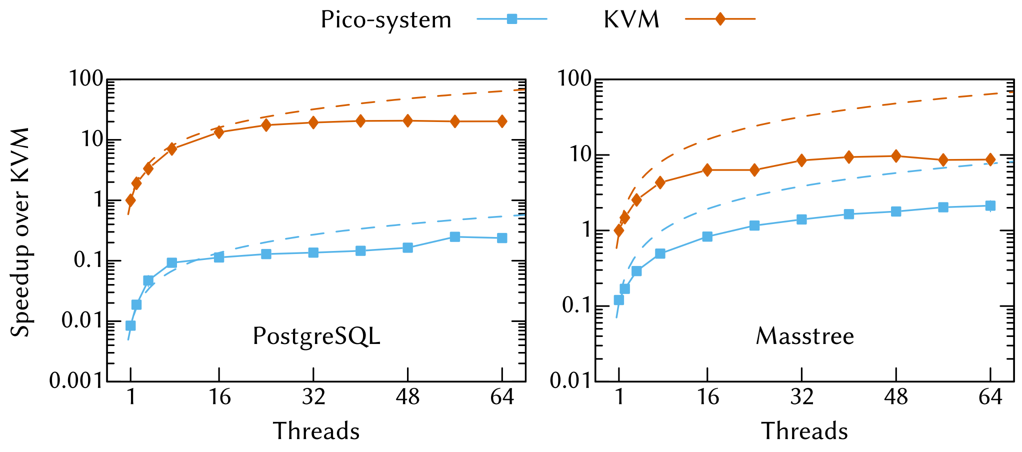


Figure 3.9: Speedup vs. KVM on *AMD* for server workloads under Pico-system.

Figure 3.9 shows the results of the experiment. Both of the benchmarks scale up to 64 threads. In the case of PostgreSQL, which performs disk and socket I/O as well, device emulation is a bottleneck at 16 to 40 threads. In this range, KVM can take advantage of optimizations to device emulation, such as moving the CPU's interrupt controller (local APIC) inside the hypervisor and triggering the QEMU I/O thread directly from the hypervisor. Pico on the other hand cannot keep all cores pegged at 100% CPU utilization. Nevertheless, the scalability of Pico is in line with KVM's.

Masstree, which is a memory-only workload, scales under Pico up to at least 64 threads. As in the PARSEC benchmarks, emulation scales better than KVM because of the reduced rate of shared memory accesses. In neither case code translation turns out to be a bottleneck.

3.4.5 Mismatches in the Memory Consistency Model

In order to quantify the cost of emulating parallel code from strongly-ordered ISAs on weakly-ordered hosts, we implemented the two state machines provided by ArMOR for executing x86_64 guest code on a PowerPC host. We then ran x86_64 SPEC06 code on *P8*. The two state machines, which we call SYNC and PowerA, are summarized as follows.

- *SYNC:* Insert a full memory barrier (`sync` in PowerPC assembly language) before every load or store. This is always correct for all possible legal code.

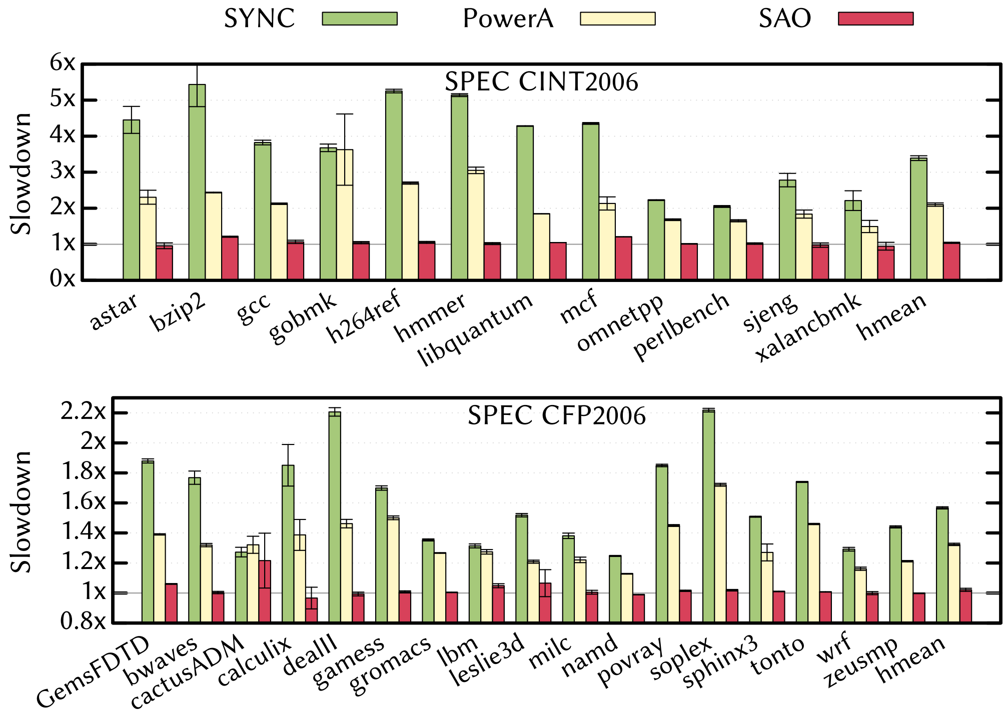


Figure 3.10: Slowdown on *P8* for Pico-user running x86_64 SPEC06 benchmarks, with two ArMOR state machines and hardware strong-access ordering, relative to omitting all barriers in the translated code.

- *PowerA:* Separate loads with `lwsync` barriers, pretending that PowerPC is multi-copy atomic even though it is not[6]. While this does allow illegal behavior for the `iriw` litmus test [32], it allows for greater performance if the user is sure that the system does not include code similar to `iriw`—which indeed is practically never seen.

The results are shown in Figure 3.10. SYNC provides full correctness but incurs significant slowdown, on average surpassing 3X for integer workloads. PowerA has a more adequate average slowdown of approximately 2X for CINT2006, at the expense of not handling correctly the `iriw` pattern. The slowdown of both approaches is significantly lower for CFP2006, since floating point emulation dominates execution time.

[6]See the ArMOR paper for details on this state machine.

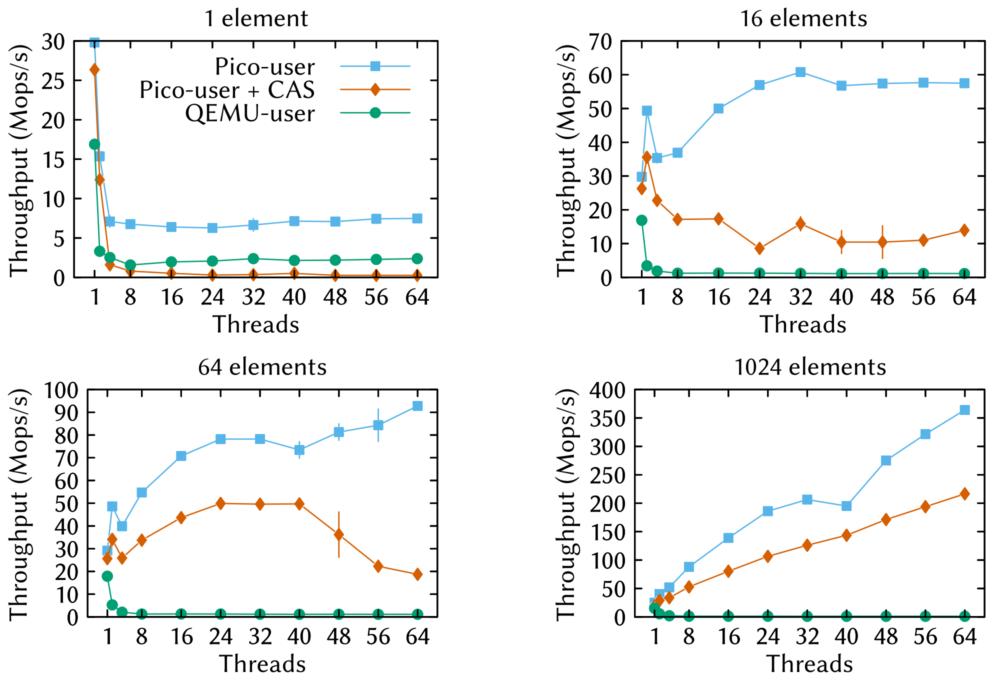


Figure 3.11: Performance on *AMD* for x86_64 *atomic_add*.

P8 has hardware support for strong-access ordering (*SAO*) [8], which in our tests shows negligible overhead. Unfortunately, SAO is only available on recent IBM hardware, which makes its use non-portable. Nevertheless, ArMOR is a viable strategy to correctly emulate parallel code from strongly-ordered ISAs on weakly-ordered hosts.

3.4.6 Bus-Locked Atomics

Figure 3.11 compares the performance of emulating bus-locked atomics with the equivalent bus-locked atomics on the host (Pico) and emulating them with a *CAS* loop (Pico + CAS). In hardware, bus-locked atomics can be implemented more efficiently than *CAS*. This explains why Pico's performance is superior with just one array element, which is a worst-case scenario from a scalability viewpoint. Furthermore, when contention is slightly lower (64 elements), Pico shows greater scalability, scaling to 64 cores whereas the *CAS* loop's performance collapses around 40 cores. QEMU shows no scalability in either scenario, due to its serial emulation of atomics.

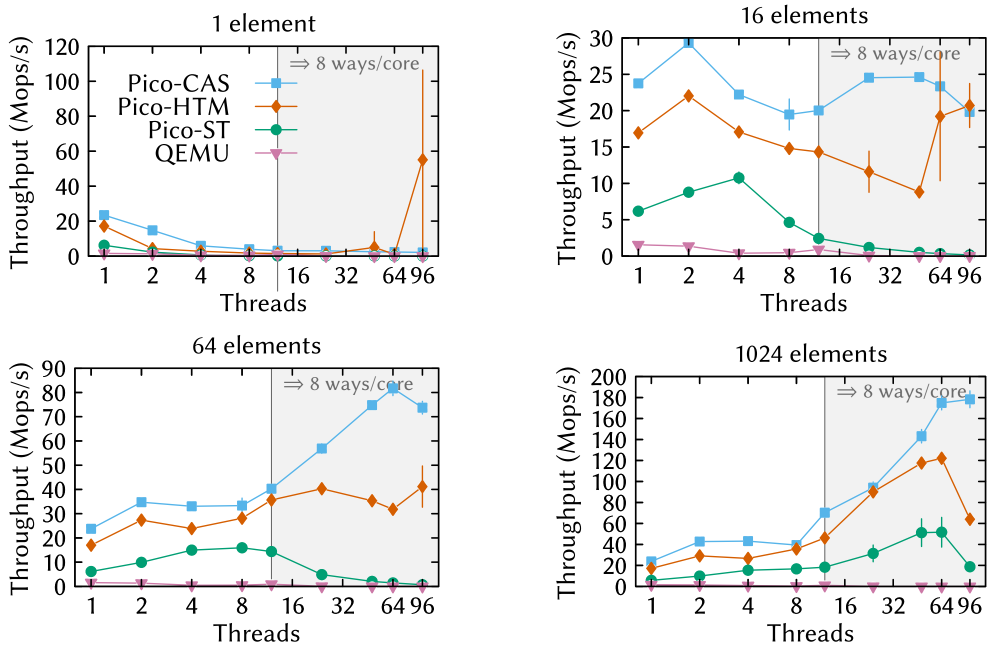


Figure 3.12: Performance on *P8* for Aarch64 *atomic_add*.

3.4.7 Load-Locked/Store-Conditional (*LL/SC*)

Figure 3.12 compares *atomic_add* performance on *P8* for QEMU-user and the three Pico approaches presented in Section 3.3.3. We ran these experiments on *P8* because of its HTM support and large number of hardware threads.

The overall performance of the Pico approaches depends on the overhead of the emulation mechanism. Thus, Pico-CAS is the fastest, followed by Pico-HTM and Pico-ST. All three approaches show scalability that grows as contention (i.e., number of array elements) is reduced.

The results become increasingly noisy as the number of used SMT threads grows (i.e., beyond 12 threads). For 96 threads, Pico-HTM is occasionally able to beat Pico-CAS and scale almost linearly. Our hypothesis is that the POWER8 microcode optimizes the case where all hardware threads in the same core are contending for the same cache lines, and effectively ensures that the transactions do not conflict. In fact, by forcing the benchmark to run on fewer cores we were able to see the same behavior also for 32, 48 and 64 threads (corresponding to 4, 6 and 8 fully-utilized cores).

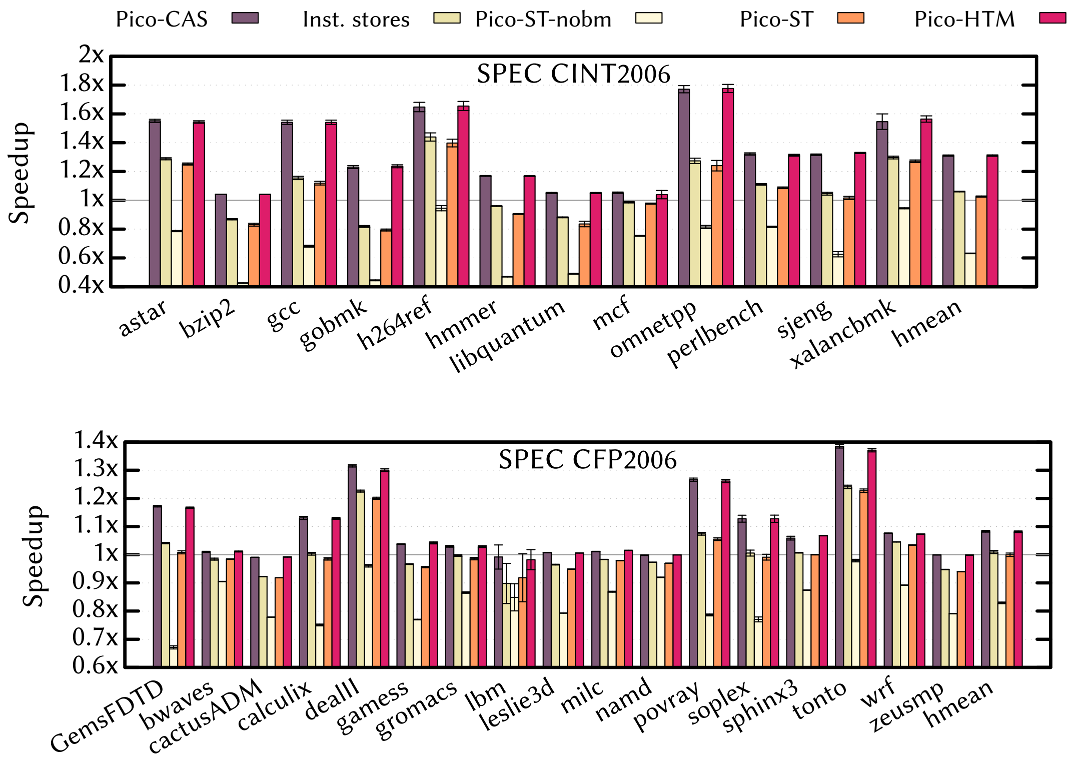


Figure 3.13: Speedup over QEMU on *SKL* of different implementations of *LL/SC* in Pico-user runs of Aarch64 SPEC06.

The plots show the trade-off between correctness, scalability and portability. If fully correct *LL/SC* emulation is not necessary, Pico-CAS has the highest performance while maintaining portability. Failing that, Pico-HTM provides good performance if processor support is available or, for hosts without HTM extensions, Pico-ST is both fully correct and portable.

Of the three, Pico-ST is also the only one to have non-trivial single-threaded overhead. To characterize this, we emulate SPEC06 benchmarks (compiled for Aarch64) on *SKL* in user-mode. Figure 3.13 shows the results.

Pico-HTM and Pico-CAS show the highest performance, which is not surprising given that they only affect the emulation of atomic operations; these are rare in SPEC06. The speedup is similar to that in Figure 3.7; as discussed in Section 3.4.2, it is due to improvements in Pico, and not to atomic instruction emulation.

In order to analyze the performance of Pico-ST, we include two additional sets of results in Figure 3.13. The *Inst. stores* set is obtained from instrumenting stores in Pico with empty helper functions, thereby measuring the cost of calling C code around every store; *Pico-ST-nobm* is Pico-ST without the bitmap acting as a filter for QHT lookups. Two observations can be made. First, the limitations in QEMU's register allocator introduce a high overhead in Pico's store instrumentation; the C helpers slow down emulation on average by around 20%. Second, the bitmap plays a key role in filtering accesses to the hash table; QHT, despite its high performance, requires a non-trivial amount of instructions that are easily outperformed by a bitmap lookup. Thanks to the bitmap, Pico-ST only has 4% overhead above that of store instrumentation. Since the bitmap check is so effective, a natural improvement to Pico-ST would be for the backends to inline it in the generated assembly code. This change would be more invasive than the other changes we made in Pico, for which portable C code was enough.

3.5 Additional Related Work

Concurrent hash tables. The idea of using cache-friendly buckets, as done by QHT and CLHT, was introduced earlier in MemC3 [95], albeit partial hashes were used to minimize cache references due to Cuckoo hashing. Leveraging RCU for concurrency in hash tables was proposed by Triplett et al. [232]. We use RCU for deferring QHT bucket's deletion, and handle dynamic resizes (which in our case are rare) by acquiring all bucket locks.

3.6 Summary

In this chapter we have presented Pico, a novel design for multi-threaded cross-ISA emulators. Pico leverages multi-core hosts by using a shared code cache from a highly parallel fast path. Furthermore, Pico adopts recent research for handling memory consistency model mismatches between guest and host, and proposes different strategies for the correct emulation of atomic instructions (including *LL/SC* on *CAS* hosts) and strongly-ordered memory accesses.

We evaluated our design in the QEMU open-source emulator. Our experimental evaluation covers both user-mode and full-system emulation, comparing Pico against QEMU and native execution. Our results show that Pico's design scales to 64 cores without forgoing simplicity or memory efficiency.

Pico's implementation was merged into upstream QEMU during the development of the v2.9 version, which was released in September 2017. The only two differences between QEMU and Pico's implementation are that (1) QEMU uses Pico-CAS for *LL/SC* emulation, and (2) support for full-system parallel emulation of x86_64 guests was merged later—it debuted in QEMU v3.1, which was released in December 2018.

Chapter 4

Qelt: Cross-ISA Machine Instrumentation

A note on the use of "QEMU". In this chapter, with "QEMU" we refer to a QEMU version that already incorporates the implementation described in Chapter 3, which has been part of QEMU since the v2.9 release. □

4.1 Introduction

The design presented in the previous chapter, while an upgrade over the original QEMU, does not address two common scenarios that are challenging for cross-ISA machine emulators. First, emulated floating point (FP) code incurs a large overhead (2× slowdowns are typical [24]), which hinders the use of these tools for guest applications that are FP-heavy, such as the emulation of graphical systems (e.g. Android) or as a front-end to computer architecture simulators that run scientific workloads. As discussed in Section 2.2.3.3, this FP overhead is rooted in the difficulty of correctly emulating the guest's FP environment, which can greatly differ from that of the host. This is commonly solved by forgoing the use of the host FPU, using instead a much slower *soft-float* implementation, i.e. one in which the host executes no FP instructions.

Second, efficient scaling of code translation when emulating multi-core systems is non-trivial. The scalability of code translation is not an obvious concern in DBT, since code execution is usually more common. However, some server workloads [40] as well as parallel compilation jobs (e.g., in cross-compilation testbeds of software projects that support several ISAs, such as the Chromium browser [1]) can show both high parallelism and large instruction footprints, which can limit the scalability of their emulation, particularly when using a shared code cache.

In this chapter[1] we first address these two challenges by improving cross-ISA DBT performance and scalability. We then combine these improvements with a novel ISA-agnostic instrumentation layer to produce a cross-ISA dynamic binary instrumentation (DBI) tool, whose performance is higher than that of existing cross-ISA DBI tools (e.g., [88, 113, 230, 247]).

Same-ISA DBI tools such as DynamoRIO [38] and Pin [160] provide highly customizable instrumentation in return for modest performance overheads, and as a result have had tremendous success in enabling work in fields as diverse as security (e.g., [58, 130]), workload characterization (e.g., [30, 216]), deterministic execution (e.g., [12, 85, 186]) and computer architecture simulation (e.g., [47, 124, 181]). Our goal is thus to narrow the performance gap between cross-ISA DBI tools and these state-of-the-art same-ISA tools, in order to elicit similarly diverse research, yet for a variety of guest ISAs.

In this chapter we make the following contributions:

- We describe a technique to leverage the host FPU when performing cross-ISA DBT to achieve high emulation performance. The key insight behind our approach is to limit the use of the host FPU to the large subset of guest FP operations that yield identical results on both guest and host FPUs, deferring to soft-float otherwise (Section 4.2.1).
- We present the design of a parallel cross-ISA DBT engine that, while remaining memory efficient via the use of a shared code cache, scales for multi-core guests that generate translated code in parallel (Section 4.2.2).

[1]This chapter incorporates and extends work scheduled for publication in the proceedings of the upcoming 2019 VEE conference [67].

- We present an ISA-agnostic instrumentation layer that converts a cross-ISA DBT engine into a low-overhead cross-ISA DBI tool, with support for state-of-the-art instrumentation features such as instrumentation injection at the granularity of individual instructions, as well as the ability to instrument guest operations that are emulated outside the DBT engine (Section 4.2.3).

We implement our approach in Qelt, a cross-ISA machine emulator and DBI tool based on QEMU [24]. Qelt achieves high performance by combining our novel techniques with further DBT optimizations, which are described in Section 4.2.4.2. As shown in Section 4.3, Qelt scales when emulating a guest machine used for parallel compilation, and it outperforms (1) QEMU as both a user-mode and full-system emulator and (2) state-of-the-art cross-ISA instrumentation tools. Last, for complex instrumentation such as cache simulation, Qelt can match the performance of state-of-the-art, same-ISA DBI tools such as Pin.

4.2 Qelt Techniques

We now present the techniques that allow Qelt to achieve high performance and portability. First, we accelerate the emulation of FP instructions by leveraging the host FPU for the vast majority of guest FP instructions. Next, we improve the scalability of multi-core emulation with a DBT engine that avoids global locks while keeping a shared code cache. Third, we describe an ISA-agnostic instrumentation layer that allows us to convert a DBT engine into a cross-ISA DBI tool. Finally, we cover some additional DBT optimizations that further increase Qelt's speed.

4.2.1 Fast FP Emulation using the Host FPU

Speeding up the emulation of guest FP instructions using the host's FPU is deceptively simple. A naïve implementation would first clear the host's FP flags, execute the equivalent FP instruction on the host, check the host FP flags and then raise the appropriate flags on the guest. This approach would be trivial to implement, and would be correct for many FP instructions. Performance, however, would be abysmal, as we show in Section 4.3.4. This is due to the lack of optimizations *in the FPU hardware* for the use case of clearing/checking the FP flags, which is justified by how rare these operations are in FP workloads.

Our approach is thus to leverage the host FPU but only for a subset of all possible FP operations. Fortunately, as we discuss next, this subset covers the vast majority of FP instructions in real-world code. Our approach is guided by the following observations:

- FP workloads operate mostly on normal or zero numbers. In other words, speeding up the handling of denormals, infinities or not-a-numbers (NaNs) is not necessary to accelerate most FP workloads.
- With some trivial checks, we can select FP operations capable of raising only three exceptions: inexact, overflow and underflow.
- FP flags are rarely cleared by FP workloads. This explains why FP flags are cumulative (or *sticky*). That is, once an exception occurs, the corresponding bit remains set until it is explicitly cleared by software.
- Due to FP's finite precision, most FP operations raise the inexact flag.
- FP workloads rarely change the rounding mode, which defaults to round-to-nearest-even.

We thus accelerate the *common case*, i.e.: the rounding is round-to-nearest-even, inexact is already set, and the operands are checked to limit the flags that the operation can raise. Otherwise, we defer to a soft-float implementation.

Figure 4.1 shows the application of our approach to double-precision multiplication. First, we flush the operands to zero if flush-to-zero mode is enabled (line 2). Next, we check whether this is the common case, checking both operands as well as the emulated FP environment (3-6). If so, we first perform a small optimization, checking for the trivial case of either operand being zero (7-10). As shown in Section 4.3.4, this can

```
0  float64 float64_mul(float64 a, float64 b, fp_status *st)
1  {
2    float64_input_flush2(&a, &b, st);
3    if (likely(float64_is_zero_or_normal(a) &&
4               float64_is_zero_or_normal(b) &&
5               st->exception_flags & FP_INEXACT &&
6               st->round_mode == FP_ROUND_NEAREST_EVEN)) {
7      if (float64_is_zero(a) || float64_is_zero(b)) {
8        bool neg = float64_is_neg(a) ^ float64_is_neg(b);
9        return float64_set_sign(float64_zero, neg);
10     } else {
11       double ha = float64_to_double(a);
12       double hb = float64_to_double(b);
13       double hr = ha * hb;
14       if (unlikely(isinf(hr))) {
15         st->float_exception_flags |= float_flag_overflow;
16       } else if (unlikely(fabs(hr) <= DBL_MIN)) {
17         goto soft_fp;
18       }
19       return double_to_float64(hr);
20     }
21   }
22 soft_fp:
23   return soft_float64_mul(a, b, st);
24 }
```

Figure 4.1: Pseudo-code of a Qelt-accelerated double-precision multiplication.

improve performance for some workloads, since we avoid accessing the host FPU's registers. The *else* branch leverages the host FPU to compute the result (12-14). Finally, we handle overflow (16-17) and resort to soft-float if there is a risk of underflow (18-19).

The key insight behind our technique is the identification of a large set of FP operations that can be run on the host FPU, while deferring corner cases (whether in the result or in the flags to be raised) to the slower soft-float code. We implement this technique in Qelt, accelerating commonly-used single and double-precision operations, namely addition, subtraction, multiplication, division, square root, comparison and fused multiply-add. Profiling shows that on average Qelt accelerates (i.e., does not defer to soft-float) 99.18% of FP instructions in SPECfp06 benchmarks.

Our approach has three main limitations. First, it does not speed up applications that frequently clear the inexact flag or that mostly operate with denormal numbers. Native hardware does not perform well in these cases either, so deferring to soft-float for these is appropriate. Second, applications with rounding other than round-to-nearest-even are not accelerated. Our approach could be changed to handle other rounding modes (particularly with regards to overflow), but we believe that the corresponding slowdown due to additional branches in the code is not justified, given how rare it is to find applications that require a non-default rounding mode. Last, while our approach does not require the guest or host to be IEEE-754 compliant (since compliance diverges only for operands outside of the *common case*), it requires the host FPU to natively support the same precision as that of the guest. This is, however, unlikely to be an issue in practice, since most FP workloads use only single and double precision, which are widely supported by commercial CPUs.

4.2.2 Scalable Dynamic Binary Translation

The design presented in Chapter 3 for DBT engines with a shared code cache enables parallel guest execution, which allows parallel workloads to scale when emulated on multi-core hosts. Focusing on making code *execution* parallel pays off, because the runtime of most DBT workloads is largely spent on code execution and not on translation.

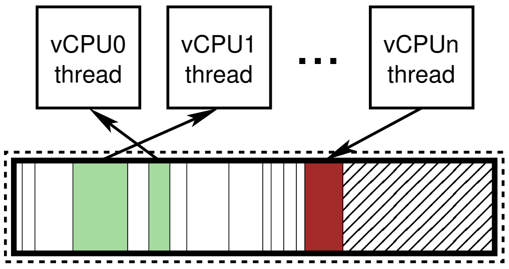
While this observation holds for many workloads, others can show significant amounts of parallel code *translation*. This scenario is typical in parallel server workloads [40], e.g. during parallel compilation jobs that require large amounts of guest code execution with little code reuse. In these cases, achieving scalability for unified code cache translators is challenging, since scalability is limited by the contention imposed by global locks protecting code generation and translation block (TB) chaining state [40, 87].

We address this challenge by starting from the design in Chapter 3, which is now part of upstream QEMU. In Qelt, we modify this baseline design, in which a single lock protects both code translation/invalidation as well as code chaining, to make each vCPU thread work—in the fast path—on uncontended cache lines. As we describe next, we achieve this by distributing state across vCPUs, and combining lock-free operations with fine-grained locks that are unlikely to be contended.

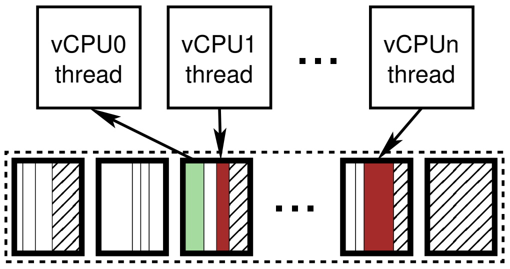
Translator state and code cache. We distribute the translator's state by replicating it across the vCPU threads. We keep the baseline's single, contiguous (in virtual memory) buffer for the code cache, since doing otherwise would greatly complicate cross-ISA code generation. However, we partition this buffer so that each vCPU generates code into a separate *region*. Figure 4.2 illustrates the impact of region partitioning: while a monolithic cache forces writers to be serialized, a partitioned cache allows vCPUs to generate code in parallel. Partitioning can reduce the effective size of the code cache, since vCPUs generate code at different rates. However, in most practical scenarios this reduction is negligible due to adequate region sizing, which accounts for the number of vCPUs and the size of the code cache.

Program counter (PC) TB lookups. PC TB lookups take the program counter (as a host virtual address) of some translated code and provide the corresponding TB descriptor. To serve these lookups we maintain a per-region binary search tree that tracks the beginning and end host addresses of the region's TBs. Operations on each of the trees are serialized with the same per-region lock used for writing code into the region. This has little to no impact on scalability, since PC TB lookups and TB invalidations are rare; the writer thread therefore acquires an uncontended lock, which is fast.

Physical memory map. Descriptors of guest memory pages are kept in the memory map, which is implemented as a radix tree. We modify this tree to support lock-free insertions, and rely on the fact that the tree already supports RCU for lookups (see Chapter 3). A spin lock is added to each descriptor to serialize accesses to the page's list of TBs. This list is accessed when TBs are either added or removed during code translation or invalidation, respectively. Some operations (e.g. invalidating a range of virtual pages) require atomic modifications over a range of non-contiguous



(a) Monolithic TB cache



(b) Partitioned TB cache

Figure 4.2: With a monolithic code cache (a), TB execution (green TBs) can happen in parallel, yet TB generation (red) is serialized. With partitioning (b), each vCPU writes to a separate region, thereby enabling both parallel code execution *and* translation.

physical pages. To avoid deadlock we acquire the locks of all the associated page descriptors in ascending order of physical address.

TB index. We rely on QHT (described in Section 3.2.3) to implement scalable TB bookkeeping. Accesses to the index are used as synchronization points. For instance, if two threads are contending to insert the same TB, the first one to complete the insertion into the hash table will win the race. The other thread will realize this at insertion time, subsequently undoing prior changes (e.g. insertion into the page's list of TBs) to then use the TB translated by the other thread.

Code chaining. TBs that are linked via direct jumps are chained together during code execution by patching the generated code to directly jump across translated code, thereby increasing performance. The linking and patching requires serialization to prevent chaining to a TB that is being invalidated and to protect the list of incoming TBs. Instead of relying on a global lock, we use a per-jump spinlock and add two tagged pointers to each TB descriptor to point to its two destination TBs. The pointers, which are accessed atomically with compare-and-swap, are tagged to ensure that no jumps from invalidated TBs can be linked.

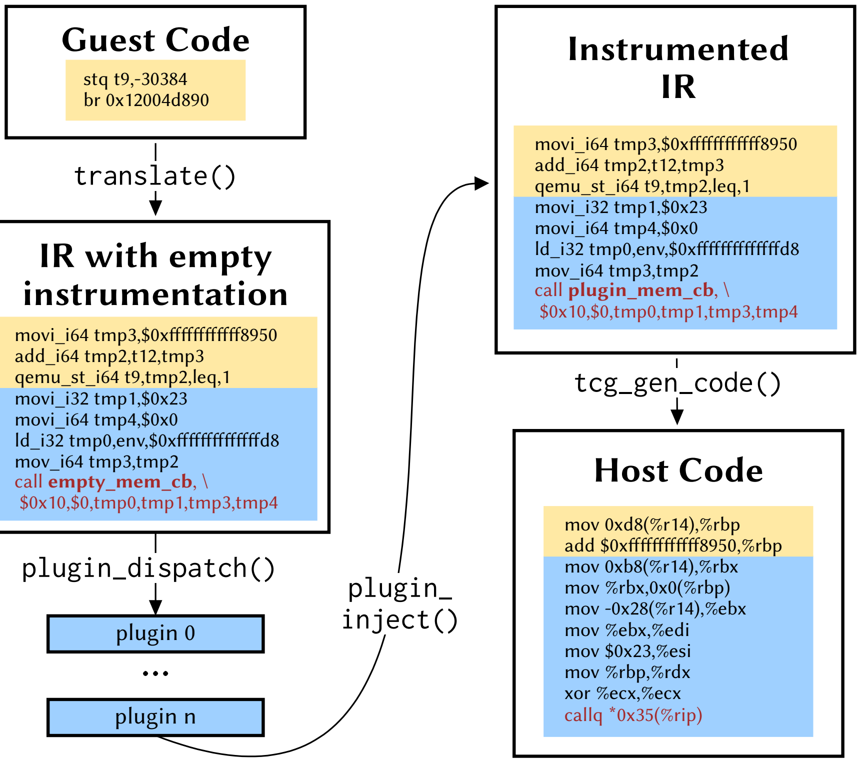
4.2.3 Portable Cross-ISA Instrumentation

We now describe Qelt's technique to convert a cross-ISA DBT engine into a cross-ISA DBI tool. This technique has four main properties. First, it provides injection points at an instruction level, which is in line with state-of-the-art instrumentation tools such as Pin and DynamoRIO. Second, it is ISA-agnostic, i.e., it remains portable by only requiring modifications to the ISA-independent parts of the DBT engine. Third, it is suitable for full-system instrumentation, since it can also instrument the side-effects of emulation performed via helpers. Last, it is high performance, with support for inline callbacks[2] and multiple event subscriptions.

ISA-agnostic instrumentation. Figure 4.3 shows the flow of a TB, from its creation to its translation into instrumented host code. The guest code snippet in Figure 2.2 is reused here; note that additional instrumentation-related code is added to both the IR and host code. In particular, this additional code implements a memory callback to a plugin.

The flow begins from a guest program counter from which code is to be executed. A single guest-to-IR translation pass is performed, and along the way, empty instrumentation is inserted. For instance, if a memory access is encountered, a callback to a placeholder "empty" memory callback is generated into the IR. Once the TB is fully formed, we dispatch it to plugins. Plugins add their instrumentation calls to the TB using the instrumentation API, which correspondingly annotates the TB's descriptor. We then perform the *injection* pass. That is, we go through all the empty instrumentation points, and either remove them from the IR if no subscriptions to them were requested, or copy them as needed (one copy per plugin request), substituting the "empty" placeholder with the plugin-supplied callback and/or data. The flow finishes by translating the instrumented IR into host code.

[2]Just like in tools such as Pin, instrumentation code is built into *plugins*, i.e. shared libraries that are dynamically loaded by the DBI tool. Plugins subscribe to guest events of their interest by registering functions (*callbacks*) that are called as soon as the appropriate event occurs.



```
stq t9,-30384
br 0x12004d890
```

```
movi_i64 tmp3,$0xffffffffffff8950
add_i64 tmp2,t12,tmp3
qemu_st_i64 t9,tmp2,leq,1
movi_i32 tmp1,$0x23
movi_i64 tmp4,$0x0
ld_i32 tmp0,env,$0xffffffffffffffd8
mov_i64 tmp3,tmp2
call empty_mem_cb, \
 $0x10,$0,tmp0,tmp1,tmp3,tmp4
```

```
movi_i64 tmp3,$0xffffffffffff8950
add_i64 tmp2,t12,tmp3
qemu_st_i64 t9,tmp2,leq,1
movi_i32 tmp1,$0x23
movi_i64 tmp4,$0x0
ld_i32 tmp0,env,$0xffffffffffffffd8
mov_i64 tmp3,tmp2
call plugin_mem_cb, \
 $0x10,$0,tmp0,tmp1,tmp3,tmp4
```

```
mov 0xd8(%r14),%rbp
add $0xffffffffffff8950,%rbp
mov 0xb8(%r14),%rbx
mov %rbx,0x0(%rbp)
mov -0x28(%r14),%ebx
mov %ebx,%edi
mov $0x23,%esi
mov %rbp,%rdx
xor %ecx,%ecx
callq *0x35(%rip)
```

Figure 4.3: Example instrumentation flow of Alpha-on-x86_64 emulation. We inject empty instrumentation as we translate the guest TB. Once the TB is well-defined, we dispatch it to plugins, which annotate it with instrumentation requests. Empty instrumentation is then either removed if unneeded or replaced with the plugin's requests. Finally, the instrumented IR is translated into host code.

Injection points. Instead of letting plugins inject instrumentation directly into the IR, we keep the IR entirely internal to the implementation, injecting during guest TB translation empty instrumentation that we can later remove if unneeded. This approach, which at first glance might seem wasteful in that it might perform unnecessary work, has several strengths:

- It enables the implementation of *instruction-grained* instrumentation, which lets plugins inject instrumentation for events associated to a particular instruction. For instance, a plugin can, *at translation time*, insert instrumentation before/after memory accesses associated with a particular instruction, instead of subscribing to *all* guest memory accesses and then selecting those of interest *at run-time*.

- It is ISA-agnostic, i.e., it requires no modifications to ISA-specific code. The instrumentation layer only modifies generic code, leaving instruction decoding to plugins.

- It incurs negligible cost. As we show in Section 4.3.5, on average the injection of empty instrumentation induces negligible overhead.

Event subscriptions. We distinguish between two event types: *regular* and *dynamic*. Dynamic events are related to guest execution (e.g. memory accesses, TBs executed), and therefore occur extremely frequently. Regular events are all others, e.g. vCPU thread starts/stops, or TBs being translated. We expand later on dynamic events, whose delivery we optimize with *inlining* and *direct callbacks*. Regular subscriptions are kept in per-event RCU [174] lists, which make callback delivery fast and scalable. RCU is a fitting tool for this purpose due to the *read-mostly* nature of the access pattern: list traversals (i.e. callbacks) strongly outnumber list insertions/removals (i.e. subscription registrations/cancellations).

Helper instrumentation. Instrumenting helpers is challenging, since at translation time we do not know what they implement or when they will execute. The magnitude of the challenge increases when we consider the amount of helpers that might be in a code base. For instance, each of the 22 target translators in QEMU uses, on average, more than a hundred helpers. Thus, the straw man solution of modifying thousands of helpers to add instrumentation-related code becomes a tedious and error-prone prospect.

We present a low-overhead approach to instrument helpers that is more practical. Our approach, which we apply to memory accesses performed by helpers, relies on the following observation. Guest memory accesses from helpers are performed via a handful common interfaces. Thus, we modify those common interfaces—about a dozen call sites in QEMU—instead of editing potentially thousands of helpers.

Most of the work is done at translation time. We start by tracking which guest instructions have emitted helpers. For each of these instructions, if plugins have subscribed to their memory accesses, we proceed in two steps. First, we allocate the subscription requests from the appropriate plugins into an array, which we track using QHT (see Section 3.2.3) so that it can be freed once the TB is invalidated. Second, we inject IR code before the guest instruction that sets *helper_mem_cb*, which is a field in the state of the vCPU that will execute the helper, to point to the subscription array. We also insert code after the instruction to clear this field.

At execution time, we rely on our modified common interfaces for accessing memory from helpers. Thus, when an instrumented helper accesses memory, the generic memory access code checks the executing vCPU's *helper_mem_cb* field, and if set, delivers the callbacks to plugins.

Inlining. We support manual inlining of instrumentation code for *dynamic* events. Plugins can explicitly insert inline operations, which they choose via the plugin API. These operations implement typical actions needed by instrumentation code, such as setting a variable or incrementing a counter, and are independent from the IR, since the latter is internal to the translator's implementation and therefore always subject to change. In Section 4.3.6 we show how inlining can increase performance for instrumentation-heavy workloads.

Direct callbacks. We treat dynamic events differently from regular ones. The reason is performance: dynamic events—such as memory accesses or TBs/instructions executed—can be generated *extremely* frequently, and therefore the overhead of instrumenting these events can easily dominate execution time. Although inlining can help mitigate this overhead, complex instrumentation (e.g. code that inserts an address into a hash table) cannot benefit from it, which brings our focus to the performance of callback delivery.

Most existing cross-ISA DBI tools deliver dynamic event callbacks using an intermediate helper that iterates over a list of subscriptions [88, 89, 113]. This is convenient from an implementation viewpoint, but introduces an unnecessary level of indirection. We eliminate this indirection by leveraging the injected empty instrumentation, which allows us to embed callbacks *directly* in the generated code. As we show in

Section 4.3.5, direct callbacks result in better performance over delivering callbacks from an intermediate helper. They, however, complicate subscription management. To cancel a direct callback subscription, instead of just updating a list as in regular callbacks, we must re-translate the TB. This, while costly, is not a practical concern, since instruction-grained injection points virtually eliminate the need for frequent subscription cancellations from dynamic events.

An example Qelt plugin. Figure 4.4 shows an example Qelt instrumentation plugin that counts guest memory accesses. Execution begins in the plugin at load time, with Qelt calling the `plugin_install` function. The plugin subscribes to two events: TB translations and guest exit—i.e., termination of a user-program or shutdown of a full-system guest.

Instrumentation of guest code occurs in `vcpu_tb_translate`. For each instruction in a TB, instrumentation is added after the instruction's memory accesses, if any. Depending on `do_inline`'s value, instrumentation is either via a direct callback to `vcpu_mem` or through an inline increment to the counter, `mem_count`. Note that to keep the example simple, the counter's increment is not implemented with an atomic operation, which could result in missed counts when instrumenting parallel guests.

We conclude by discussing three points that are not obvious from the example. First, the API exposes no ISA-specific knowledge. For example, instructions are treated as opaque objects; this requires plugins that need instruction information to rely on an external disassembler, but as we show in Section 4.3.5 this has negligible overhead. Second, vCPU registers can be queried from callbacks. The example does not use this feature, and thus it disables register copying at callback time with the `CB_NO_REGS` flag. Third, instead of supporting user-defined functions like Pin [160] or QTrace [230] do, we attach a *user data* pointer (`udata`) to direct callbacks, which achieves the flexibility of user-defined functions while being more portable and simpler to implement.

```c
static uint64_t mem_count;
static bool do_inline;

static void plugin_exit(plugin_id_t id, void *p)
{
  printf("mem accesses: %" PRIu64 "\n", mem_count);
}

static void vcpu_mem(unsigned int cpu_index,
  plugin_meminfo_t meminfo, uint64_t vaddr, void *udata)
{
  mem_count++;
}

static void vcpu_tb_translate(plugin_id_t id,
  unsigned int cpu_index, struct plugin_tb *tb)
{
  size_t n = plugin_tb_n_insns(tb);
  size_t i;
  for (i = 0; i < n; i++) {
    struct plugin_insn *insn = plugin_tb_get_insn(tb, i);
    if (do_inline) {
      plugin_register_vcpu_mem_inline__after(
        insn, PLUGIN_INLINE_ADD_U64, &mem_count, 1);
    } else {
      plugin_register_vcpu_mem_cb__after(
        insn, vcpu_mem, PLUGIN_CB_NO_REGS, NULL);
    }
  }
}

int plugin_install(plugin_id_t id, int argc, char **argv)
{
  if (argc && strcmp(argv[0], "inline") == 0)
    do_inline = true;
  plugin_register_vcpu_tb_trans_cb(id, vcpu_tb_translate);
  plugin_register_atexit_cb(id, plugin_exit, NULL);
  return 0;
}
```

Figure 4.4: Example Qelt plugin to count memory accesses either via a callback or by inlining the count.

4.2.4 Additional DBT Optimizations

We now describe DBT optimizations implemented in Qelt that are derived from those in state-of-the-art DBT engines.

4.2.4.1 TLB Emulation

Guest TLB emulation in full-system emulators is a large contributor to performance overhead. As discussed in Section 2.2.2, the softMMU maps guest virtual addresses to host virtual addresses. SoftMMU overhead comes from three sources, which we list in order of importance. First, non-compulsory misses in the TLB result in guest page faults, which take hundreds of host instructions to execute. Second, even if the TLB miss rate is low, hits still incur non-negligible latency, since each guest memory access is translated into several host instructions which index and compare the contents of the TLB against the appropriate portion of the guest virtual address. And third, clearing the TLB on a flush can also incur non-trivial overhead due to frequently-occurring flushes. It is for this reason that QEMU has a small, static TLB size.

Tong et al. [231] present a detailed study in which they evaluate different options to mitigate softMMU overhead. One of these options is to resize the TLB depending on the workload's requirements. They resize the TLB only during flushes, since doing it at any other time would require rehashing the table, which is expensive. They propose a simple resizing policy: if the TLB use rate at flush time is above a certain upper threshold (e.g. 70%), double the TLB size; if the rate is below a certain lower threshold (e.g. 30%), halve it. Note that the upper threshold should not ever be too close to 100%, for otherwise we are at risk of incurring a large amount of conflict misses, given that the softMMU is direct-mapped to keep TLB lookup latency low. In addition, computing the table index dynamically incurs a slight lookup latency increase. The rationale, however, is that the reduced number of misses is likely to amortize this additional cost.

We observe that this policy can lead to overly aggressive resizing. This can be illustrated with two alternating processes, of which one is memory-hungry and the other uses little memory. With this policy, when the guest schedules out the memory-hungry process, the TLB size doubles, yet the next process will not make much use of it, which will induce a downsize. This results in a sequence of TLB size doubling/halving, which neither process can benefit from.

We improve upon this policy by incorporating history into it. We track the maximum use rate in the most recent past (e.g. a *history window* of 100ms), and resize based on that maximum observed use rate. The rationale is that if a memory-hungry process has been recently scheduled, it is likely that it will be scheduled again in the near future. This can result in an oversized TLB for processes that are scheduled next, but this cost is likely to be offset by an eventual reduction in overall misses. In other words, we incorporate some history into the policy to perform the same aggressive upsizing, yet downsize more conservatively. In Section 4.3.2 we compare these two policies, with 70%-30% thresholds and a history window of 100ms, against QEMU's static TLB.

4.2.4.2 Indirect branches in DBT

Baseline QEMU handles indirect branches by simply returning to the dispatch loop. As discussed in Section 2.2.3.1, Hong et al. [117] present an approach for QEMU that leverages caching to minimize expensive exits to the dispatcher. In Qelt we follow an approach similar to theirs, with four differences. First, instead of adding a cache solely for this purpose, we reuse the existing *TB jmp cache*, a small, direct-mapped private cache that is always accessed first by vCPU threads when searching for a TB (see Figure 3.2b). Second, we improve the hashing function used in user-mode to access the *jmp cache*, resulting in an effective capacity increase. Third, we perform a lookup in the global TB hash table after missing in the *TB jmp cache*, which reduces the exits to the dispatcher to only compulsory cases (i.e. invalidated or previously untranslated TBs) and makes performance less sensitive to the sizing of the *jmp cache*. Last, we abstract these operations by adding an instruction (*"lookup and goto ptr"*) to the TCG IR, which minimizes the amount of target and host-specific code needed to support this optimization[3]. Performance-wise, our results (Section 4.3.2) are comparable to those reported by Hong et al. [117]; note however that a fair quantitative comparison would be difficult, given that the baseline QEMU implementations used are several years apart.

[3]The use of an IR operation allowed the QEMU community to quickly expand on our work: within days of the merging of our implementation into QEMU v2.11, the number of target and host ISAs that implement `lookup_and_goto_ptr` grew from 3 and 1 to 8 and 7, respectively.

4.3 Evaluation

4.3.1 Setup

Host. We run all experiments on a host machine with two 2.6GHz, 16-core Intel Xeon Gold 6142 processors, for a total of 32 cores. The machine has 384GB of RAM, and runs Ubuntu 18.04 with Linux kernel v4.15. We compile all source code with GCC v8.2.0 with `-O2` flags.

Workloads. We measure single-threaded performance with SPEC06's *test* set, except for libquantum, xalancbmk, gamess, soplex and calculix. For these workloads we use the *train* set, since *test* is too lightweight (e.g. *libquantum* runs natively under 0.02s) for us to draw meaningful conclusions when running them under different DBT engines.

We run all experiments multiple times. We report the measured mean as well as error bars or shaded regions (for bar charts or line plots, respectively) representing the 95% confidence interval around the mean.

Guest ISA. We use x86_64 guest workloads, which allows us to compare against existing DBI tools (all of them support x86_64) and to benchmark against native runs on the host machine, including full-system guests virtualized with KVM.

QEMU baseline. Our QEMU baseline is derived from QEMU v3.1. Given that several of Qelt's techniques are already part of QEMU v3.1, our baseline (hereafter *QEMU*) is the result of reverting their corresponding changes from v3.1.

4.3.2 Performance Impact of Qelt's Techniques

We begin our evaluation by characterizing the performance impact of implementing Qelt's techniques in sequence on top of QEMU when running single-threaded guest workloads.

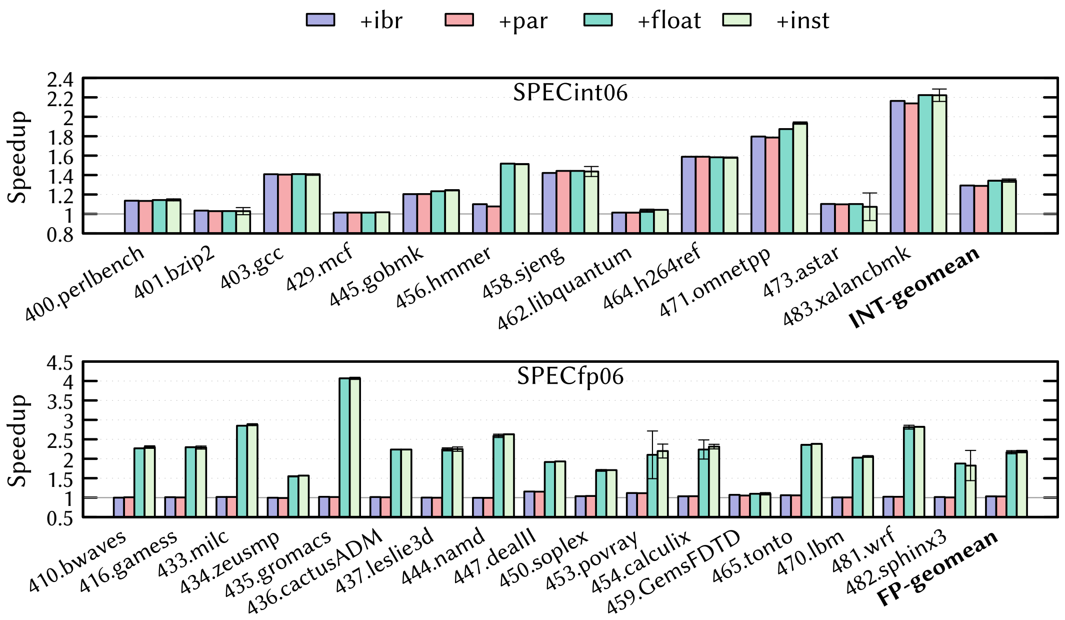


Figure 4.5: Cumulative speedup of Qelt's techniques over QEMU for user-mode x86_64 SPEC06.

Figure 4.5 shows the resulting speedup for user-mode x86_64 SPEC06. Qelt's indirect branch optimizations (+ibr, Section 4.2.4.2) yield an average 29% performance gain for integer workloads. Qelt's parallel code generation (+par, Section 4.2.2) and instrumentation support (+inst, Section 4.2.3) show negligible performance impact. Last, Qelt's FP emulation improvements (+float, Section 4.2.1) shows the largest improvement: SPECfp06's performance increases more than 2× on average.

Figure 4.6 shows Qelt's speedup over QEMU for full-system x86_64 SPEC06. The techniques presented in Figure 4.5 are combined as Qelt-statTLB. Their resulting speedup is lower than in user-mode due to the overhead of full-system mode's softMMU. Adding a TLB resizing policy based solely on the TLB use rate at flush time (+dynTLB, as described in [231]) results in a slowdown on average, since the system also runs system processes with low memory demands. Qelt's policy (+history, Section 4.2.4.1) bases its resizing decisions on the use rate over the recent past, which leads to overall mean speedups of 1.76× and 2.18× for integer and FP workloads, respectively.

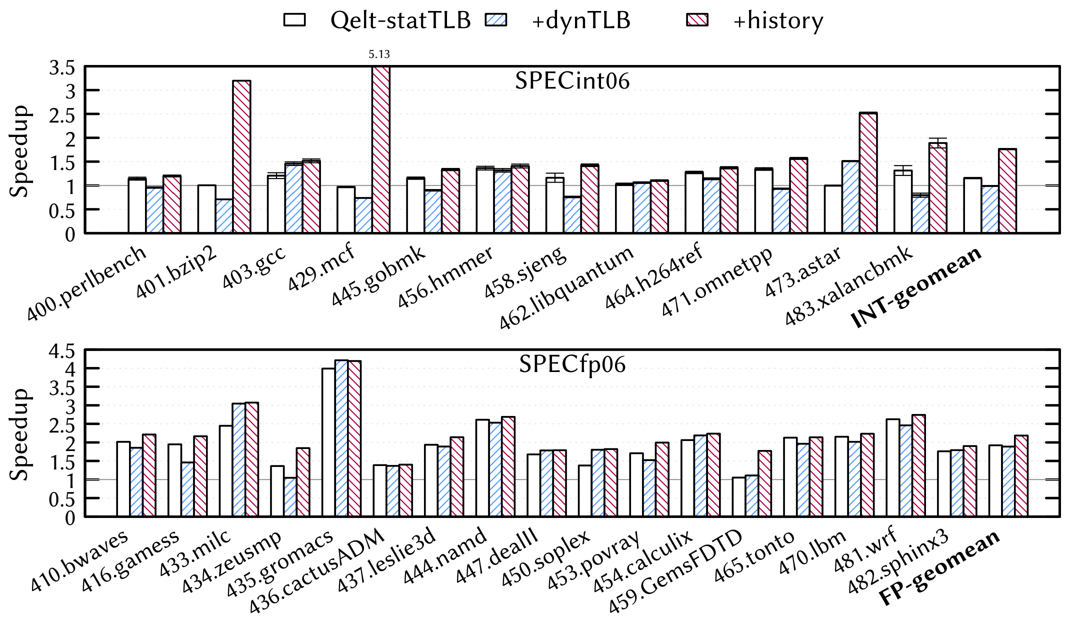


Figure 4.6: Cumulative speedup of Qelt's techniques over QEMU for full-system x86_64 SPEC06.

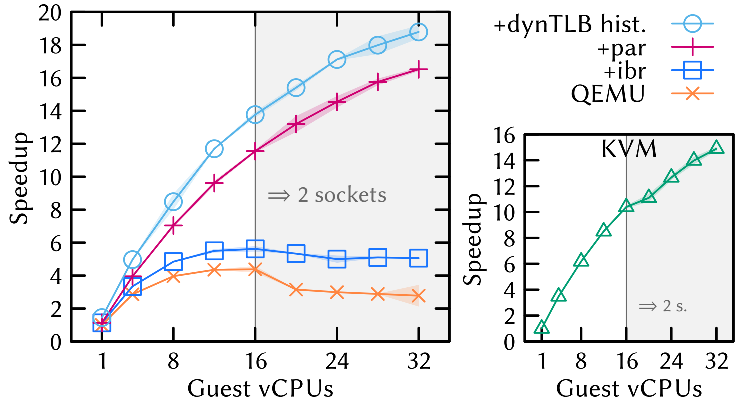


Figure 4.7: Cumulative Qelt speedup over 1-vCPU QEMU for parallel compilation inside an x86_64 VM. On the right, KVM scalability for the same workload.

4.3.3 Scalable Dynamic Binary Translation

We now evaluate Qelt's scalability with a workload that requires large amounts of parallel DBT. For this we build 189 Linux v4.19.1 kernel modules with `make -j N` (where N is the number of guest cores) inside an x86_64 virtual machine (VM) running

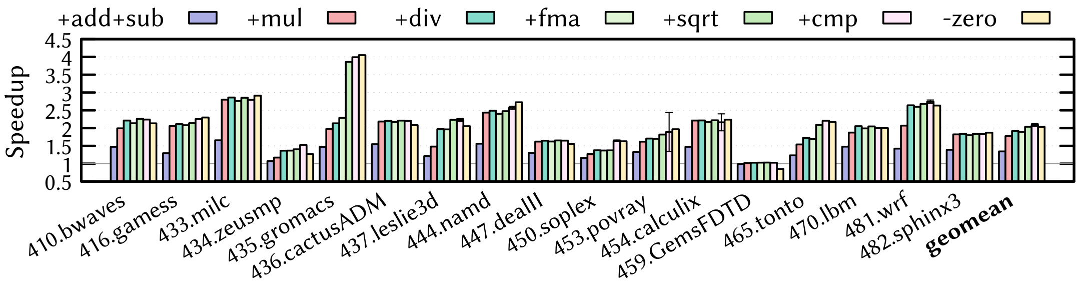


Figure 4.8: Cumulative speedup over QEMU of accelerating the emulation FP instructions with Qelt for user-mode x86_64 SPECfp06. The -zero results show the impact of removing Qelt's zero-input optimization.

Ubuntu 18.04. Figure 4.7 (left plot) shows the results, which are normalized over those of QEMU with N=1. QEMU shows poor scalability, with a maximum speedup of 4× at 16 cores. Qelt's indirect branch optimizations (+ibr) slightly improve performance, but do not address the underlying scalability bottleneck. Qelt's parallel translator (+par, Section 4.2.2) brings scalability in line with that of KVM (right plot), for a maximum speedup above 16× at 32 cores. Scalability is further improved with Qelt's dynamic TLB resizing (+dynTLB hist.), which brings the overall speedup up to 18.78×.

4.3.4 Fast FP Emulation using the Host FPU

Qelt accelerates the following operations, in both single and double precision: addition, subtraction, multiplication, division, square root, comparison and fused multiply-add (FMA). We validate our implementation against real hardware (ppc64, Aarch64 and x86_64 hosts) as well as against Berkeley's Testfloat v3e [110] and IEEE-754-compliant test patterns from IBM's FPgen [27].

Figure 4.8 shows the speedup of accelerating FP emulation with Qelt for SPECfp06 in user-mode x86_64. The results shown are cumulative, which reveals the impact of accelerating each group of FP instructions separately. Note that QEMU's FP operations are implemented as a C library that is shared by all ISA translators. Thus, while the final SPECfp06 speedup is similar across all ISAs, the broken-down speedups are specific to each translator. For instance, the x86_64 translator (shown here) is sensi-

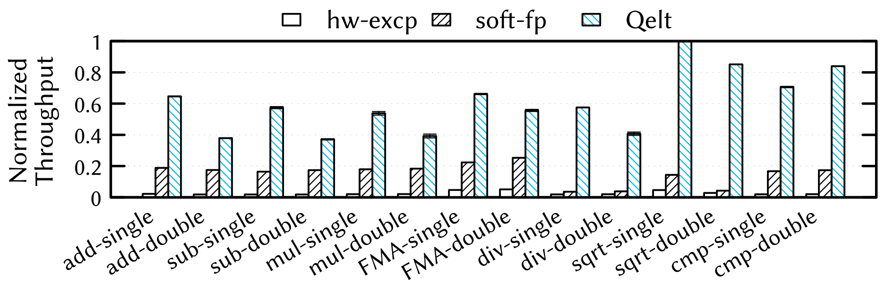


Figure 4.9: FP microbenchmark results. Throughput is normalized over that of an ideal native run.

tive to changes to both addition and multiplication's performance, whereas Aarch64's (results not shown) is most sensitive to FMA. The last set of results (-zero) shows the effect of removing the zero-input optimization in Figure 4.1. Its removal hurts average performance, since some benchmarks frequently execute zero-input FP operations (e.g., cactusADM, GemsFDTD).

FP Microbenchmark. The above results depend on the distribution of FP instructions in SPECfp06. To better understand Qelt's impact on FP emulation, we wrote a microbenchmark that feeds random *normal* FP operations to the FP emulation library. Figure 4.9 shows the resulting throughput, normalized over that of an ideal, incorrect run on the host, i.e. without any checks on either the result or FP flags.

The first set of results (hw-excp) in Figure 4.9 corresponds to a naïve implementation that, as described in Section 4.2.1, for each FP instruction first clears the host's FP flags (with `feclearexcept(3)`), then executes the FP operation on the host, and finally checks the host FP flags (`fetestexcept(3)`). This approach has poor performance, even when compared against QEMU's soft-float implementation (soft-fp). We have reproduced this on other machines as well, which suggests that FPUs are optimized for fast, overlapping execution of FP instructions, and not for frequent FP flag checks.

Qelt improves performance over soft-float, with speedups ranging from 2.16× (mul-double) to 19.84× (sqrt-double). The performance gap between Qelt and ideal FP performance quantifies the cost of correctness; recall from Figure 4.1 that we have

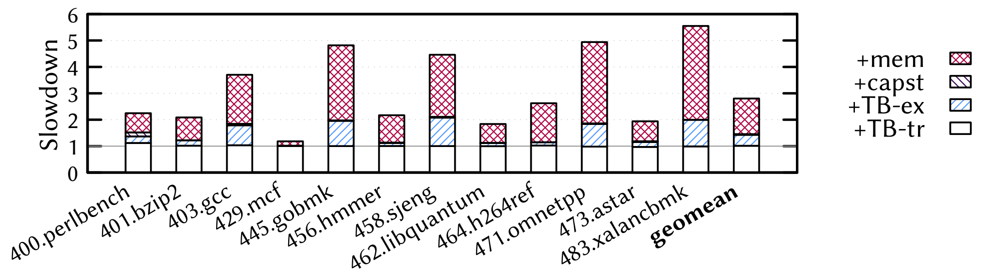


Figure 4.10: Impact of increasing instrumentation on user-mode x86_64 SPECint06.

to perform checks on the input as well as on the computed output (to detect under/overflow). The latter checks, however, are not needed for comparison and square root (a non-negative normal-or-zero square root cannot under/overflow), which explains the narrow performance gap between them and the ideal implementation.

4.3.5 Instrumentation

We now characterize the performance of Qelt's instrumentation layer. We first analyze the overhead of typical instrumentation plugins, and then evaluate the impact of different *direct callback* implementations.

Overhead. Figure 4.10 shows Qelt's slowdown over the baseline for typical instrumentation plugins, broken down per instrumented event. Subscribing to TB translation events (+TB-tr) incurs negligible average overhead (1.1%), with perlbench showing the maximum overhead (12%) since it is the workload that executes the most guest code.

Subscribing to TB execution callbacks (+TB-ex) has significant overhead (mean 41%, maximum 107% for *sjeng*). The overhead is caused by the high frequency of guest TB execution and, therefore, TB execution callbacks. It is for this reason that instrumentation is preferably done at translation time whenever possible. The "capstone" plugin (+capst) is an example of translation-time processing that mimics what an architectural simulator would do during decode: it disassembles each translated TB using Capstone [200] and then allocates a per-TB descriptor to be passed to the

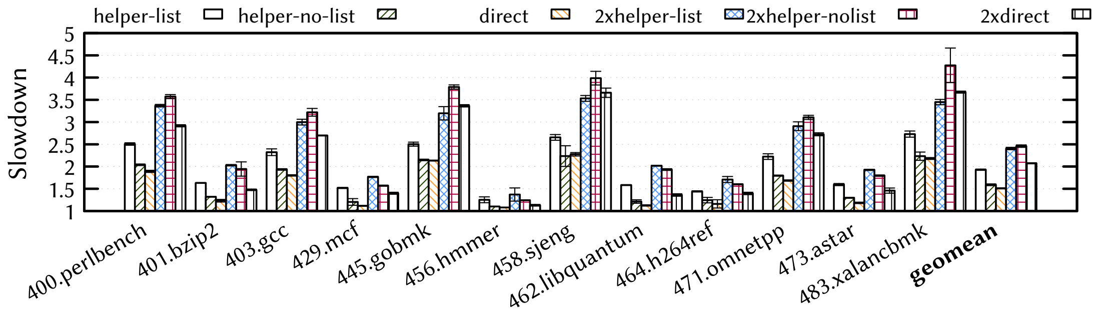


Figure 4.11: Slowdown of user-mode x86_64 SPECint06 for helper-based and direct callbacks.

TB execution callback. This translation-time processing incurs negligible additional overhead (mean 2.6%), which supports our decision to not export via the plugin API any interfaces with ISA-specific knowledge of the target's instructions.

Memory callbacks (+mem) incur large overhead due to their high frequency. Fortunately, this cost can be mitigated with inlining, as shown in Section 4.3.6.

Direct callbacks. We now discuss the impact of instrumenting *dynamic* (i.e. high-frequency) events. Figure 4.11 compares the instrumentation of a dynamic event (TB execution) for three different implementations and either one or two (denoted with the 2x prefix) plugin subscriptions. The helper-list implementation uses a *helper* function from which callbacks are dispatched by iterating over a list of subscribers. When the event has a single subscriber, it pays off to avoid the list altogether, which improves performance—as helper-nolist shows—due to increased cache locality. An additional improvement is obtained by using direct callbacks (direct), which incur one less function call (i.e., the helper) per event.

With two subscribers, helper-list performs three function calls per event, for helper-nolist's four. However, the former's subsequent gains are canceled out by iterating over the subscribers' list (2.40× vs. 2.45× mean slowdown), which has poor data cache locality. Using direct callbacks outperforms them both (2.07× slowdown), since it incurs one function call per subscriber and has optimal data cache locality.

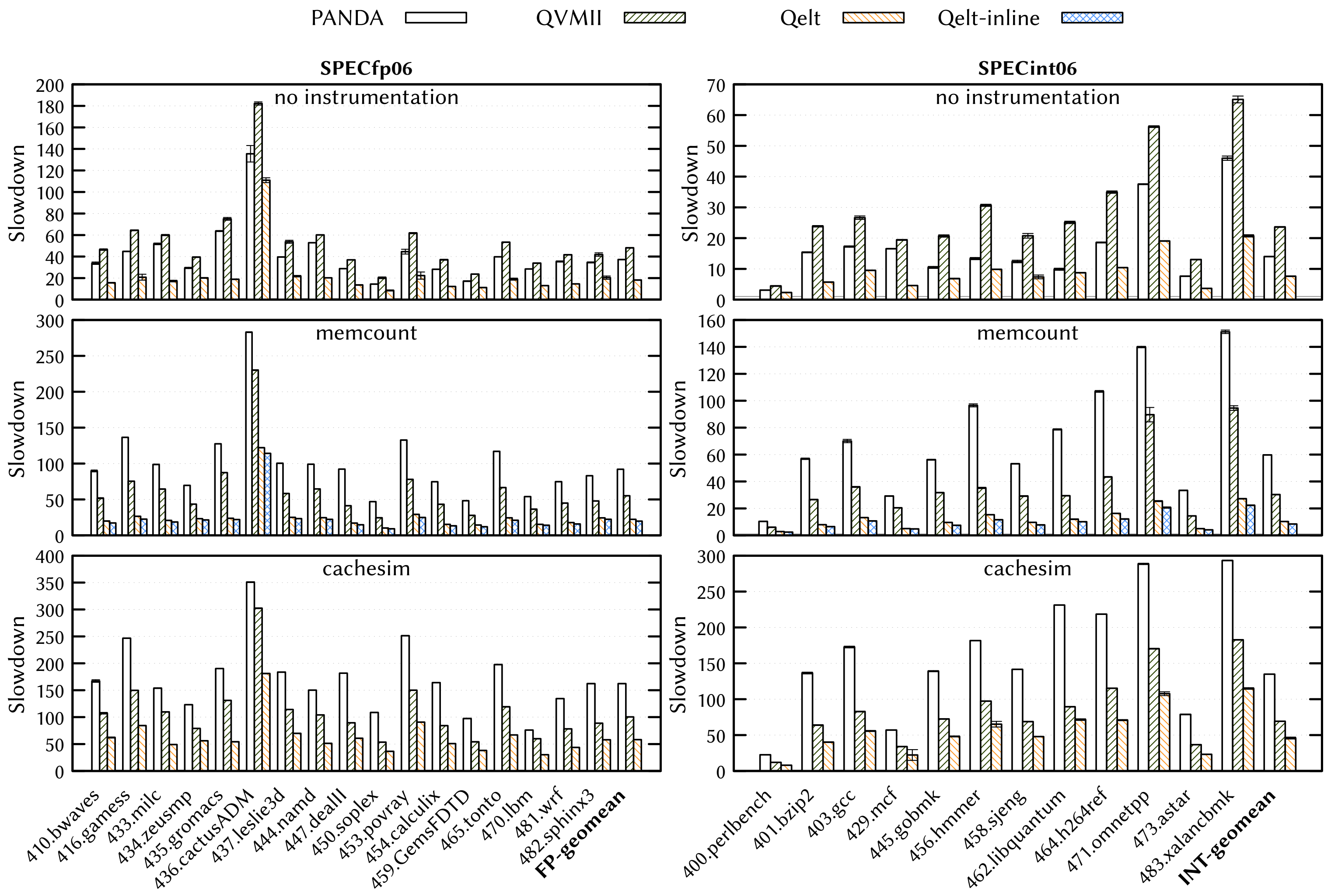


Figure 4.12: Slowdown over KVM execution for PANDA, QVMII and Qelt for full-system emulation of x86_64 SPEC06.

4.3.6 DBI Tool Comparison

We conclude our evaluation by comparing Qelt against the state of the art in full-system and user-mode DBI tools.

Full-System DBI. We compare Qelt against PANDA [88] (version d886146, Aug 3 2018) and QVMII [89] (dc7d35d, Jul 18 2018). We also considered other QEMU-derived tools such as QTrace [230], PEMU [247] and DECAF [113], but discarded them due to their slow baseline emulation performance (we measured QTrace to be on average 11.3× slower than Qelt for SPEC06int, and [247] reports a 4.33× average slowdown of PEMU over QEMU) or lack of support for x86_64 (DECAF). Figure 4.12 shows the resulting slowdown over using KVM to virtualize an x86_64 VM running SPEC06. For both integer and FP workloads (top row), Qelt is the fastest emulator, with PANDA coming second with performance similar to that of baseline QEMU (PANDA's fork point is close to our QEMU baseline, which we evaluated in Figure 4.6). QVMII is the slowest, since by default it instruments all memory accesses in case any plugins subscribe to them.

Instrumenting the execution of memory accesses with a counter increment (memcount) shows the differences in instrumentation overhead. PANDA lags behind because for simplicity of the implementation it disables key QEMU optimizations (translation block chaining and TCG softMMU lookups [24], respectively). Qelt is 3×/2.5× faster than QVMII for integer/FP workloads, with a slight improvement with inlining, a feature not supported by the other tools. The gap between QVMII and Qelt is explained by their different baseline emulation performance as well as Qelt's use of direct callbacks instead of QVMII's helper-based approach.

We then perform heavy instrumentation (cachesim), similar to what an architectural simulator would do. We simulate L1 instruction and data caches, without a directory and with an LRU set eviction policy. This is implemented by instrumenting all memory accesses, as well as simulating the corresponding instruction cache accesses when a basic block executes. Instrumentation now dominates execution time for SPECint06, making QVMII 1.53× slower than Qelt. This performance gap reduction vs. baseline emulation is less pronounced for SPECfp06; Qelt is 1.72× faster than QVMII thanks to Qelt's improved FP performance.

User-Mode DBI. Figure 4.13 compares Qelt against Pin [160] (v3.7-97619, May 8 2018) and DynamoRIO (v7.0.17735-0, Jul 30 2018). These tools are not cross-ISA, which to a large extent explains their higher performance for pure emulation (top row). DynamoRIO and Pin stay below or close to 2× slowdown over native, while Qelt is 4.20×/13.89× slower than native for integer/FP. Note that despite Qelt's FP improvement over QEMU, it still requires a *helper* function call to the FP library on every guest FP instruction, which explains the large FP gap vs. Pin/DynamoRIO.

Inlining is key to DynamoRIO's performance when instrumenting frequent events. This is a consequence of DynamoRIO's flexibility, since it allows plugin developers to make arbitrary changes to the guest code stream. Unfortunately, when inlining cannot be performed (either by disabling it or when a function is too complex to be inlined), constructing a callback involves a significant amount of work: "switching to a safe stack, saving all registers, materializing the arguments, and jumping to the callback" [140]. On the other hand, tools like Pin in its "classic" mode (which we use) or Qelt do not allow arbitrary guest code modifications, and therefore can efficiently insert a call/trampoline to a plugin. This explains DynamoRIO's large overhead in memcount, whereas it is the fastest tool for inline memcount.

For memcount, Pin is in most cases faster than Qelt, although Pin's well-known performance issues with large instruction footprints (e.g., perlbench, gcc) [160] bring Pin's SPECint06 mean slowdown slightly above Qelt's for both out-of-line and inline memcount's. For SPECfp06 memcount, Qelt is only slightly slower than Pin, since the frequent callbacks dominate execution time. However, with inlining Qelt is slower than Pin/DynamoRIO, because of its slower FP emulation.

The callbacks in cachesim are too complex to be inlined, which explains DynamoRIO's large slowdown. On average, Qelt's cachesim performance is similar to Pin's. This is due to cachesim's substantial overhead, which dominates over the difference in emulation speed between Qelt and Pin.

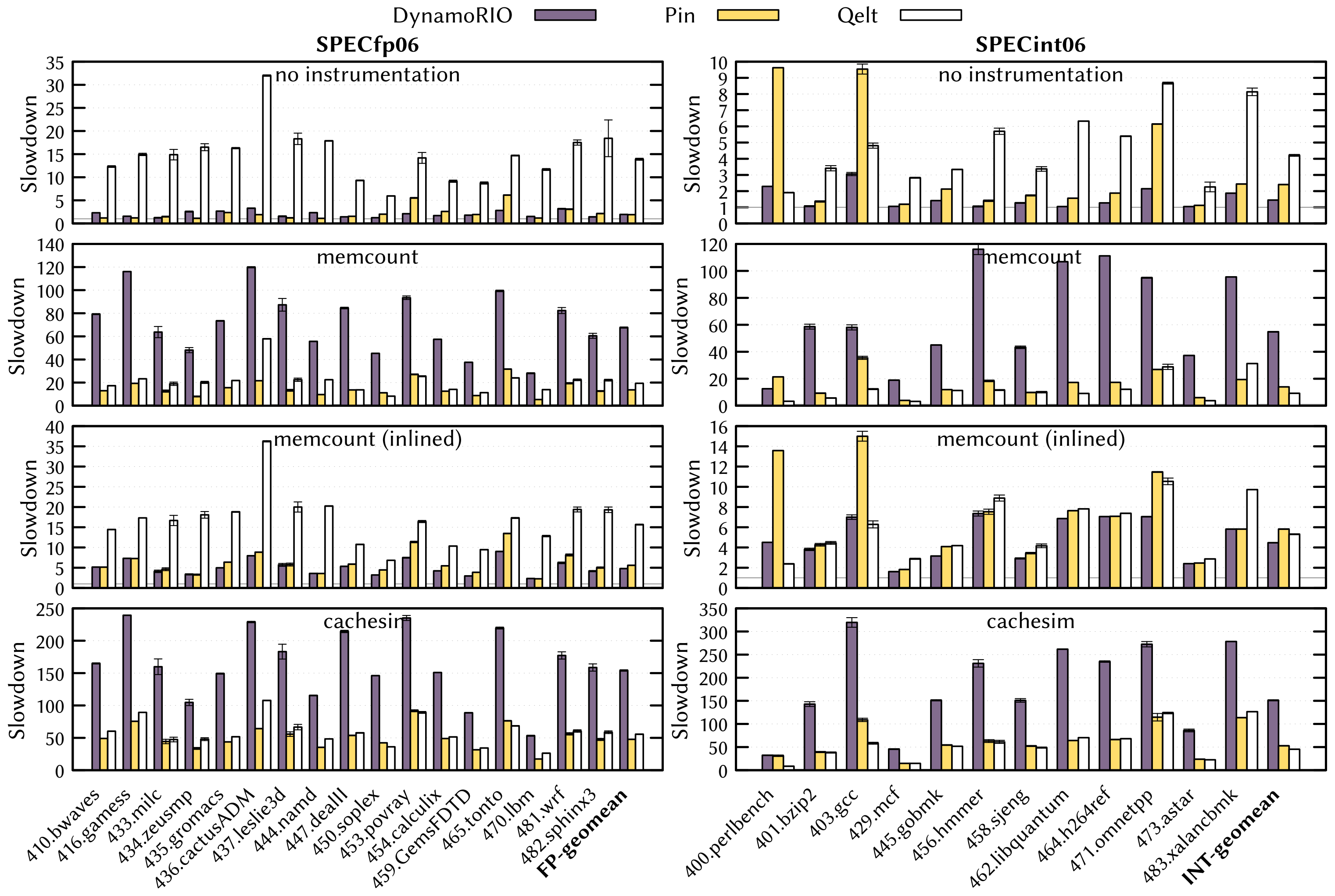


Figure 4.13: Slowdown over native execution for DynamoRIO, Pin and Qelt for user-mode x86_64 SPEC06.

4.4 Summary

In this chapter we have presented two novel techniques to increase cross-ISA DBT emulation performance: fast FP emulation leveraging the host FPU and scalable DBT generation and chaining for emulating multi-core guests. We have also introduced a novel ISA-agnostic instrumentation layer, which can be used to convert cross-ISA DBT engines into cross-ISA DBI tools.

We combined these techniques together with further DBT optimizations to build Qelt, a cross-ISA machine emulator and DBI tool that outperforms the state of the art in both cross-ISA emulators and DBI tools. Further, Qelt can match the performance of Pin, a state-of-the-art, same-ISA DBI tool, when performing complex instrumentation such as cache simulation.

Except for the instrumentation layer, which is currently under review by the QEMU community, Qelt's implementation has been merged into upstream QEMU during the development of the v4.0 version, which is scheduled for release in April 2019.

Chapter 5

Accelerator Coupling in Heterogeneous Architectures

In this chapter[1] we present a study that compares different accelerator coupling models, which we briefly introduced in Section 2.1. The study is enabled by Cargo, a machine simulator that we developed to model systems that integrate accelerators. Cargo allowed us to quantitatively compare accelerator coupling models and to also reason about their impact on system software (e.g., kernel-space drivers), which highlights the usefulness of machine emulation for conducting research on heterogeneous systems.

5.1 Introduction

As discussed in Chapter 1, fixed power budgets and the end of Dennard scaling have led researchers to embrace accelerators in order to sustain performance and energy efficiency increases. Thus, research on accelerating relevant applications is ongoing (e.g., [50, 242]), and accelerator-rich architectures are on the horizon [46, 63, 233]. Unfortunately, most existing research on accelerators has focused on computational aspects and has disregarded design decisions with practical implications, such as the

[1]This chapter incorporates and extends work previously published in the proceedings of the 2015 DAC conference [69].

model for accelerator invocation from software and the interaction between accelerators and the components (e.g. general-purpose cores, caches) surrounding them.

In this chapter we attempt to shed some light on these issues by developing seven high-throughput accelerators and the software to drive them. We designed each accelerator conforming to three design models: tight coupling behind a CPU, loose out-of-core coupling with Direct Memory Access to the last-level cache (hereafter *LLC-DMA*), and loose out-of-core coupling with DMA to DRAM (*DRAM-DMA*).

Our experiments on these accelerators induce the following observations.

- Private local memories (PLMs) are key to performance and energy efficiency. Accelerator logic is capable of processing vast amounts of data provided they can be fed at high throughput. This is hardly achievable attaching accelerators to CPU caches; most accelerators can exploit parallelism in the algorithms they implement and thus require many-ported memories tailored to their needs. This makes loosely-coupled accelerators better suited to high-throughput applications than tightly-coupled accelerators.

- DRAM bandwidth can quickly become a bottleneck. Making an accelerator rely on the LLC can help in certain cases given its higher bandwidth over DRAM. However, when the working set is of streaming nature, LLC thrashing occurs and DRAM bandwidth becomes the dominant bottleneck.

- Cache pollution plays a minor role when comparing LLC-DMA vs. DRAM-DMA loosely-coupled accelerators. LLC-DMA has slightly higher performance and significantly higher energy efficiency when the workload can fit in the LLC. Further, the LLC can mitigate DRAM saturation under high accelerator activity.

- The run-time overhead of abstracting loosely-coupled accelerators as just SoC-like devices becomes negligible once the granularity of the acceleration (i.e. working set size) becomes non-trivial. Moreover, abstracting accelerators using device drivers is not conceptually more complex than the alternatives, such as expanding the ISA to invoke accelerators tightly-coupled with the CPU.

In summary, this chapter's main contribution is an extensive study on the design and integration of high-throughput accelerators, with a focus on rarely-treated aspects such as coupling with the rest of the system and its impact on software. Results from our experiments can help future designers as well as increase understanding of existing accelerator implementations.

5.2 Accelerator Models

Our analysis focuses on configurable, non-programmable accelerators for applications that have high-throughput requirements. We study the implementation of accelerators following three models, which we present in this section.

Tightly-coupled accelerators (TCAs). They consist of one or more specialized hardware functional units which can accelerate critical portions of an application kernel; for example, the body of an inner loop for an algorithm or a sequence of trigonometric functions. This type of accelerator is located inside, or very close to, the processing core [224]. Figure 5.1 shows a diagram of a tightly-coupled accelerator (TCA) integrated in a CPU core. The core shares key resources (register file, memory management unit (MMU) and L1 data cache) with the TCA, and thus stalls until the TCA completes execution. TCAs do not require internal storage apart from status registers, since they use the L1 to hold data. L1 misses are served by the memory hierarchy on behalf of the coupled CPU. Reorders in the cache are hidden so that the accelerator can exploit multi-ported caches transparently [121].

A strength of TCAs is the nil run-time overhead of their invocation. In a similar manner to coprocessors, TCAs require an expansion of the ISA to include special instructions to manage their operation. This ISA expansion usually percolates to software via the compiler or through low-level libraries.

From a hardware viewpoint, however, TCAs can pose integration challenges. First, they further complicate the design of the CPU. Second, they can pose timing closure challenges, since it is common to require the TCA logic to meet the same clock-frequency constraints that are set for the CPU. Third, they have limited portability

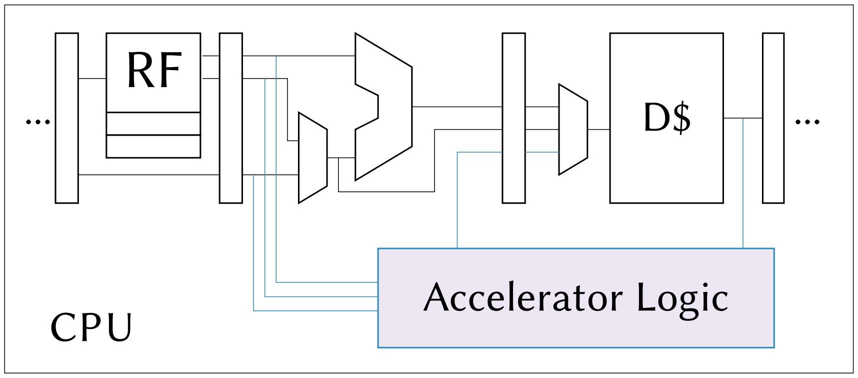


Figure 5.1: Tightly-coupled accelerator (TCA) model. The accelerator shares key resources (register file, MMU and L1) with the CPU.

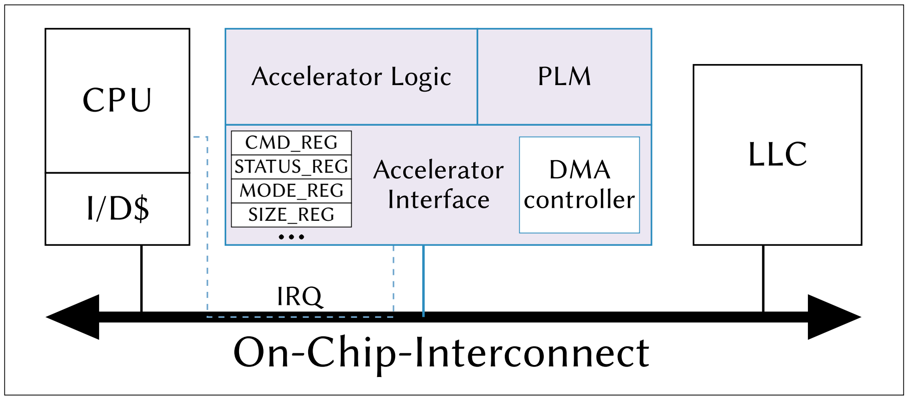


Figure 5.2: Loosely-coupled accelerator (LCA) model. The integrated DMA controller transfers data between the accelerator's PLM and either the LLC or DRAM.

across different system designs, since it is often necessary to adapt the accelerators' interfaces to CPU-dependent structures.

Loosely-coupled accelerators (LCAs). We consider two options, depending on whether the accelerators are capable of direct memory access to either the last-level cache (**LLC-DMA**) or to DRAM (**DRAM-DMA**). Loosely-coupled accelerators (LCAs) are located outside CPU cores and interact with them through the on-chip interconnect, as shown in Figure 5.2. Being out-of-core affords LCAs a greater area budget than TCAs since they cannot degrade the processor pipeline's performance or the L1 access time. This allows for coarse-grained accelerator logic blocks with

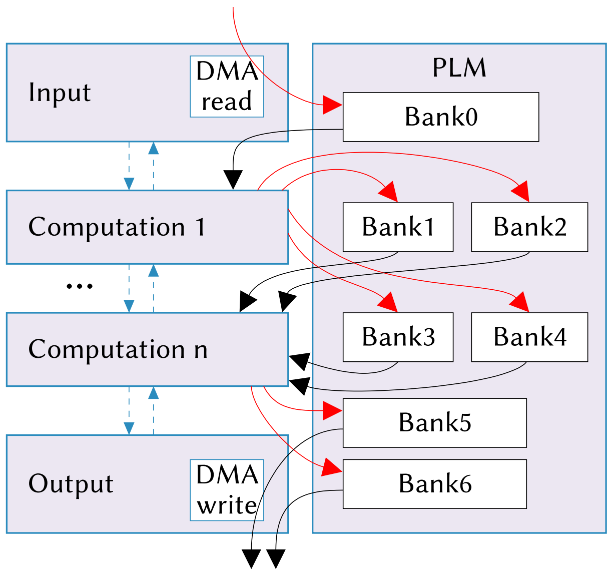


Figure 5.3: Typical LCA structure. Aggressive SRAM banking enables multi-ported memories for computation blocks.

complex data paths that implement and accelerate a complete application kernel, for instance a fast Fourier transform (FFT) or a full image encoding algorithm.

The relaxed area constraint due to being out-of-core allows LCAs to implement private local memories (PLMs), which store the input data to be processed, temporary results, and the output data to be written back to memory. Crucially, these PLMs can be tailored to the accelerator's specific needs. Given that memories are usually implemented with static RAM (SRAM) and each SRAM bank has at most two ports, LCA PLMs typically combine several independent SRAM banks to form multi-ported memories of sizes tailored to the LCA's needs. This is a key advantage of LCAs versus TCAs: LCAs can unleash the parallelism inherent in the kernels they accelerate by instantiating these tailored many-ported memories, whereas the granularity of TCA data is at the cache line size and their parallelism is severely constrained by the unavoidably low number of ports of L1 caches.

Figure 5.3 shows a typical breakdown of an LCA into four main components: (1) a DMA input block; (2) one or more computation blocks that handle data tokens of different sizes; (3) a DMA output block; and (4) a PLM. These blocks interact by means of control signals and shared memory banks.

From a system-level perspective, the decoupling of LCA from the CPU results in greater flexibility than that of the TCA model. For instance, when the accelerator is running the CPU is free to run other tasks or be turned off to save energy. The TCA model, however, requires less effort from software, since LCAs require software to make sure that data tokens are available and consistent. This is usually accomplished with a device driver running in kernel space, since low-level control of memory is required. Similarly to the case of a TCA, the user application is responsible for preparing the data in memory. Unlike TCAs, however, the application must issue a system call (e.g. `read()`, `write()`, `ioctl()`) to invoke the corresponding device driver, which passes the physical addresses of this memory to the LCA's DMA controller. Then, once the command to start is issued, the driver puts the calling thread to sleep until an interrupt from the accelerator arrives.

From a system integrator viewpoint, LCAs offer better design-reuse opportunities because their design is mostly independent from the design of the CPU and the rest of the system. To couple the LCA with these it is sufficient to have a thin hardware wrapper that interfaces its configuration registers and DMA controller with the on-chip interconnect. Furthermore, provided that the system allows for multiple clock domains, an LCA doesn't necessarily need to run at the same frequency of the CPU, thus simplifying the porting of the accelerator across different technology processes.

5.3 Target Applications

To carry out our analysis we design a custom platform composed of CPUs, accelerators, user applications and device drivers. We choose candidates for acceleration from MachSuite [201] and the PERFECT Benchmark Suite [22]. Out of them we select those applications that present interesting memory-access patterns and are suitable to architectural optimizations, e.g. ping-pong data buffering (Figure 5.4), circular buffering or data caching. We adopt high-level synthesis (HLS) [170] to automatically synthe-

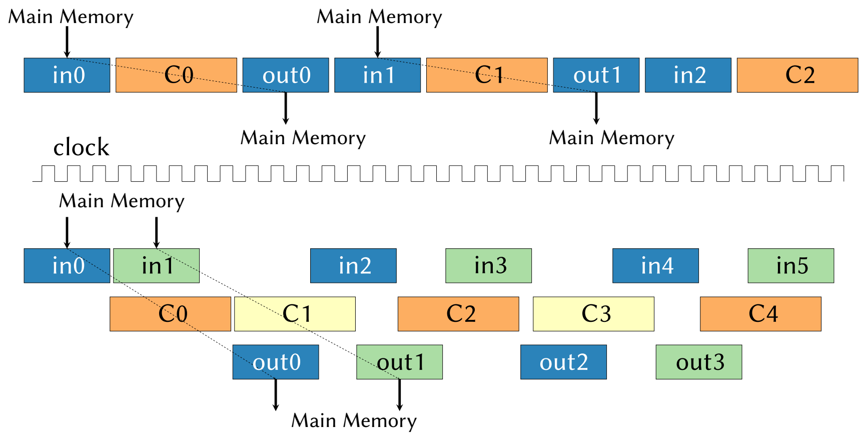


Figure 5.4: Ping-pong data buffering (below) improves throughput over single buffering (above) by overlapping in time computation and communication.

	Footprint		Accelerator	PLM	
Application	N	Size *(bytes)*	Area *(um^2)*	Area *(um^2)*	Size *(bytes)*
AES	5 - 1000	80 - 16K	192,792	–[*]	192
FFT	8 - 12	2K - 32K	337,770	299,605 (88%)	40K
FFT-2D	4 - 10	2K - 8M	146,199	98,273 (67%)	16K
Sort	8 - 128	32K - 524K	302,672	210,636 (69%)	25K
Debayer	16 - 1024	512 - 2M	207,206	196,522 (94%)	32K
Lucas Kanade	32 - 512	8K - 2M	588,001	538,775 (91%)	41K
Change Detection	32 - 512	71K - 18M	189,826	134,954 (71%)	16K

[*] Small PLMs are mapped on registers.

Table 5.1: Accelerators' footprint, area and aggregate PLM's characteristics.

size the C implementations from the suites into custom RTL accelerators, which we then integrate with the CPUs using a virtual (simulated) bus. Section 5.4 has more details on the full-system simulation of our platform. In the remainder of this section we describe the main challenges in designing the seven accelerators.

Footprint and PLM. Figure 5.5 depicts the applications' input data tokens, highlighting the minimum addressable element that each algorithm requires. The value N represents which dimensions are used for parameterizing the corresponding input size. For example, in AES, N is the number of 128-bit input blocks.

Table 5.1 reports for all applications the considered N ranges and their corre-

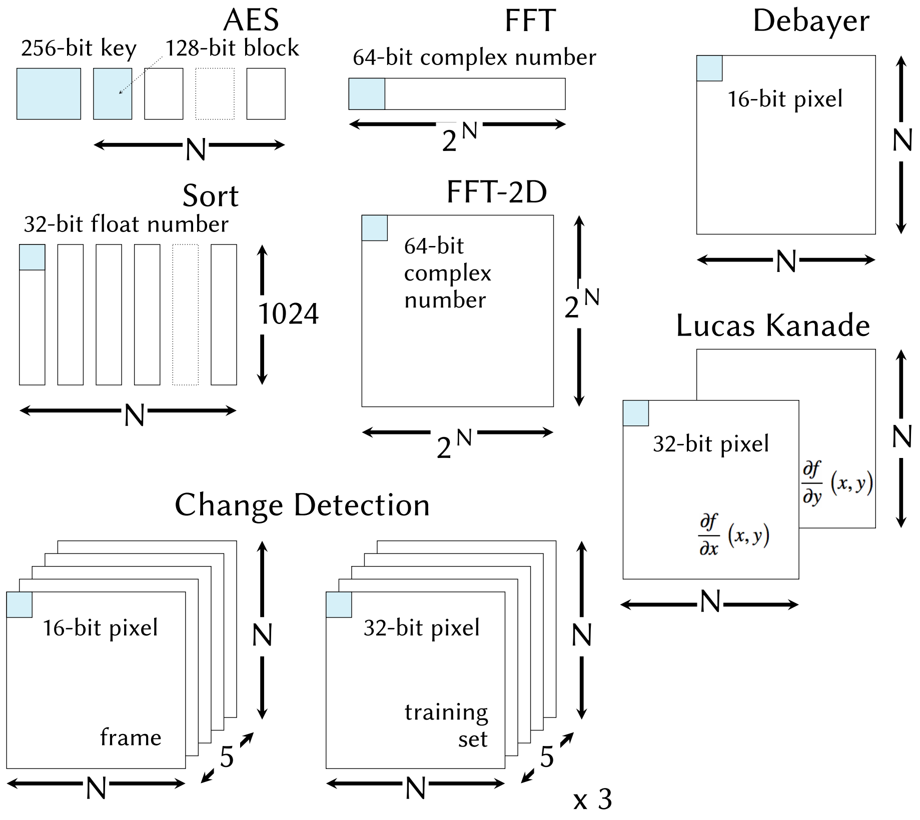


Figure 5.5: Application memory footprints.

sponding *memory footprint* size. This is the measure of the total number of bytes an application uniquely addresses as input data. These sizes are chosen according to the size of the adopted LLC (4MB); we thus cover applications whose footprint is significantly smaller (AES, FFT and Sort), approximately the same size (Debayer and Lucas Kanade) and significantly larger (FFT-2D and Change Detection) than the LLC.

We perform logic synthesis and preliminary place and route using a 32*nm* SOI CMOS technology. In Table 5.1 we report the accelerators' total area, as well as the area and size aggregates of the PLMs they integrate. Note that for most accelerators the aggregate PLM size is of the same order of magnitude as the CPU's L1 data cache (64 KB).

Architectural choices. We adopt HLS in order to efficiently evaluate multiple implementation alternatives through design space exploration (DSE). We observe that the design of the PLM largely determines the resulting design space: on the system's side we have to define the communication between the accelerator, its local-memory subsystem and the off-chip memory; on the accelerator side, we have to make several micro-architectural choices that are directly correlated with the PLM's architecture, e.g. number of ports and banks. Let us consider some examples.

As shown in Section 5.2, the accelerator model consists of hardware modules that share a local memory subsystem. The model supports the overlapping of I/O with computation, which provides the designer with a significant degree of optimization.

> *Example 1.* In order to achieve high throughput, we adopt ping-pong data buffering to implement the data transfer among the off-chip main memory and the accelerator PLM. Image-processing applications (e.g., Debayer, Change Detection) that are of streaming nature significantly benefit from this solution. □

HLS offers a rich set of knobs for the RTL design optimization, e.g. for manipulating loops, pipelining portion of a design, inserting states, implementing array as memories or registers etc.

> *Example 2.* Figure 5.6 reports a portion of synthesizable C code of the Debayer application. This estimates via interpolation non-sampled values of red, green, and blue of a given image. In particular, the code applies an interpolation mask to estimate values of green for an image stored in the two-dimensional array `bayer`. A first micro-architectural choice is to allocate the arrays `bayer` and `debayer` as memories rather than flattening them as registers: the use of registers produces very fast but excessively large hardware. A second micro-architectural choice targets the loops of the application. Combinational loops cannot be implemented in hardware: they must be either broken or unrolled, and choosing between these options is an area vs. performance trade-off.
>
> An important constraint is the scheduling of memory accesses. For instance, the loop `COL_LOOP` contains read and write operations on the memory-allocated arrays `bayer` and `debayer`. Unrolling such loop generates an implementation with multiple read and write memory operations per clock cycle, therefore improving performance. This however reduces the ability to schedule the design,

```
#define PAD 2
#define NUM_ROWS 1024
#define NUM_COLS 1024

uint16_t bayer[NUM_ROW][NUM_COLS];           // input img
uint16_t debayer[NUM_ROW-PAD][NUM_COLS-PAD]; // output img

// interpolate green value for pixels on even row and column
ROW_LOOP: for (row = PAD; row < NUM_ROWS-PAD; row += 2)
{
  COL_LOOP: for (col = PAD; col < NUM_COLS-PAD; col += 2)
  {
    u16 pos =
         2*bayer[row-1][col] + 2*bayer[row][col-1] +
         4*bayer[row][col]   +
         2*bayer[row][col+1] + 2*bayer[row+1][col];
    u16 neg =
         bayer[row][col+2] + bayer[row-2][col] +
         bayer[row][col-2] + bayer[row+2][col];

    debayer[row-PAD][col-PAD].green = ((pos - neg) >> 3);
  }
}
```

Figure 5.6: DSE-enabled implementation of the Debayer kernel.

given that the number of available memory ports is heavily vendor and technology dependent. □

In summary, we observe that accelerator design effort is usually concentrated on (1) the accelerator memory subsystem, which is responsible for most accelerator area, and (2) how to efficiently bring data in and out of accelerators.

5.4 Experimental Methodology

Simulated System. To conduct our evaluation we developed Cargo, a full-system simulator derived from the QEMU-based Qsim [132]. We model a one-issue in-order core with a 3-stage (Fetch + Decode, Execute, Memory + Writeback) pipeline and a 2-level cache hierarchy. The last-level cache is connected to a memory controller modeled by DRAMSim2 [206]. Table 5.2 summarizes the system's parameters.

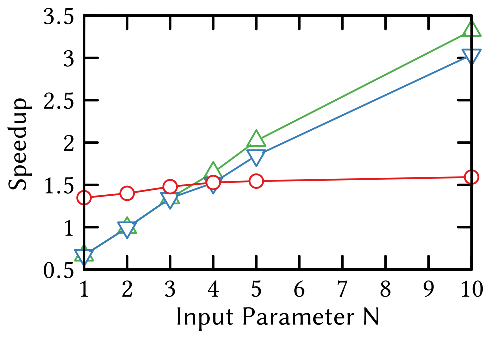
Cores	2 cores, i386 ISA, 3-stage pipeline, 2 GHz
Exec. Latency	1 cycle except IMUL=4, IDIV=15, FPADD=5, FPMUL=5, FPDIV=25 [98]
L1 caches	32KB I, 64KB D, 4 ways, 2+2 I/O ports, 1-cycle latency, LRU
L2 cache	4MB, 16 ways, 16 banks, 4 MSHRs, 1+1 I/O ports, 11-cycle latency, LRU
DRAM	1 Controller, 3.5GB, Micron DDR3 400MHz
OS	Linux v2.6.34

Table 5.2: System configuration for experimental results

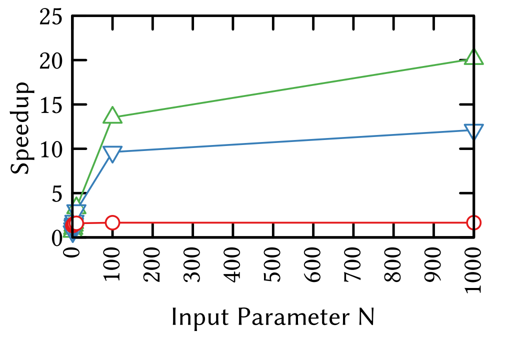
Energy consumption is modeled by combining our performance numbers with power models. We use McPAT1.0 [155] for modeling core/directory/L1 power, CACTI 6.5 [185] to obtain power/latency numbers for sequential-access low-leakage L2 cache banks, and approximate DRAM power by assigning 2.78W to background power and 51nJ per access for a single DIMM as done by Sampson and Wenisch [207].

Simulated Accelerators. We simulate TCAs by substituting the applications' core kernel code with special code blocks that contain latencies back-annotated from the RTL implementation of the accelerators. Our simulator recognizes these special code blocks and freezes the CPU pipeline for the latency specified in each block. Additional latency is appropriately added to the freeze if within a block the modeled accelerator performs memory accesses that miss in the L1 data cache.

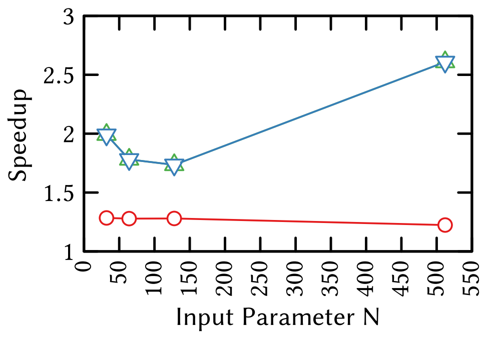
LCAs are simulated by attaching back-annotated SystemC accelerator code to our event-driven simulation engine. We synchronize the system's event queue with SystemC's event queue every 100 cycles—this results in a minimal event skew and provides significant gains in simulation time. LCA interrupt latency is set to 2000 CPU cycles, which is commensurate with current Inter-Processor Interrupt (IPI) latencies. DRAM-DMA LCAs are fed from noncacheable memory buffers.



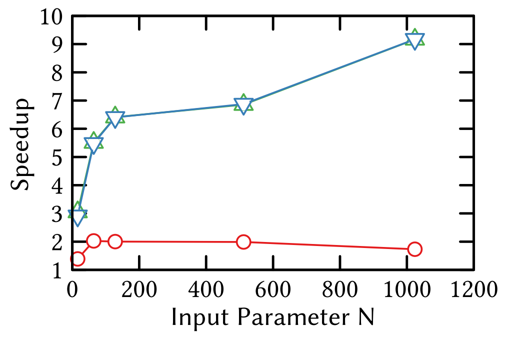
(a) AES (tiny inputs)



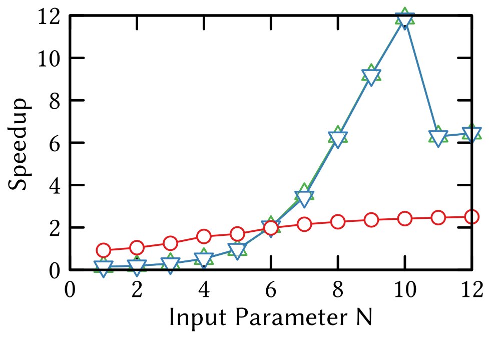
(b) AES



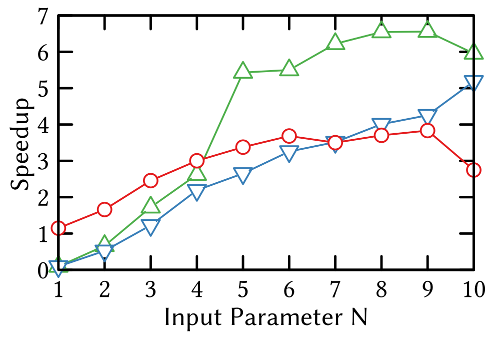
(c) Change Detection



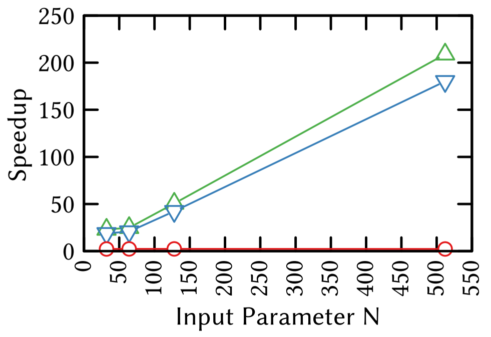
(d) Debayer



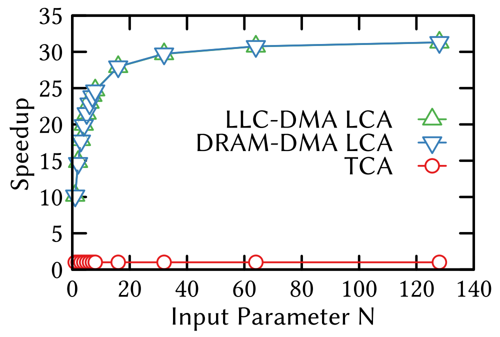
(e) FFT



(f) FFT-2D



(g) Lucas-Kanade



(h) Sort

Figure 5.7: Speedup over software for all accelerators. Input sizes are parameterized as described in Table 5.1.

5.5 Experimental Results

Performance. Figure 5.7 reports the speedup for all accelerators over a software implementation running on a simulated CPU core. The reported speedup is on average much greater for LCAs than for TCAs. TCAs only outperform LCAs when the input size is small (i.e. fits in a CPU cache line), which is due to the fixed cost in setting up LCA accelerators—i.e. system call latency, I/O access latency and interrupt latency. TCAs outperform LCAs for small inputs of AES and FFT. Note however that in these cases the speedup over software is not significant (less than 2x), the reason being that given the small inputs there is not much computation to do.

Moving to larger input sizes greatly improves performance for LCAs. In these scenarios the fixed cost of operating LCAs is amortized by the significant computation throughput they can sustain thanks to their tailored PLMs. However, two causes can limit LCA performance.

First, the limited size of the PLM may force the algorithm to abandon ping-pong data buffering when dependent data chunks outsize the available PLM. An example of this is the FFT: note the drop in performance at $N = 11$, where the accelerator is forced to alternate between communication and computation. Second, the high throughput that the accelerator can sustain may not be sustainable by DRAM. This leads to a flattening of the speedup with increases in input size. Clear examples of this effect are Sort (flattening above 30X) and FFT-2D (5X).

Whether DRAM becomes a bottleneck is ultimately a function of the kernel to accelerate. If the kernel has a high ratio of computation over communication, that is, relatively short data transfers lead to significant calculation, then an accelerator with a sufficiently large PLM can sustain high throughput. Lucas-Kanade is an example of this: the speedup for the largest input set reaches 200X over software. Increasing speedups with input size can also be seen for AES, Change Detection and Debayer, but these are less pronounced due to the lower computation/communication ratio.

Energy Efficiency. Figure 5.8 aggregates speedup and energy reduction results from the experiments shown in Figure 5.7 (which total 55 distinct experiments for each coupling model), sorting them in monotonically increasing order. The perfor-

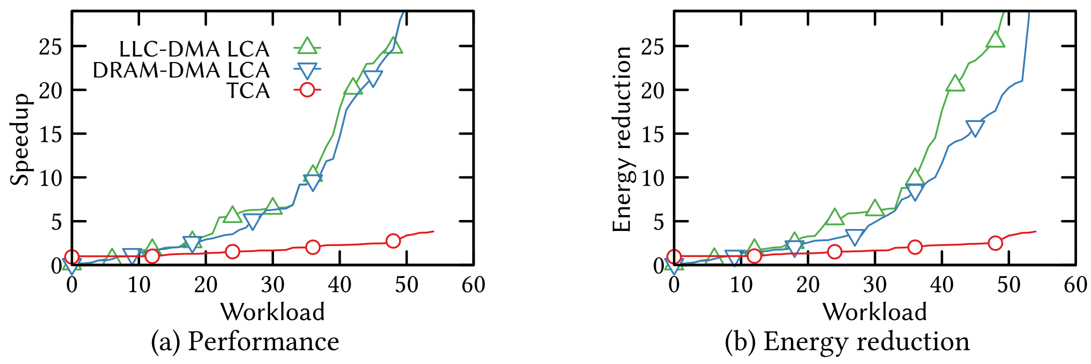


Figure 5.8: Improvements over software for all workloads.

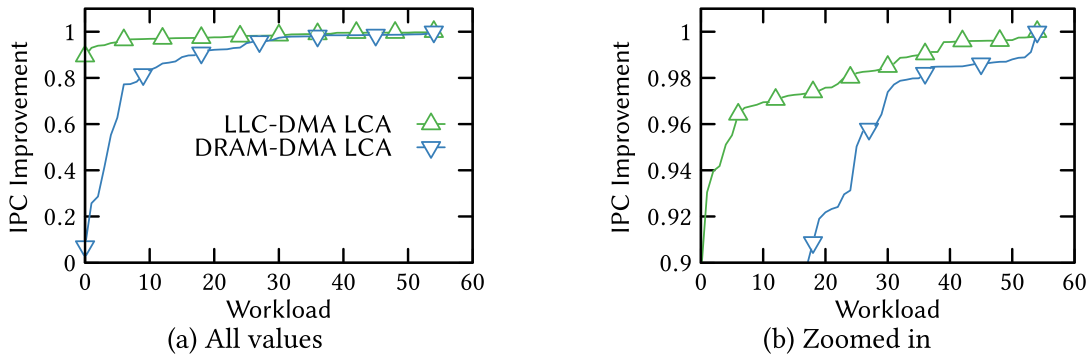


Figure 5.9: omnetpp IPC increase over running in isolation.

mance gap between the two LCAs and the TCA becomes here even more evident than in Figure 5.7. Further, the gap in energy reduction between LLC-DMA and DRAM-DMA LCAs widens with respect to their gap in performance. The reason is that LLC-DMA accelerators perform less accesses to off-chip DRAM, which incur in a significant energy penalty. It is thus clear that on average, performance and energy improvements are greater with LLC-DMA LCAs.

LLC-DMA vs DRAM-DMA LCAs. A potential source of concern with regards to LLC-DMA loosely-coupled accelerators is cache pollution. To measure this effect we simulate a 2-core system running omnetpp, a SPEC06 benchmark that is sensitive to LLC size around 4 MB [123], together with a program that runs an LCA-accelerated application in an infinite loop, where each workload and input size is a unique pair chosen from the aforementioned set of 55 experiments. We fast-forward simulation for 100M cycles of omnetpp, and then record omnetpp's IPC over 256M cycles.

Figure 5.9 shows the resulting IPC improvements for omnetpp. The improvement is always below 1, i.e. the interference is always detrimental to performance. The left plot (all values) shows a dramatic performance reduction for omnetpp when coexisting with certain DRAM-DMA LCAs. This is explained by the DRAM bottleneck we alluded to earlier: these accelerators saturate DRAM bandwidth (sort, FFT-2D) thereby adding high latency to omnetpp's relatively rare LLC misses. LLC-DMA LCAs do not suffer from such a severe performance degradation due to the mediation of the LLC; off-chip accesses to DRAM from the accelerator are thus kept to a minimum, which leaves enough DRAM bandwidth to ensure moderate LLC miss latencies.

Note that the interference seen for the first ten DRAM-DMA workloads is very noticeable, which is partly due to the fact that we invoke the accelerator in an infinite loop. More realistic workloads might not necessarily run accelerators in a loop as tight as ours, which would moderate the performance degradation. However, many-accelerator systems are likely to encounter scenarios in which several accelerators execute at the same time, which would bring the bandwidth demands close to those of our experiments. Our results show that given an adequately sized LLC, LLC-DMA LCAs are better equipped to mitigate this effect.

5.6 Related Work

Most work on accelerator research focuses on the performance and energy efficiency improvements that accelerators can provide [233, 50, 242], rarely considering system-level implications. Interesting examples of the latter are by Kelm and Lumetta [129] and Cong et al [63], who focus on software support for loosely-coupled accelerators. Our study is complementary to their work; we study a wider range of accelerator models and a wider set of applications, while also evaluating the memory-hierarchy interference that results from having high-throughput accelerators share the die with memory-intensive software.

Vo et al make a case for OS-friendly accelerators [234]. Our approach differs from theirs in that the applications we consider show little benefit from tightly-coupled acceleration since many-ported tailored memories are necessary to sustain high-throughput. Further, our results show that the software model for device management prevalent in SoCs is directly applicable to these high-throughput accelerators. Stuecheli et al [225] attach accelerators to the PCIe bus using a cache that is coherent with other CPUs in the system, thereby removing the need for device drivers. This is a promising approach for workloads that (1) must be accelerated off-chip, e.g. due to prohibitive area requirements or need for reconfigurability (e.g. on FPGA), and (2) require frequent communication with general-purpose cores.

5.7 Summary

In this chapter we have considered system integration and programmability issues inherent in different accelerator models. Through full-system simulation we performed a quantitative and qualitative comparison of three such models: tight coupling behind a CPU, loose (i.e. out-of-core) coupling with DMA to the LLC, and loose coupling with DMA to DRAM. Our experiments on these accelerators induce a set of observations that can help future designers and increase understanding of existing designs.

Observations. From our quantitative study we observe the key role of private memory blocks in high-performance acceleration. Loosely-coupled accelerators can leverage these blocks by tailoring SRAM banks to the needs of their computation blocks; the resulting multi-ported memories enable the exploitation of parallelism inherent in kernels, which is where the potential for performance and energy efficiency improvements lies. This potential might not be realized if the application requires excessive DRAM bandwidth or is not amenable to sustained computational bursts that fit in the accelerator's PLM. Equipping accelerators with direct memory access to the LLC can mitigate this in some cases, which also reduces the risk of DRAM bandwidth saturation. With regards to software, abstracting these high-throughput loosely-coupled accelerators using device drivers similar to those for SoC on-chip devices shows to be a low-complexity and efficient approach. Recent work by Giri, Mantovani and Carloni [102] corroborates this last point through an evaluation of accelerators and CPUs implemented on FPGA.

Limitations. There are potentially as many accelerators as applications in existence. Therefore we cannot claim that our observations apply to every workload imaginable. We instead restrict our scope to high-throughput applications that (1) have clear memory access patterns and have input sizes large enough to make vector processing impractical and (2) are irregular enough to not map well into GPUs; an exception to this is the FFT, which we chose for its popularity.

Chapter 6

ROCA: Reducing the Opportunity Cost of Accelerator Integration

In this chapter[1] we present a novel technique to expand the last-level cache of a heterogeneous system by reusing memory from otherwise unused on-chip accelerators. The evaluation of this technique underscores the importance of scalable emulation; by enhancing Cargo (introduced in Section 5.4) to support multicore-on-multicore simulation, we are able to simulate a heterogeneous system that runs large parallel workloads (executing billions of instructions) at an acceptable speed.

6.1 Introduction

Owing to their specific purpose, non-programmable, high-throughput accelerators allow designers to tailor the microarchitecture to a specific workload, thereby delivering near-optimal performance and energy efficiency [35]. Unfortunately, these come at the expense of generality; a given accelerator can only speed up a specific instance of an algorithm, with little or no flexibility to incorporate potential improvements to it. As a result, the opportunity cost of integrating accelerators in general-purpose architectures is usually prohibitive, since few accelerators are likely to apply

[1]This chapter incorporates and extends work previously published in *Computer Architecture Letters* [70] and in the proceedings of the 2016 ICS conference [68].

to a workload that is unknown at design time. Thus, investing area in accelerators for these architectures frequently implies forgoing more generally applicable and therefore productive alternatives, such as larger caches and/or cores, greater core counts, or not making the investment at all.

Prior work improves average on-chip memory utilization as a way to reduce accelerator opportunity cost by building on two observations: accelerators are mostly made of private local memories (PLMs) and have low average utilization. Thus, the resulting solutions reduce the overall amount of integrated PLMs by providing storage that accelerators can allocate memory from. This storage is implemented either as an accelerator-only memory pool [162] or by allocating blocks for accelerators from the last-level cache (LLC) [65, 94]. Although these techniques reduce the total on-chip SRAM investment, they are only applicable to low-bandwidth PLMs, which are scarce in high-throughput accelerators as we discuss in Section 6.2.

We leverage an additional observation to further reduce the opportunity cost of accelerator integration. We observe that accelerator PLMs disseminated across the chip provide a de facto non-uniform cache architecture (NUCA), which is the optimal organization for large, multi-megabyte caches [135]. Thus, instead of providing storage external to accelerators, our goal is to *expose* accelerator PLMs to the LLC [70], thereby extracting utility from *all* PLMs and not just from low-bandwidth ones. Moreover, this simplifies the design effort over prior work, since the designer does not need to artificially break PLMs into high and low bandwidth groups.

Three difficulties make exploiting accelerator PLMs by the LLC challenging. First, exposing PLMs to the cache substrate requires logic to abstract the PLMs' diversity in size, bitwidth and number of ports. Second, the cache substrate has to dynamically handle the intermittent availability of some of its capacity. Last, the delay due to the eviction of dirty blocks from PLMs reclaimed by an accelerator can potentially degrade the accelerator's performance.

We propose Roca, a technique to exploit accelerator PLMs to *reduce the opportunity cost of integrating accelerators*. Roca transparently exposes accelerator PLMs to the cache substrate, thereby extending LLC capacity while accelerators are not

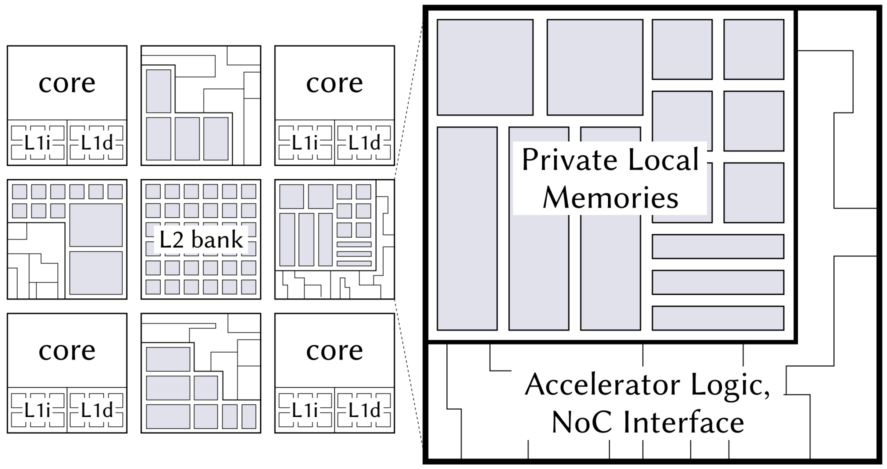


Figure 6.1: 4-core chip with 2-level cache and four accelerators. Most accelerator area is devoted to private local memories (PLMs) that, when inactive, are used by ROCA to expand the LLC. The memory blocks of the resulting LLC are shaded in gray.

in use. Figure 6.1 illustrates the goal of our technique. A four-core chip shares a network-on-chip (NoC) interconnect with four accelerators, whose PLMs' diversity (in number of ports, size, and bitwidth) is represented by rectangles of varying dimensions. With Roca, the LLC is formed by combining regular non-uniform cache architecture (NUCA) banks with PLMs from otherwise unused accelerators.

Roca uses a combination of old and new techniques to achieve low complexity, high performance and modest area overhead. First, it adds a minimum amount of logic around PLMs to expose them through an additional port to the cache substrate. Second, it relies on the decoupling between cache tags and the corresponding data [55], keeping tag storage external to accelerators. Third, it leverages selective cache ways [10] to dynamically adapt to the intermittent availability of PLMs, flushing dirty blocks to DRAM as PLMs are reclaimed by accelerators.

In this chapter we make the following contributions:

- We present Roca, a technique that leverages accelerator PLMs to dynamically expand the last-level cache (Section 6.3), and show its modest area overhead, which is almost entirely due to additional tag storage (Section 6.4).

- We perform full-system simulation of multiprogrammed workloads running on a multi-core architecture whose LLC is mostly implemented from Roca accelerators, evaluating Roca's performance and energy efficiency overhead, which we show to be low, and studying Roca's sensitivity to different accelerator reclamation frequencies for several spatial configurations. (Section 6.5).

In summary, Roca shows that extending the last-level cache by exploiting accelerators PLMs when otherwise unused is an effective way of reducing accelerators' opportunity cost, since it decouples the utility of accelerators from the workload under consideration.

6.2 Background

Recent work on non-programmable accelerators hinges on two main observations:

Private memories are key to accelerator performance and energy efficiency. Specialized hardware can potentially exploit all parallelism inherent in a given computational kernel. However, a necessary condition to realize this parallelism is the ability to fetch data, possibly in highly irregular patterns, at the same rate as they are processed by a specialized datapath. Attaching accelerators to existing CPU caches in order to save area and therefore energy might seem worth pursuing, but unfortunately cache structures cannot fulfill most accelerators' requirements. First, high-throughput accelerators require memories with a far greater number of ports than what caches can efficiently implement [50, 108]. Second, a fixed cache block size cannot serve well the needs of reads and writes of various widths that occur even within just a single accelerator. And third, the high associativity needed by caches to provide fast lookups imposes significant energy and area overheads, which goes against the efficiency goal of acceleration. Accelerators are thus best served by private local memories, which as we discussed in Chapter 5 are memory blocks only exposed to accelerator hardware and tailored in their number of ports, banks and widths to precisely match the needs of each computational block within an accelerator.

Accelerators are mostly memory. Given the importance of low-latency, high-bandwidth memory accesses for accelerators' performance, private memory blocks take a substantial portion of accelerators' area. For instance, a survey of eleven publicly available accelerators reveals that "an average of 69% of accelerator area is consumed by memory" [162], and recent high-performance accelerators show significant private memory investments as well [50, 145].

A corollary to the second observation is that average accelerator memory utilization is low on many-accelerator systems, since not all accelerators are likely to run at the same time. Thus, prior work has reduced accelerator opportunity cost by moving private memories out of the accelerator: proposals range from an on-chip memory pool that accelerators can allocate from [162], to providing a substrate that can store cache blocks as well as accelerator data [94, 65].

Roca has two advantages over these proposals. First, it applies to *all* accelerator memories, regardless of their bandwidth or access pattern (fixed or data-dependent). Second, it requires modifications on the accelerators that are simpler to implement: designers do not have to worry about predicting access patterns or what memory blocks are (or could be engineered to be) of higher/lower bandwidth. Instead, with Roca designers just focus on optimizing their design without any additional requirements, and once the design is done, a small amount of logic around memory blocks is all that is needed to make an accelerator Roca-compliant.

Roca also adds logic to the cache substrate to make it view accelerator PLMs as de facto NUCA storage. The complexity of the logic needed, apart from additional tag storage, is low due to two ideas. First, we exploit the observation by Chishti et al. [55] that, since tag and data lookups happen sequentially in large caches, their placement can be decoupled. Second, we leverage selective cache ways by Albonesi [10] as the mechanism to accommodate accelerator PLMs of diverse sizes, and to rapidly and efficiently adapt to their intermittent activity and, therefore, availability.

6.2.1 Accelerator Example: Sort

We now illustrate the typical structure of a high-throughput accelerator through a particular example, which we will also use in the next section to describe the modifications that Roca imposes on accelerators.

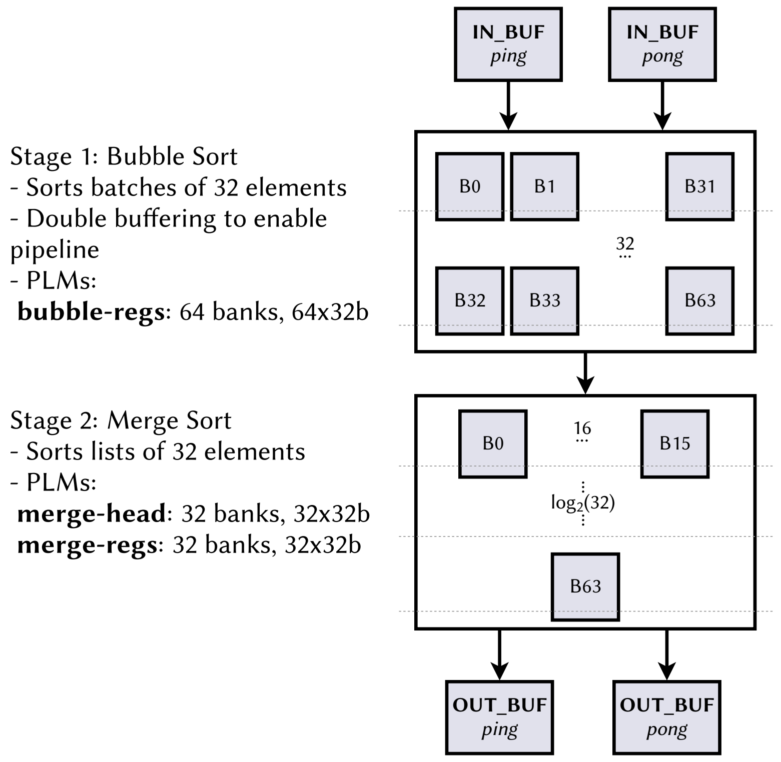

IN_BUF
ping
IN_BUF
pong
Stage 1: Bubble Sort
- Sorts batches of 32 elements
- Double buffering to enable pipeline
- PLMs:
bubble-regs: 64 banks, 64x32b
B0
B1
B31
32
B32
B33
B63
Stage 2: Merge Sort
- Sorts lists of 32 elements
- PLMs:
merge-head: 32 banks, 32x32b
merge-regs: 32 banks, 32x32b
B0
16
B15
log2(32)
B63
OUT_BUF
ping
OUT_BUF
pong

Figure 6.2: Structure of the Sort accelerator. Ping-pong buffering enables simultaneous processing of vectors. The dashed lines denote different pipeline stages. Banks of the five PLMs are shaded in gray.

Our *Sort* accelerator is designed to meet the PERFECT benchmark suite's requirements [22] by sorting batches of floating point vectors of up to 1024 elements each. The accelerator, whose structure is depicted in Figure 6.2, has five PLMs and computes in two stages. The first stage sorts groups of 32 elements through a parallel implementation of bubble sort: the innermost loop of the algorithm can complete in one clock cycle due to aggressive SRAM banking to feed a specialized datapath with 32 comparators. Moreover, PLMs in this stage are doubled to support the processing of groups of 32 elements in a pipeline, thus allowing reads and swap-induced writes to occur in the same clock cycle. The second stage sorts the resulting 32-element lists (with a maximum of 32 lists for a total of 1024 elements) using merge sort. PLMs are again heavily banked to maximize performance and enable pipelining.

The accelerator was optimized for performance and energy efficiency: compared

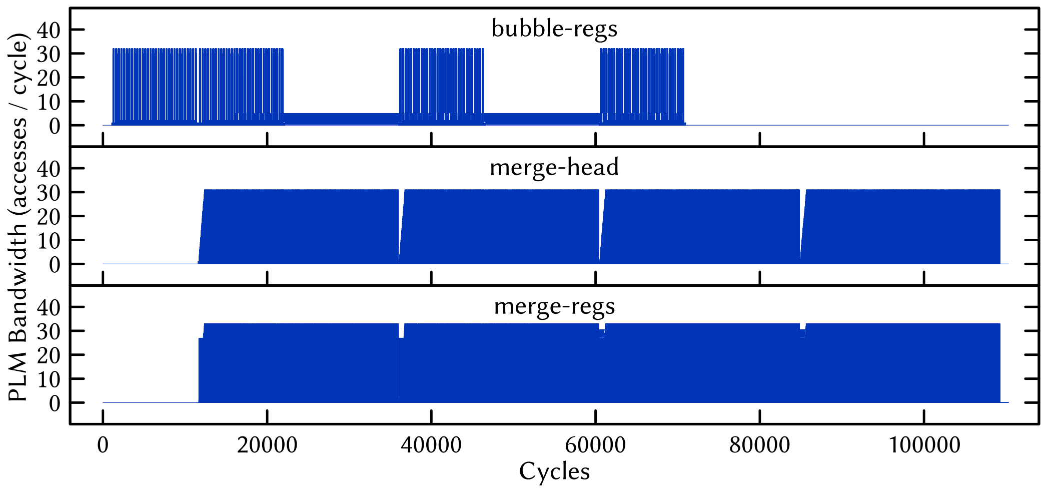


Figure 6.3: PLM Bandwidth of the Sort accelerator when sorting 4 vectors of 1024 elements. High PLM bandwidth is enabled by heavy SRAM banking.

to a software implementation running on an Intel Haswell processor clocked at 2.3GHz, a 1GHz silicon implementation[2] of the accelerator is up to 3.5X faster, while requiring only 0.5% of the energy; the speedup of the accelerator over a software implementation for an OpenRISC CPU, with both accelerator and CPU synthesized in an FPGA at 100MHz, is of up to 300X while consuming 3% of the energy. These significant gains are enabled by the use of high-bandwidth PLMs. Their effect is illustrated in Figure 6.3, which plots the PLM bandwidth over time for the two accelerator stages when sorting 4 vectors of 1024 elements. The plots show how aggressive SRAM banking enables large amounts of concurrent PLM accesses (up to 32 accesses per PLM per cycle) which are key to achieving high performance. Additional banking can also help pipelining. For instance, the *bubble-regs* PLM has twice as many banks as it would otherwise need, which provides the necessary intermediate buffering for processing different vectors in a pipeline.

Applying prior approaches [65, 94, 162] to this accelerator could only be done for a subset of the PLMs and would result in a substantial impact on performance or in-

[2]The accelerator logic is simple and therefore can be clocked at high frequencies. However, the achievable clock frequency of our silicon implementation is limited by the memory generators available to us.

PLM	Size (KB)	Banks	Peak Bandwidth
merge-head	4	32	32
merge-regs	4	32	32
IN_BUF	8	1	1
OUT_BUF	8	1	1
bubble-regs	16	64	32

Table 6.1: Characteristics of the PLMs in the Sort accelerator. Bandwidth is measured in number of accesses per cycle.

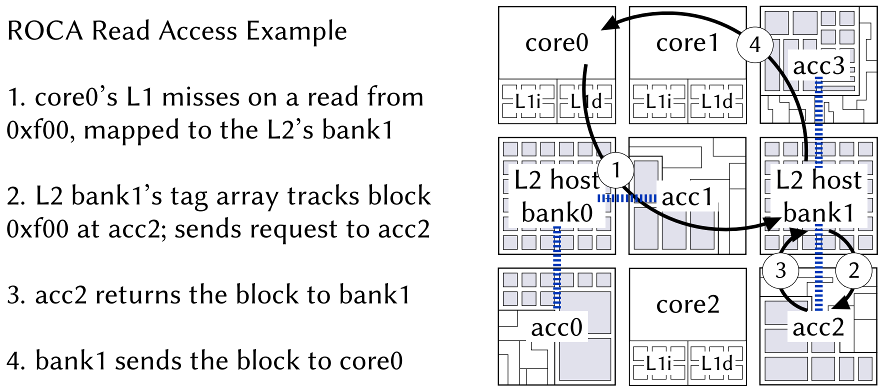
crease in complexity. Moving memories out of the accelerator would only be possible to do for input and output buffers, since they are the only ones whose access patterns are fully predictable, i.e. do not depend on input values. As shown in Table 6.1 these memories (*IN_BUF* and *OUT_BUF*) amount to only 40% of the PLM total, i.e. 16 out of 40 KB. Moreover, implementing this move would be complex without significantly impacting area or performance: buffers would have to be added to hide the pipeline-induced latency of reading/writing data to a remote bank via the interconnect.

In the next section we show how Roca adds simple logic to accelerators to transparently expose all of their PLMs to the cache substrate.

6.3 ROCA: Exposing Accelerator Memory to the NUCA Substrate

Our implementation of Roca combines the following hardware elements, which we describe in detail in this section:

- An enlarged tag array in the last-level cache to track blocks stored in accelerator PLMs.
- Logic and gating hardware in the tag array to enable/disable ways as accelerators reclaim their memory, as in selective cache ways [10].
- Registers (one per logical bank) that accelerators or privileged software can


ROCA Read Access Example

1. core0's L1 misses on a read from 0xf00, mapped to the L2's bank1

2. L2 bank1's tag array tracks block 0xf00 at acc2; sends request to acc2

3. acc2 returns the block to bank1

4. bank1 sends the block to core0

Figure 6.4: High-level operation example. Three cores share a LLC with blocks interleaved across two logical banks, each composed of a ROCA host bank and associated accelerators (marked with a dashed line). Requests to a logical bank are always routed to the host bank (1-2), as well as responses from ROCA accelerators with the requested data (3-4).

access to enable/disable the participation of accelerators in Roca based on the accelerators' activity rate.

- Logic on accelerators to coalesce SRAMs of different widths, sizes, and ports to expose them as a single PLM to the cache substrate via the interconnect.

- Logic on the tag array to support the flushing of dirty cache blocks in accelerator PLMs.

6.3.1 High-Level Operation

We explain the high-level operation of Roca in Figure 6.4. For the shown 3-core chip the physical address space is split in two *logical banks*, which are regular LLC banks that are extended with Roca-enabled accelerators. In the example, core 0 misses in its L1 on a read access to the block at address 0xf00, which is assigned to be cached at the L2's *host bank* 1. Core 0 then sends its block request to host bank 1. The bank checks its enlarged tag array (which also tracks blocks in accelerators 2 and 3) and forwards the request to accelerator 2. The accelerator sends back to host bank 1 the contents of the block, and the host bank then forwards it to core 0.

In this operation example, accelerator 2 could have directly sent the block to core 0, therefore lowering access latency. However, this modification would be ill-advised. For example, consider the scenario in which immediately after the access request for 0xf00, host bank 1 received a command to disable accelerator 2 from caching, i.e. the accelerator was reclaimed for acceleration. If there were no communication back to the host bank, the bank would not know when it would be safe to hand over the accelerator PLMs; this could result in data corruption (if the handover were too early and thus accelerator activity overwrote the not-yet-read block) or in unnecessary waiting to guarantee that the access completed.

Write accesses also require an acknowledgment message back from accelerators to their host bank. Consider two consecutive writes from different cores to the same block whose data is cached by an accelerator. Without an acknowledgment message, correctness would have to rely on an interconnect with point-to-point ordering, which is a constraint that we do not wish to impose since it limits Roca's applicability.

6.3.2 Host Bank Organization

Roca uses two ideas to form host banks whose capacity can be expanded to exploit accelerator PLMs. First, it leverages selective cache ways to integrate accelerators with PLMs of varying sizes, which also serves as a simple mechanism to dynamically adapt to the intermittent availability for caching of accelerators. Second, it expands the tag array in the host bank to track accelerator blocks, thereby requiring minimal changes to accelerator designs.

Figure 6.5 shows the organization of a 4-way Roca host bank. The in-bank data array only contains ways 0 and 1. The tag array, however, tracks these two *local* ways as well as two *remote* ways, i.e. ways whose data blocks are stored in accelerators. Storage for tags and state of remote blocks is where the bulk of Roca's area overhead lies. In addition, logic in the bank controller is required to handle on-chip communication related to cache block transfers to/from remote Roca banks, as well as a *Way Enable Register* to configure which ways to enable. Software can access this register to enable/disable ways as the corresponding accelerators change their availability.

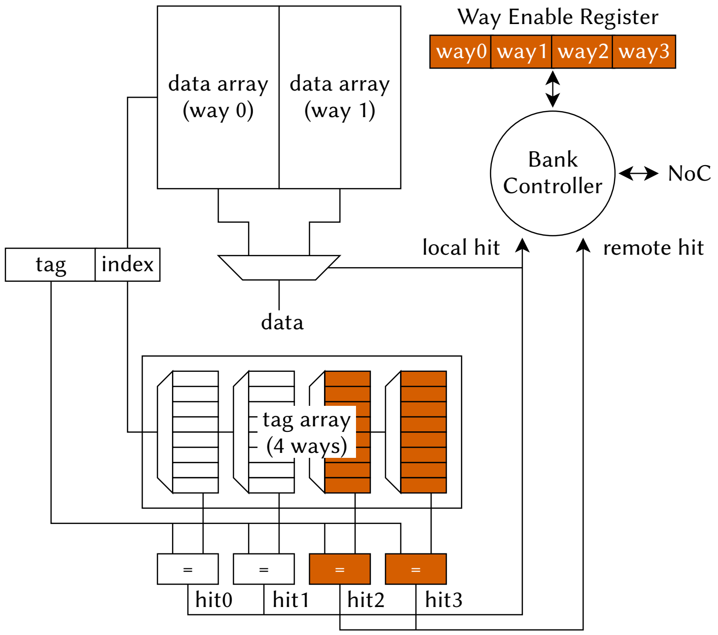


Figure 6.5: 4-way ROCA host bank. Two ways are local to the bank; the other two are remote, i.e. their data arrays are accessed via the interconnect. Shaded in orange are the hardware structures necessary to convert a regular bank into a ROCA host bank.

6.3.3 Way Allocation in Logical Banks

Way allocation is performed at design time to coalesce accelerators with diverse PLM sizes into the same logical bank. Figure 6.6 illustrates way allocation through several examples of a Roca logical bank composed of a host bank and three associated accelerators. Example 1 is the simplest; accelerators are all of the same size, which is a power of two that evenly divides the size of the memory in the host bank. Thus, in Example 1.a, the chosen number of sets is such that each accelerator can host one element per set. In other words, one way is assigned to each accelerator. The number of sets can be subsequently halved to double the number of ways, as 1.b shows.

Examples 2 and 3 are more diverse in their accelerator PLM sizes and therefore are more realistic. Both examples show that higher associativity (i.e. smaller numbers of sets) helps minimize waste due to uneven accelerator memory provisioning. In practice, an associativity typical for last-level caches (10 to 20 ways) is enough to result in negligible or no memory waste, as illustrated in examples 2.b and 3.b.

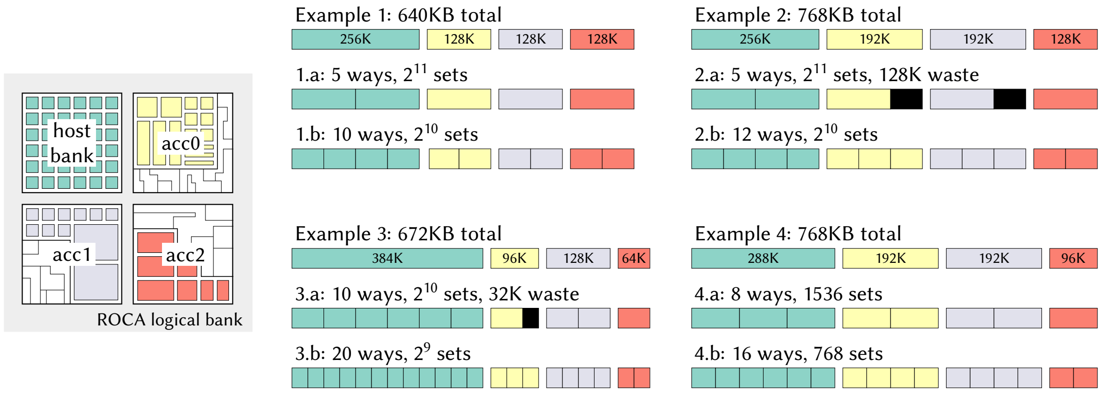


Figure 6.6: Way allocation examples for a ROCA logical bank with three accelerators, assuming a block size of 64 bytes.

The number of sets in all three examples discussed so far is a power of two, which makes the tag/index acquisition logic a simple bit selection from the block address. Examples 4.a and 4.b show way allocations that result in non-power-of-two numbers of sets. This is in principle a plausible option in the unusual scenarios where wasting some memory is unacceptable and the necessary associativity to achieve zero waste with power-of-two numbers of sets is prohibitive. However, designers need to consider the drawbacks of this choice before committing to it. Feasible numbers of sets are limited to those whose arithmetic modulo have a fast hardware implementation, i.e. numbers of the form $2^c * (2^n - 1)$ or $2^c * (2^n + 1)$, where c and n are non-negative integers [215]. More importantly, tags need to be enlarged to include all bits in the block offset, since the modulo operation cannot be constricted to just the least significant bits of the block address.

Designers are free to assign different numbers of accelerators to logical banks, allocating ways as they see fit. This can result in varying sizes and associativity across logical banks; given enough accelerators, however, this variability can be minimized by uniformly distributing accelerators based on their PLM sizes across the chip.

6.3.4 Impact on Cache Coherence Traffic

The intermittent availability of accelerators and the use of way allocation can have a profound impact on coherence traffic. For instance, maintaining a directory cache embedded in an inclusive LLC comes with substantial overhead, since the blocks cached in an accelerator about to be reclaimed would have to either be recalled from the private caches or relocated.

A simpler alternative is to give up LLC inclusion. Implementing a standalone directory cache guarantees that recalls can be made infrequent (by having enough associativity in the directory cache), and dissociates coherence from the last-level cache; logical banks are then free to be of any size and associativity, and to silently flush dirty blocks from reclaimed accelerators. The cost of having a standalone directory cache is the storage overhead of its tag array, which comes for free in an inclusive LLC. This cost, however, is modest. For example, a standalone directory cache adds 2.5% of overhead to the last-level cache when the latter is 8 times larger than the sum of the private caches, a typical ratio for inclusive designs [154]. This relative overhead grows as the shared-to-private ratio shrinks, reaching 11% when the shared cache is of the same size as the sum of the private caches [171].

6.3.5 Coalescing and Exposing PLMs

Roca exposes accelerator PLMs to the cache substrate through an additional memory port managed by a Roca *controller*. To describe these two additional components, which require a small amount of logic in the form of multiplexers and in some cases a small lookup table (LUT), we use the Sort accelerator described in Section 6.2.1.

Figure 6.7 depicts a memory-centric view of the Sort accelerator, in which each arrow represents a memory port. We first focus on the PLM manager, shaded in gray: its role is to export and coalesce SRAM banks into multi-ported memories [193]. For example, the 64 banks of *bubble-regs* are exported as a 64-port, 32b-wide PLM that has 32 ports connected to each of the parallel bubble sort and merge sort logic blocks. Given that the number of banks in the PLM is a power of two, the PLM manager is implemented as a trivial address translation unit that via bit selection determines for each access the correct bank and offset within it.

We now describe the Roca port added to the PLM manager, shaded in orange in the figure. Its goal is to transfer a cache block with minimum latency, whose lower

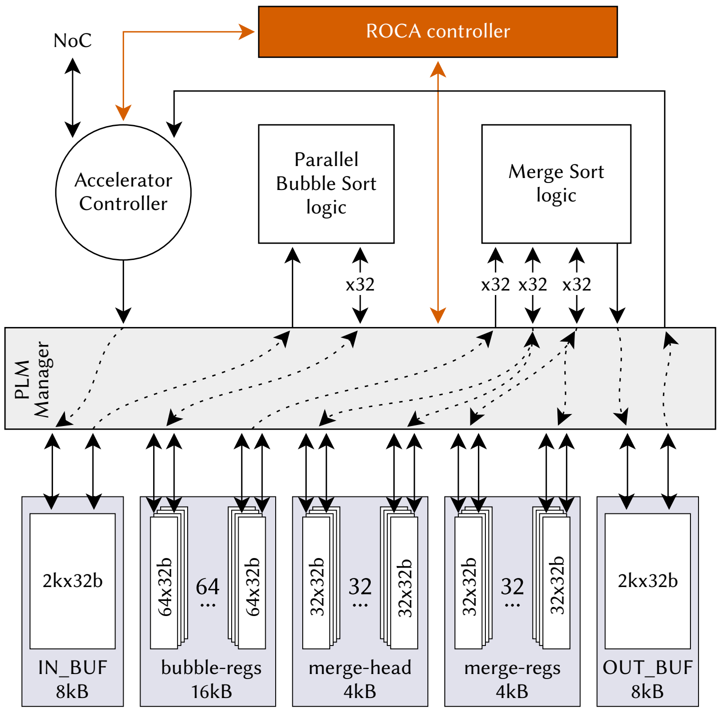


Figure 6.7: Memory ports in the Sort accelerator. The PLM manager aggregates SRAM banks to export them as multi-ported memories. An additional NoC-flit-wide port is exported to the ROCA controller, shaded in orange.

bound is imposed by the serialization in the NoC, i.e. one flit per cycle. Thus, the Roca port aggregates all the accelerator SRAMs (totaling 40KB) through an interface of the same bit width as the NoC's flit length, which for this example we assume to be 128b. The attachment to the SRAMs is attained by multiplexing the SRAM control signals, since Roca-enabled caching and acceleration do not overlap in time.

The addressing logic for the Roca port is in general not as trivial as that for regular accelerator ports; the number of banks is rarely a power of two and the banks are not uniform in their size and bit width. The complexity of the addressing logic can be minimized by choosing an appropriate arrangement of the accelerator SRAMs; furthermore, an adequate arrangement can minimize SRAM waste and maximize bandwidth to match the target of NoC flit per cycle.

Such an arrangement for the Sort accelerator is shown in Figure 6.8: banks are accessed in pairs resulting in a bandwidth of 128b per cycle, with 64b/cycle coming

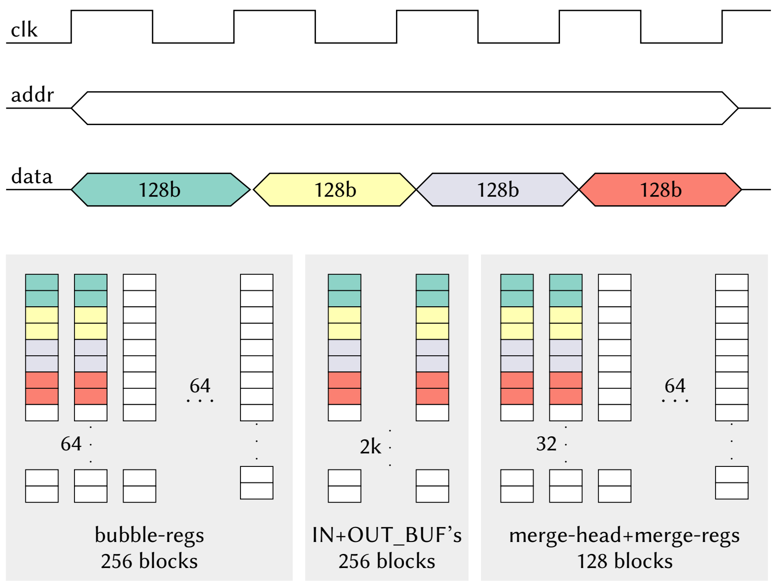


Figure 6.8: SRAM arrangement for the ROCA controller in the Sort accelerator. Adequate pairing of dual-ported banks brings bandwidth to one NoC flit (128b) per cycle.

from each bank thanks to exploiting the two ports of the dual-ported SRAMs. The SRAMs in each pair do not have to come from the same original PLM; for instance, *IN_BUF* and *OUT_BUF* as well as *merge-head* and *merge-regs* are paired together for caching purposes yet operate in separate PLMs during acceleration.

The Roca controller converts block addresses within the total of 640 64-byte blocks (40KB) of memory into a physical offset within the appropriate pair of SRAM banks. To achieve this with minimum latency, integer division is to be avoided. Thus, we group banks of the same size as shown in Figure 6.8 and then sort them by size in descending order, obtaining three groups of 256, 256 and 128 blocks. Block addresses with the 9th bit set go to the third group; all other addresses go to the first two groups, which are matched via bit selection since both groups are of the same power of two size. In all cases the offset within a group is also obtained via bit selection, which results in trivial logic.

Handling less uniform groups of SRAMs is also possible while avoiding integer division. A viable option is to have a small lookup table (LUT) to match the upper order bits (over that of the group sizes' greatest common factor) to the appropriate group;

the offset within a group is then calculated via a shift, whose amount is also precomputed in the LUT. Another option, when the size of the banks is the same yet their number is not a power of two, is to follow the procedure presented by Seznec [215], which we mentioned when describing way allocation in logical banks (Section 6.3.3). A last option, when dealing with highly non-uniform cases, is to either discard some memory for caching, or pad the accelerator with additional memory (clock-gated when the accelerator is not in Roca mode) in order to obtain more regular sizing across banks.

The same *"to pad or to waste"* decision is faced when coalescing SRAMs from PLMs of different bit widths. For example, if each bank is 8b-wide then 8 dual-ported banks need to be accessed to sustain a bandwidth of 128b per cycle. If the width is instead 7, then accessing nine banks of which one is extended to 8 bits is an advisable option. Accelerators with diverse PLM bit widths are a common occurrence, despite what the Sort accelerator example might suggest.

So far we have assumed that the SRAM banks are dual-ported, i.e., in a single clock cycle two non-conflicting accesses to the same bank are allowed. Using single-ported SRAM banks is also possible with Roca; the per-bank bandwidth therefore halves, requiring the number of banks (for Roca or for acceleration) to double in order to meet the original bandwidth.

The NoC's flit length is also a variable that must be taken into consideration. In general, the larger the flit, the more aggregate bandwidth is required from the SRAMs, and therefore the larger their groups will be.

6.3.6 ROCA-to-Acceleration Transitions

When an accelerator is recalled from Roca, two options are available to maintain the LLC in a consistent state. A first option is to relocate to other LLC banks the most frequently accessed blocks as well as the dirty ones. This option, however, is costly in hardware: timestamped block access counters such as bucketed LRU [208] are necessary to enable the ordering across blocks in the same accelerator, which holds blocks from all sets in the cache. An alternative, more practical option is to silently evict blocks, flushing to DRAM if necessary. In practice this is not very different from actively migrating frequently-accessed blocks: subsequent misses to said blocks will place them in other LLC banks, and, assuming heavy accelerator activity, these blocks will eventually be placed in LLC host banks, which are always available.

From the accelerator's point of view, the flushing of dirty blocks is simply a regular block read requested from the controller in Roca's host bank. The host bank is not just the main serialization point for coherence as we discussed in Section 6.3.1; it is also a serialization point for accelerators, since an accelerator can only switch to acceleration once its Roca host bank has completed the flushing of all the dirty blocks the accelerator was holding.

Choosing to just flush dirty blocks has an additional advantage over more complex options in that it minimizes the transition latency: its lower bound is the NoC's bandwidth, since it serializes the read requests that precede the corresponding flushes to DRAM. The importance of the transition latency is nonetheless relative to the frequency at which the accelerator switches between acceleration and caching. Thus, an accelerator that is constantly being required to accelerate is a poor target for Roca, since the positive effect of temporarily increasing the aggregate cache size is dominated by the latency from flushing dirty blocks and subsequent cache misses. We thus let software decide when to enable/disable Roca on accelerators, since software has complete information about the system. Our results in Section 6.5 quantify the accelerator invocation frequency below which Roca should be enabled.

6.4 Area Overhead

Roca's area overhead is relative to the scenario it is compared against. For instance, a system with a fixed area budget and initially no accelerators could benefit from including some Roca-enabled accelerators, trading off part of the original cache for this. The resulting cache size would depend on how much of the accelerators' area were devoted to memory; e.g., assuming accelerators were 69% memory (as discussed in Section 6.2), each unit of cache capacity would approximately require 44% more area when implemented in Roca than as regular cache. In return for this area investment, however, the system would have gained the potential of drastically increasing the performance and efficiency of certain workloads by using the accelerators.

A more precise way to assess Roca's area overhead is to consider the baseline system as one already equipped with accelerators. Expanding the existing cache with Roca has then a modest cost, split between (1) tag storage for a standalone directory cache if it previously was embedded in the last-level cache, as discussed in Section 6.3.4, and (2) additional tag storage to track cache blocks in Roca accelerators. Storage for the cache tag array is unlikely to exceed 10% of the storage needed for the data it tags. To be concrete, let us assume a 48-bit physical address space and 64-byte (2^6) cache blocks; then, in case the number of banks is not a power of two, tags need to include the entire block offset, resulting in a (48-6+2)/(64*8)=8.5% tag array area overhead relative to the data array, assuming two bits to keep valid and dirty states. In more common scenarios in which the number of sets is a power of two (and thus tags do not include the lower-order bits of the block offset) the relative overhead shrinks to, for instance, 6.6% or 5.4% assuming 2^{10} and 2^{16} sets, respectively.

Roca requires additional logic whose area overhead is however negligible compared to that of tag storage. This logic enables Selective Cache Ways [10] in the host bank (discussed in Section 6.3.2), and multiplexes the control signals of accelerators' SRAM banks while appropriately translating addresses to coalesce them into a single PLM (Section 6.3.5).

6.5 Energy and Performance Evaluation

6.5.1 Experimental Methodology

Simulation platform. We extend Cargo (see Section 5.4) to support parallel simulation of the memory hierarchy of a cache-coherent multicore system, in a similar fashion to Sniper [47] or Zsim [209].

Modeled Systems. We conduct full-system simulation of an i386 machine running Linux. We initially model a 5x5 tiled CMP with 16 general-purpose cores as

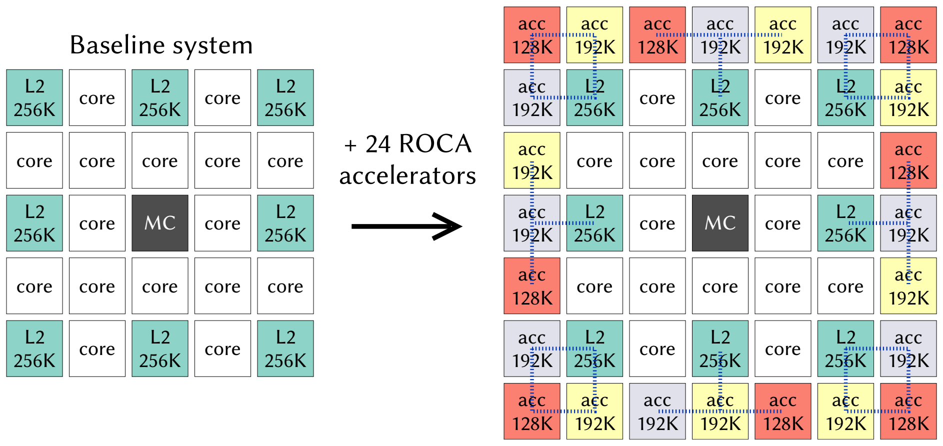


Figure 6.9: The two simulated systems. The baseline system (left) is a 16-core 5x5 CMP with a 2MB S-NUCA LLC. We augment it with 24 ROCA-enabled accelerators to form a 7x7 CMP with a 6MB ROCA LLC (right). Dashed lines show the eight logical banks into which the address space is split. MC stands for memory controller.

the baseline system, which we then augment with 24 Roca-enabled accelerators as depicted in Figure 6.9, or with enlarged cache banks as described below.

Our modeled cores are single-threaded and in-order, with a two-level cache hierarchy. We choose in-order cores to maximize performance sensitivity to variations in cache latency, as is commonly done when studying the impact of changes to the cache hierarchy (e.g. [208]). Energy efficiency is modeled by combining our performance models with: McPAT 1.0 [155] for core/directory/L1 power, CACTI 6.5 [185] to obtain power/latency numbers for sequential-access low-leakage L2 cache banks, DSENT [226] for NoC link/router power, and the memory power model in [81]. We approximate memory power by assigning 2.78W to background power and 51nJ per access for a single DIMM as done by Sampson and Wenisch in [207].

Modeled Last-Level Caches. We model a Shared L2 LLC in all experiments. We do not consider private caches beyond the L1's nor a L3 cache in order to maximize sensitivity to L2 hit latency, which is common practice among NUCA studies [109, 135, 148]. We report all results normalized over those from the baseline system, which has a 16-way 2MB 8-bank S-NUCA LLC [135].

We augment the baseline system with 24 Roca-enabled accelerators and convert the regular L2 banks into Roca host banks, which renders a 7x7 system as depicted

Cores	16 cores, i386 ISA, in-order IPC=1 except on memory accesses, 1 GHz
L1 caches	Split I/D 32KB, 4-way set-associative, 1-cycle latency, LRU replacement
L2 caches	8-cycle latency, LRU replacement S-NUCA: 16 ways, 8 banks, 2MB or 8MB total Roca: 12 ways, 8x(256K+2x192K+128K)=6MB
Coherence	MESI protocol, 64-byte blocks, standalone directory cache
DRAM	1 Controller, 200-cycle latency, 3.5GB physical
NoC	5x5/7x7 mesh, 128-bit flits, 2-cycle router traversal, 1-cycle links, XY routing
OS	Linux v2.6.34

Table 6.2: Configuration of the simulated system.

in Figure 6.9 (right). We conservatively assume that on average the accelerators' reusable memory area is slightly below the typical (as discussed in Section 6.2) 69%; we thus model 16 accelerators with 192KB of memory and 8 with 128KB, with a way allocation as in Example 2.b in Figure 6.6. The resulting memory average corresponds to 66% of the total area (100% memory would mean that each accelerator has 256KB of memory, i.e. the same capacity as a regular L2 bank of the same area).

We compare the augmented system against a system that has the same total area, but instead of integrating accelerators it features larger S-NUCA banks. We conservatively assume that this system can be laid out without changing the chip's tiled structure, and therefore model a 5x5 8-bank 8MB S-NUCA configuration. Table 6.2 summarizes the characteristics of the three systems.

Energy Consumption Model for Roca **banks.** We use CACTI to model leakage and per-access energy in the Roca banks and to account for the energy consumption overhead of the additional tag storage. Furthermore, to account for a worst-case scenario we purposely overestimate the leakage power of accelerator-specific logic (i.e. the logic that does not participate in Roca) as if it was induced by SRAMs instead of regular logic.

Latency Model for Roca **banks.** We assume that cache blocks are accessed from both Roca accelerators and LLC host banks in 8 cycles: 4 cycles for 128b-flits over the NoC and 4 cycles for message processing and tag lookup. This is in addition to a 2-cycle-per-router NoC latency, as shown in Table 6.2.

Workloads. We assess the performance impact of varying Roca accelerator availability by simulating multiprogrammed workloads that we assume are not amenable to acceleration. Ideally, we would simulate a mix of accelerated and non-accelerated workloads. However, we decide against this for two reasons. First, there exist few accelerator benchmarks, and those that are available (e.g., [201]) are not integrated into larger, real applications. Second, even if those accelerated applications were available, we would not be able to obtain insight with respect to accelerator availability rates beyond those present in the particular applications under study.

We therefore sweep in simulation the availability rate of Roca accelerators, studying its impact on energy efficiency and performance when CPU cores are executing a total of 45 multiprogrammed workloads taken from SPEC06. We simulate those that when multiprogrammed can fit in the systems's 3.5GB of physical memory. The 13 SPEC06 benchmarks used are thus: astar, gobmk, gromacs, h264ref, hmmer, libquantum, namd, omnetpp, perlbench, povray, sjeng, soplex and sphinx3. The remaining 32 multiprogrammed workloads result from random combinations of those 13 benchmarks, with repetitions allowed.

We use the SPEC06 reference input sizes, launching as many benchmarks as cores. For each benchmark we fast-forward for one billion instructions to then record for 256 million instructions. Threads that reach the 256 million instruction mark earlier than others continue running, so that they still contend for shared resources. We do not pin threads to particular cores; the Linux scheduler is free to migrate threads across cores as it sees fit. All benchmarks are compiled with gcc v4.6.3 enabling -O2 optimizations.

Metrics. Energy efficiency is presented in billions of instructions per Joule (BIPJ). Given that our workloads are multiprogrammed, performance is considered equivalent to IPC throughput, i.e. $\sum_i IPC_i$, which is a consistent throughput metric [179].

Model of Accelerator Memory Availability. We make the following assumptions when simulating the availability of accelerator PLMs:

1. Accelerators are always considered to run in bursts of 100K cycles (100us given the 1GHz clock). Assuming a speedup over software of 100x, in this amount of time an accelerator can do an amount of work that is roughly the same as a CPU can do in a scheduling quantum, which is typically around 10ms. Therefore, when in our experiments we say that an accelerator is 50% active, we mean that it alternates between bursts of acceleration and Roca caching, each 100us long. That is, it has an **activity period T**=0.2ms. Similarly, an accelerator that is 1% active runs every 10ms, i.e. T=10ms.

2. We do not consider the work done by the accelerators as part of the workload: we limit our attention to the *overhead* that accelerator activity causes on the cache substrate. Thus, when reporting energy efficiency, we only count the leakage and dynamic power of accelerators for the time period they serve as Roca banks.

3. If an experiment sweeps over the accelerator use rate of *n* Roca banks, throughout the simulation we consider only that same set of *n* banks. Further, their bursts of activity all start and terminate in unison, i.e. at the same simulated time. This is done to measure a worst-case scenario that maximizes block flushes from the accelerators as they reclaim their PLMs from Roca.

6.5.2 Evaluation Under No Accelerator Activity

We first measure the energy efficiency and performance of a cache built on Roca relative to a standalone S-NUCA, without considering any accelerator activity. This would be a common scenario in general-purpose Roca-enabled systems: since the workload is not known at design time, the integrated accelerators are highly unlikely to apply to the actual workload. However, with Roca they can nonetheless provide value by expanding the LLC.

Figure 6.10 shows the throughput and energy efficiency improvements of the 6MB Roca and 8MB S-NUCA configurations over the 2MB S-NUCA baseline. Each line plots the improvement of all workloads for each configuration, with the results sorted so that every line is monotonically increasing. An additional dashed line per configuration is also shown to represent the improvements' cumulative geometric mean.

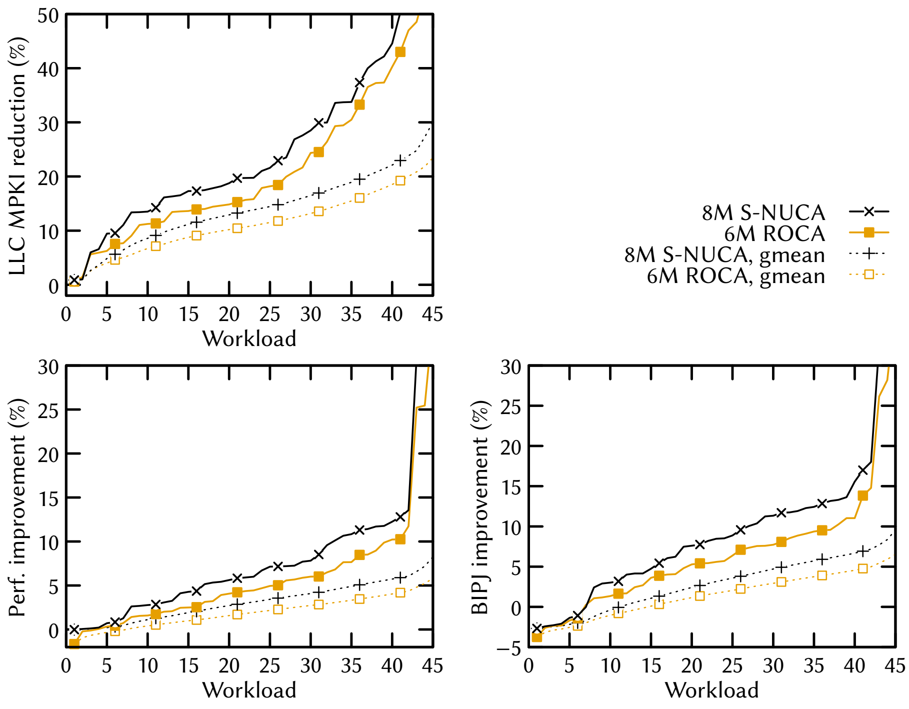
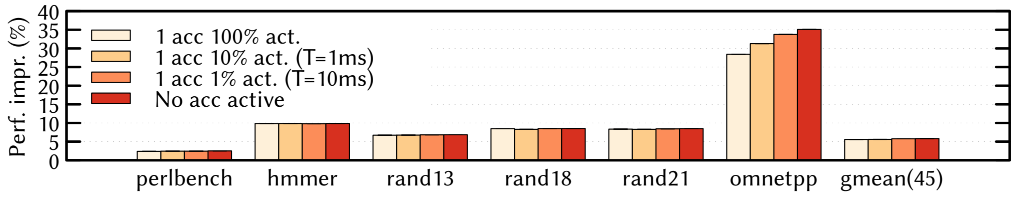


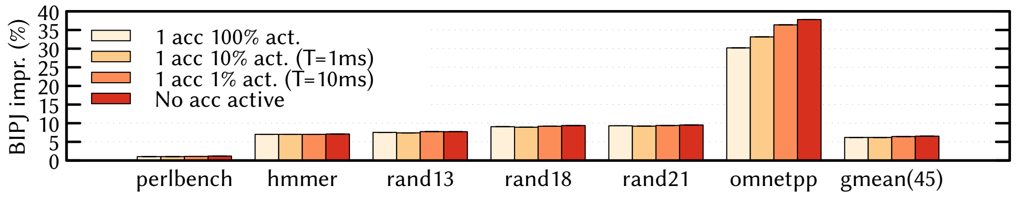
Figure 6.10: MPKI, performance and energy efficiency improvements over the 2MB S-NUCA baseline for all workloads for 8MB S-NUCA and 6 MB ROCA configurations. All accelerators in ROCA are inactive.

We observe that the results' gap between the 6M Roca and the same-area 8MB S-NUCA is commensurate with their 25% difference in capacity: Roca realizes 78% of the MPKI reduction of the 8MB S-NUCA, while relative performance (70%) and energy efficiency (68%) improvements are slightly lower. This decrease is explained by the additional level of indirection that Roca requires: L2 accesses that hit in an accelerator bank require NoC transfers that are not necessary in S-NUCA, where each regular bank comprehends both cache tags and data. These NoC accesses between the Roca host bank and associated accelerators result in additional overhead by consuming energy and increasing hit latency.

Some workloads show an energy efficiency degradation over the baseline for both configurations close to 5%. This is explained by the unresponsiveness of these workloads to increased last-level caching, due to either streaming access patterns or the workload already fitting in the baseline LLC.



(a) Performance improvement



(b) Energy efficiency improvement

Figure 6.11: Performance and energy efficiency improvements over 2MB S-NUCA for 6MB ROCA with one accelerator intermittently active. Shown are improvements for some workloads, plus the gmean for all workloads.

The low performance and energy overhead of Roca compared to a same-area S-NUCA *in the absence of accelerator activity* is therefore established. In the remainder of our evaluation we consider the impact of intermittently removing accelerator banks from Roca, which happens when accelerators become active and reclaim PLMs.

6.5.3 Same-Logical-Bank Accelerator Activity

We now consider the effect on performance and energy efficiency of the intermittent activity of Roca-enabled accelerators within the same logical bank. Such a scenario would be most likely found on a general-purpose architecture owned by a power user whose workload we assume can exploit only up to three of the integrated accelerators. We first study the case of a single active accelerator, and then consider the worst case of three accelerators being on the same logical bank; this results in the loss of 8 ways out of the 12 ways that are assigned to a logical bank.

Single Active Accelerator. Figure 6.11 shows the improvements in performance and energy efficiency over the baseline system for 6MB Roca with one of the 128K accelerators switching between acceleration and caching. The activity rate of this Roca bank shows negligible impact on performance and energy efficiency. Only the

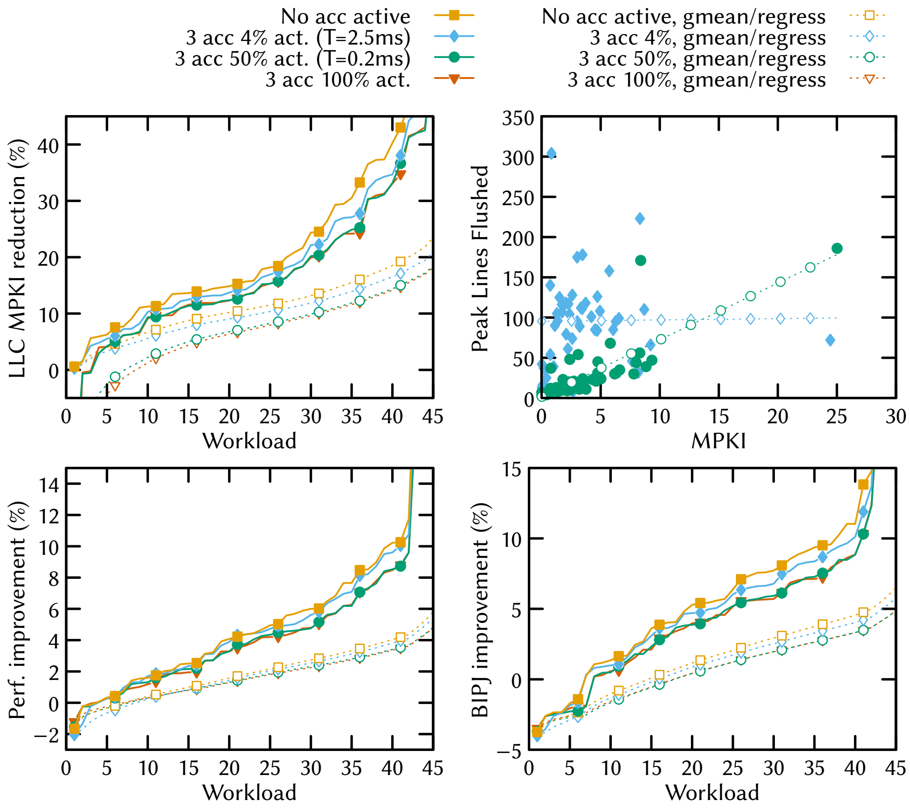


Figure 6.12: Improvements for 6MB ROCA over 2MB S-NUCA and characterization of peak number of blocks flushed vs MPKI (top right), for varying accelerator activity of 3 accelerators in the same logical bank.

workloads with large MPKIs are sensitive to the slight L2 capacity reduction caused by the intermittent loss of a single Roca bank.

Three Same-Logical-Bank Active Accelerators. Figure 6.12 shows the performance and energy efficiency improvements over the baseline for this scenario. We observe that the difference between the worst case (100% activity) and best case (no accelerators active) is small. This is due to the presence of 4 local ways in the Roca host banks; having one eighth of the address space only cached by 4 LLC ways does not greatly impact these workloads. Frequent accelerator use (T=0.2ms) has similar impact to 100% activity in MPKI, performance and energy efficiency since in 0.1ms caching there is not enough time to make effective use of the three Roca banks. Less frequent accelerator use (4% activity, T=2.5ms) yields more positive results across the three metrics, since the caching window is larger.

A side effect of a larger caching window is that more dirty blocks must be flushed to DRAM upon accelerator reclamation. This can be seen in the top right plot, where for each run the peak number of blocks flushed by a single accelerator upon a reclaim is shown against the run's MPKI together with their computed linear regression. A larger caching window thus results in larger peak values. However, the correlation between peak flushes and MPKI is stronger as the caching window narrows; the intuition behind this is that given a large enough window, the Roca bank will be fully populated regardless of how cache-hungry the workload may be. Short windows, on the other hand, are highly sensitive to the miss rate of the workload: only high-MPKI workloads are capable of inserting a significant amount of blocks in the Roca bank.

6.5.4 Chip-Wide Accelerator Activity

We complete our performance and energy efficiency evaluation by studying the impact on caching of intense accelerator activity across the chip. In this scenario the 24 accelerators switch in unison between caching and acceleration.

Figure 6.13 shows the MPKI, performance and energy efficiency improvements for this configuration. The worst case for caching (100% activity for the 24 accelerators) remains on average close to the baseline: 0.8% and 4% average performance and efficiency degradation, respectively. This difference is explained by the even larger (close to 10% on average) MPKI drop caused by the low associativity (4 ways) of the Roca host banks, which for cache-sensitive workloads are outperformed by the equally-sized 16-way baseline cache.

Between zero and full accelerator activity we observe how critical is the activity period T: a caching window of 10ms (1% acc. activity) yields performance and energy efficiency within—respectively—10% and 20% of that without accelerator activity. The higher energy overhead is due to the flushing of blocks upon accelerator reclamation.

A shorter caching window (T=2.5ms) results on average in less than half the zero-activity gains in performance and negligible gains in energy efficiency over the baseline. Moreover, it leads to the highest peak number of flushes (330) upon accelerator

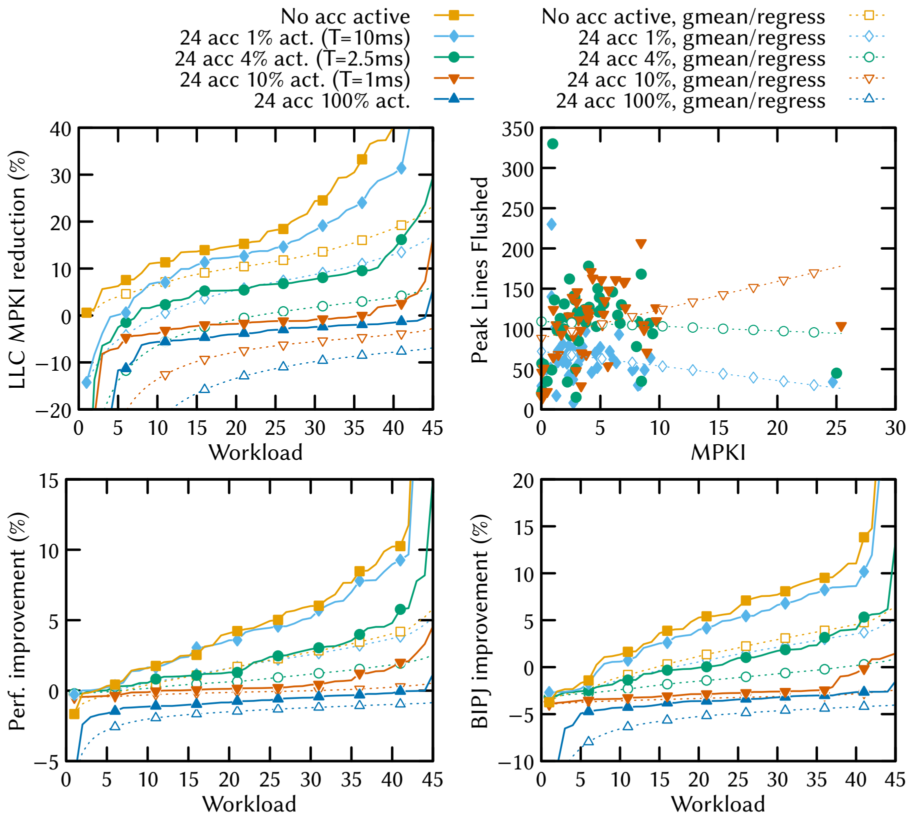


Figure 6.13: Improvements for 6MB ROCA over 2MB S-NUCA and characterization of peak number of blocks flushed vs MPKI (top right), for varying accelerator activity of all (24) accelerators.

reclamation observed over all simulations. An even shorter caching window (T=1ms) is hardly an improvement over the full-activity scenario: performance is similar to that of the baseline, yet energy efficiency is 2% below.

6.5.5 Summary of Results

Our results show that accelerators that are not used frequently (i.e. with idle windows of 10ms or longer) are prime candidates for Roca. The average performance and energy efficiency improvements obtained from them for non-accelerated workloads are, respectively, 70% and 68% of those from regular cache banks of the same area, while providing orders-of-magnitude improvements for workloads suitable for acceleration. Furthermore, our results show the importance of allocating a certain portion of the Roca LLC to host banks (e.g. 2MB out of 6MB in our study) in order to limit performance and efficiency degradation for non-accelerated workloads when most accelerators are active.

Not all memory-rich accelerators are suitable for Roca, however. An exception to this recommendation are accelerators with rigorous real-time constraints, with deadlines in the order of microseconds; the delay to flush dirty blocks to DRAM when the accelerator is reclaimed from Roca could compromise the meeting of such deadlines. For example, flushing 330 64-byte blocks, assuming a sufficiently buffered DRAM controller and 128b NoC flit length, would take around 10,560 cycles, i.e. 10.5us at a 1GHz clock rate.

6.6 Related Work

The importance of accelerator PLMs is confirmed by the amount of area dedicated to them [162]; this key role of PLMs has sparked recent efforts to automate and optimize the design of heavily banked memory subsystems for accelerators [193].

Decoupling the utility of specialized hardware from particular workloads has motivated proposals, such as Smart Memories [167], CHARM [64] and LSSD [189], that attempt to partially match the performance and energy efficiency gains of accelerators without giving up programmability. Our approach differs from theirs in that we relinquish programmability not to compromise on performance or efficiency. With Roca, accelerator utility is however decoupled from the workload by augmenting the LLC with accelerator memories when accelerators would otherwise be inactive.

An interesting approach that shares Roca's objective of reducing the opportunity cost of accelerator integration is Stash [143], whose goal is to minimize on-chip copies of data by implementing a hybrid storage element for accelerators, thereby combining the benefits of both caches and software-managed scratchpads. Stash differs from Roca in that it requires changes to the cache coherence protocol, and similarly to the proposals discussed in Section 6.2 (i.e. [65, 94, 162]), it is not a good fit as a substitute for high-bandwidth PLMs, since hiding the additional latency of accessing these memories would significantly complicate accelerator designs.

Recent NUCA research has put emphasis on trading cache capacity to reduce hit latency via controlled block placement and replication, e.g. ASR by Beckmann et al. [23], R-NUCA by Hardavellas et al. [109], and Locality-Aware Data Replication by Kurian et al. [148]. These techniques were designed for always-available, equally-sized banks; extending them to support the requirements of banks such as the ones coming from Roca accelerators, which are of diverse sizes and intermittently available, would be valuable future work.

6.7 Summary

In this chapter we have presented Roca, a technique to exploit the abundant private local memories in accelerators to mitigate the opportunity cost of their integration. Roca enables accelerators to provide *utility* even when they cannot directly speed up a workload, by exposing their private local memories to the cache substrate. Our implementation of Roca is practical, requiring minimal modifications to both accelerators and the cache substrate, and incurring a modest area overhead that is almost entirely due to additional tag storage.

Our results show that, relative to a 2MB S-NUCA LLC, a 6MB Roca LLC built upon typical accelerators (i.e. whose area is 66% memory) can, on average, realize 70% of the performance and 68% of the energy efficiency benefits of a same-area 8MB S-NUCA configuration. Further, our results suggest that accelerators with windows of inactivity of 10ms or longer are prime candidates for Roca.

Chapter 7

Future Directions

In this chapter we explore future research avenues inspired by the work presented in this dissertation.

7.1 Cross-ISA Virtualization

In their 1974 seminal paper, Popek and Goldberg [195] define virtualization as the fulfillment of the following requirements:

> *"As a piece of software a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources."* [195]

Can we then provide cross-ISA virtualization in a portable cross-ISA emulator like QEMU? The efficiency requirement is clearly the hardest to meet. Despite the performance improvements presented in this dissertation (e.g., Figures 4.5 and 4.6), single-threaded performance of emulated workloads is still far from being "essentially identical" to native execution. DynamoRIO and Pin show that DBT can be used to achieve near-native performance for same-ISA user-mode emulation (Figure 4.13, top row), and MAMBO-X64 shows similar near-native performance for Aarch32-to-Aarch64 user-mode emulation [76]. This suggests that narrowing the performance gap between these tools and portable cross-ISA ones is feasible. Such an effort would probably require a combination of old and new techniques regarding the sources of emulation overhead that we identified in Section 2.2.3:

Floating point emulation. Two complementary approaches could be followed. First, the code needed for the FP *fast path* in Figure 4.1 could be generated directly by the DBT engine, thereby removing the overhead of one helper call per emulated FP instruction. Second, our work on FP emulation could be combined with vectorization techniques to achieve higher performance without giving up correctness. Guo et al. [104] show encouraging results for emulating vectorized ARM code. QEMU has recently gained support for vectors in its IR [25], which makes it a good candidate for evaluating this idea.

Indirect branch handling. Trace-based execution could lower the cost of indirect branch handling. However, as discussed in Section 2.2.3.1, using traces in full-system emulation remains challenging. Recent work by d'Antras et al. [75] shows techniques that can yield performance gains on this front without the use of traces, for instance by leveraging the return address predictor of the host.

Code quality. Some solutions in the literature demonstrate that the cost of high code quality can be amortized for hot code blocks. Good examples are HQEMU [118], which leverages LLVM to compile hot code blocks/traces, and HERMES [249], which performs a final host-specific optimization pass on the generated code.

SoftMMU. In portable emulators, a softMMU is probably inevitable in order to emulate the guest's virtual memory. Our results show that a dynamically-sized softMMU (Section 4.2.4.1) yields improvements over a static one, yet full-system emulation is still on average 1.8× slower than user-mode for SPECint06[1].

[1]This slowdown can be easily computed: the average SPECint06 slowdown of full-system and user-mode emulation over native execution is 7.6× (Figure 4.12) and 4.2× (Figure 4.13), respectively. The softMMU-induced overhead is therefore $7.6/4.2 = 1.80\times$

Alternatives to the softMMU that leverage virtualization hardware on the host have shown encouraging results. For instance, Spink et al. [222] report an average speedup of 2.5× for SPECint06 in QEMU. However, this approach has only been demonstrated with guests whose virtual address length is shorter than that of the host, e.g. 32-on-64-bit emulation. Further research is needed to determine whether this approach could be made to work efficiently to support guests with the same virtual address length as that of the host, e.g. 64-on-64 bit emulation.

7.2 Scalable Simulation of Heterogeneous Systems

This dissertation demonstrates the feasibility of a scalable cross-ISA machine emulator, which can be used as a simulation *front-end*. Having such a front-end is a necessary requirement for scalable, cross-ISA, full-system simulation. A scalable front-end, however, is not sufficient to fulfill this goal; a scalable timing *back-end* is also needed. As reviewed in Section 2.5.1, recent work on same-ISA simulation has shown that scalable timing back-ends are feasible, although their scalability comes at the expense of accuracy when simulating events that contend on shared resources [47, 209].

Achieving both accuracy and scalability in a timing back-end while accurately modeling contention remains an open research question. This problem is particularly challenging when applied to DRAM simulation: small alterations to event ordering can lead to very different outcomes, particularly under unfair scheduling algorithms in the memory controller [184].

7.3 Emulation of Multi-ISA Machines

Asymmetric processors typically integrate one large, high-performance core and several small, low-performance ones. This arrangement allows the system to efficiently execute both serial and parallel workloads [115, 146]. An argument following the rationale behind asymmetric cores could be made for multi-ISA systems, since these systems could benefit from the larger power-performance design space that using several ISAs can bring. Multi-ISA systems, however, have not yet materialized, probably due to the lack of an operating system that can handle them. Research on this topic was pioneered by Barbalace et al. [21], although the cores they use are not on the same chip, but connected through the PCIe bus.

Machine emulation could help spur further research on multi-ISA operating systems and architectures by enabling the evaluation of ideas in simulation. To this end, the emulator design presented in this dissertation could be expanded to support multi-ISA guests. Conceptually, this would require little more than the addition of a field to each translated block to keep track of the ISA it belongs to. Implementation-wise, however, in QEMU this change might require significant engineering effort, since QEMU is a large code base that has been written without ever considering multi-ISA support as a potential feature.

Chapter 8

Conclusions

In this dissertation we have first introduced the design of a novel DBT-based machine emulator that (1) scales on multicore hosts while remaining memory efficient via the use of a shared code cache, (2) correctly handles guest-host ISA differences in atomic instruction semantics without sacrificing scalability, (3) achieves high FP emulation performance by leveraging the host FPU, and (4) supports efficient instrumentation, which—among other uses—allows it to drive the execution of architectural simulators.

We have then presented two additional contributions that highlight the usefulness of machine emulation for conducting research on heterogeneous systems. First, we analyzed the trade-offs in different accelerator coupling models. Second, we developed and evaluated a novel technique to reuse the private memories of on-chip accelerators when they are otherwise inactive to expand the system's last-level cache, thereby reducing the opportunity cost of the accelerators' integration.

We believe that fast and scalable machine emulation will become an important piece of research infrastructure. To help fulfill this vision, we have integrated our emulator's implementation into QEMU, a popular open-source machine emulator and virtualizer. We hope that this effort will bear fruit by enabling others to perform more productive research across the different layers in computing systems.

Bibliography

[1] Chromium: Running unittests via qemu. https://www.chromium.org/chromium-os/testing/qemu-unittests

[2] Intel ARK. https://ark.intel.com

[3] PostgreSQL. https://www.postgresql.org

[4] SPEC. https://www.spec.com

[5] http://concurrencykit.org

[6] https://github.com/Cyan4973/xxHash

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

[8] IBM Power ISA, version 2.06 revision b. *Book I: Power ISA User Instruction Set Architecture*, 2010.

[9] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, 2006.

[10] David H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 248–259, 1999.

[11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against hardware. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44, 2011.

[12] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multi-core debugging. In *Proc. of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 193–206, 2009.

[13] 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*, pages 483–485, 1967.

[14] Ehsan K Ardestani and Jose Renau. ESESC: A fast multicore simulator using time-based sampling. In *Proc. of the Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 448–459, 2013.

[15] ARM ARM. Architecture reference manual. ARMv7-A and ARMv7-R edition. *ARM DDI C*, 406, 2012.

[16] Krste Asanović and David A Patterson. Instruction sets should be free: The case for RISC-V. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[17] Joshua Auerbach, David F Bacon, Perry Cheng, and Rodric Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proc. of the Intl. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, volume 45, pages 89–108, 2010.

[18] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2000.

[19] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *Proc. of the USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, pages 177–192, 2008.

[20] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, page 191, 2003.

[21] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: bridging the programmability gap in heterogeneous-ISA platforms. In *Proc. of the European Conf. on Computer Systems (EuroSys)*, pages 29:1–29:16, 2015.

[22] Kevin Barker, Thomas Benson, Dan Campbell, David Ediger, Roberto Gioiosa, Adolfy Hoisie, Darren Kerbyson, Joseph Manzano, Andres Marquez, Leon Song, Nathan Tallent, and Antonino Tumeo. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual.* Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013.

[23] Bradford M Beckmann, Michael R Marty, and David A Wood. ASR: Adaptive selective replication for CMP caches. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 443–454, 2006.

[24] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of the USENIX Annual Technical Conference (ATC)*, pages 41–46, 2005.

[25] Alex Bennée and Richard Henderson. HKG18-TR08: upstreaming SVE in QEMU, 2018.

[26] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[27] E. Bin, R. Emek, G. Shurek, and A. Ziv. Using a constraint satisfaction formulation and solution techniques for random test program generation. *IBM Systems Journal*, 41(3):386–402, 2002.

[28] 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, 2011.

[29] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi, and Steven K Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, (4):52–60, 2006.

[30] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *Proc. of the Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2013.

[31] Colin Blundell, E Christopher Lewis, and Milo MK Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006.

[32] Hans-J Boehm and Sarita V Adve. Foundations of the C++ concurrency memory model. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 68–78, 2008.

[33] Igor Böhm, Tobias J.K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 74–85, 2011.

[34] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[35] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[36] Justinien Bouron, Sébastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The battle of the schedulers: FreeBSD ULE vs. Linux CFS. In *Proc. of the USENIX Annual Technical Conference (ATC)*, pages 85–96, 2018.

[37] Clair Brown, Greg Linden, and Jeffrey T Macher. Offshoring in the semiconductor industry: A historical perspective [with comment and discussion]. In *Brookings Trade Forum*, pages 279–333, 2005.

[38] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 265–275, 2003.

[39] Derek Bruening and Vladimir Kiriansky. Process-shared and persistent code caches. In *Proc. of the Intl. Conf. on Virtual Execution Environments (VEE)*, pages 61–70, 2008.

[40] Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji. Thread-shared software code caches. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 28–38, 2006.

[41] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y Wang. Bringing virtualization to the x86 architecture with the original VMWare workstation. *ACM Transactions on Computer Systems (TOCS)*, 30(4):12, 2012.

[42] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture*, 12(1):1–206, 2017.

[43] Doug Burger and Todd M Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH computer architecture news*, 25(3):13–25, 1997.

[44] Harold W Cain, Kevin M Lepak, Brandon A Schwartz, and Mikko H Lipasti. Precise and accurate processor simulation. In *Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2002.

[45] M. D. Canon, D. H. Fritz, J. H. Howard, T. D. Howell, M. F. Mitoma, and J. Rodriguez-Rosell. A virtual machine emulator for performance evaluation (summary). In *Proc. of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, page 1, 1979.

[46] Luca P Carloni. The case for Embedded Scalable Platforms. In *Proc. of the Design Automation Conference (DAC)*, pages 1–6, 2016.

[47] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12, 2011.

[48] Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. Efficient memory virtualization for cross-ISA system mode emulation. In *Proc. of the Intl. Conf. on Virtual Execution Environments (VEE)*, pages 117–128, 2014.

[49] Jainwei Chen, Lakshmi Kumar Dabbiru, Daniel Wong, Murali Annavaram, and Michel Dubois. Adaptive and speculative slack simulations of CMPs on CMPs. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 523–534, 2010.

[50] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 269–284, 2014.

[51] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M Gillies. Mojo: a dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, pages 81–90, 2000.

[52] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 367–379, 2016.

[53] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S Bharadwaj Yadavalli, and John Yates. FX! 32: A profile-directed binary translator. *IEEE Micro*, (2):56–64, 1998.

[54] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278, 2011.

[55] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 55–66, 2003.

[56] Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 225–236, 2010.

[57] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40(2-3):171–188, 2001.

[58] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proc. of the Intl. Symp. on Software Testing and Analysis (ISSTA)*, pages 196–206, 2007.

[59] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: scalable address spaces for multithreaded applications. In *Proc. of the European Conf. on Computer Systems (EuroSys)*, pages 211–224, 2013.

[60] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proc. of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 128–137, 1994.

[61] Jason Cong, Zhenman Fang, Michael Gill, and Glenn Reinman. PARADE: a cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration. In *Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD)*, pages 380–387, 2015.

[62] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. Accelerator-rich architectures: Opportunities and progresses. In *Proc. of the Design Automation Conference (DAC)*, pages 1–6, 2014.

[63] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Architecture support for accelerator-rich CMPs. In *Proc. of the Design Automation Conference (DAC)*, pages 843–849, 2012.

[64] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. CHARM: a composable heterogeneous accelerator-rich microprocessor. In *Proc. of the Intl. Symp. on Low Power Electronics and Design (ISLPED)*, pages 379–384, 2012.

[65] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Chunyue Liu, and Glenn Reinman. BiN: a buffer-in-NUCA scheme for accelerator-rich CMPs. In *Proc. of the Intl. Symp. on Low Power Electronics and Design (ISLPED)*, pages 225–230, 2012.

[66] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. Cross-ISA machine emulation for multicores. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 210–220, 2017.

[67] Emilio G. Cota and Luca P. Carloni. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation. In *Proc. of the Intl. Conf. on Virtual Execution Environments (VEE)*, pages 74–87, 2019.

[68] Emilio G. Cota, Paolo Mantovani, and Luca P. Carloni. Exploiting private local memories to reduce the opportunity cost of accelerator integration. In *Proc. of the Intl. Conf. on Supercomputing*, pages 27:1–27:12, 2016.

[69] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *Proc. of the Design Automation Conference (DAC)*, pages 202:1–6, 2015.

[70] Emilio G. Cota, Paolo Mantovani, Michele Petracca, Mario R. Casu, and Luca P. Carloni. Accelerator memory reuse in the dark silicon era. *IEEE Comput. Archit. Lett.*, 13(1):9–12, 2014.

[71] Louise H Crockett, Ross A Elliot, Martin A Enderwitz, and Robert W Stewart. *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media, 2014.

[72] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM virtualization: performance and architectural implications. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 304–316, 2016.

[73] Christoffer Dall and Jason Nieh. KVM/ARM: the design and implementation of the Linux ARM hypervisor. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 333–348, 2014.

[74] Andrew Danowitz, Kyle Kelley, James Mao, John P Stevenson, and Mark Horowitz. CPU DB: recording microprocessor history. *Communications of the ACM*, 55(4):55–63, 2012.

[75] Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Optimizing indirect branches in dynamic binary translators. *ACM Trans. on Architecture and Code Optimization (TACO)*, 13(1):7, 2016.

[76] Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Low overhead dynamic binary translation on ARM. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 333–346, 2017.

[77] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 631–644, 2015.

[78] Arup De, Maya Gokhale, Rajesh Gupta, and Steven Swanson. Minerva: Accelerating data analysis in next-generation SSDs. In *Proc. of the Intl. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16, 2013.

[79] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta code morphing™ software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 15–24, 2003.

[80] John Demme. *Overcoming the Intuition Wall: Measurement and Analysis in Computer Architecture.* PhD thesis, Columbia University, 2014.

[81] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. MemScale: Active low-power modes for main memory. In *Proc. of*

the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 225–238, 2011.

[82] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[83] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 266–277, 2001.

[84] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Trans. on Parallel and Distributed Systems*, 23:375–382, 2012.

[85] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic shared memory multiprocessing. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, 2009.

[86] Alessandro Di Federico and Giovanni Agosta. A jump-target identification method for multi-architecture static binary translation. In *Proc. of the Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10, 2016.

[87] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. PQEMU: A parallel system emulator based on QEMU. In *Proc. of the Intl. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 276–283, 2011.

[88] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Proc. of Program Protection and Reverse Engineering Workshop*, pages 4:1–4:11, 2015.

[89] Pavel Dovgalyuk, Natalia Fursova, Ivan Vasiliev, and Vladimir Makarov. Qemu-based framework for non-intrusive virtual machine instrumentation and introspection. In *Proc. of the Joint Meeting on Foundations of Software Engineering and the ACM SIGSOFT Symp. on the Foundations of Software Engineering*, pages 944–948, 2017.

[90] Kemal Ebcioğlu and Erik R. Altman. DAISY: dynamic compilation for 100% architectural compatibility. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 26–37, 1997.

[91] Lieven Eeckhout. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture*, 5(1):1–145, 2010.

[92] Lieven Eeckhout. Is Moore's law slowing down? What's next? *IEEE Micro*, 37(4):4–5, 2017.

[93] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 365–376, 2011.

[94] Carlos Flores Fajardo, Zhen Fang, Ravi Iyer, German Fabila Garcia, Seung Eun Lee, and Li Zhao. Buffer-integrated-cache: a cost-effective SRAM architecture for handheld and embedded platforms. In *Proc. of the Design Automation Conference (DAC)*, pages 966–971, 2011.

[95] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. of the USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 371–384, 2013.

[96] Antoine Faravelon, Olivier Gruber, and Frédéric Pétrot. Optimizing memory access performance using hardware assisted virtualization in retargetable dynamic binary translation. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 40–46, 2017.

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

[98] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering*, 2011.

[99] Sheng-Yu Fu, Ding-Yong Hong, Jan-Jan Wu, Pangfeng Liu, and Wei-Chung Hsu. SIMD code translation in an enhanced HQEMU. In *Proc. of the Intl. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 507–514, 2015.

[100] Yaosheng Fu and David Wentzlaff. Prime: A parallel and distributed simulator for thousand-core chips. In *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 116–125, 2014.

[101] Richard M Fujimoto. *Parallel and distributed simulation systems*, volume 300. Wiley New York, 2000.

[102] Davide Giri, Paolo Mantovani, and Luca P Carloni. Accelerators and coherence: An SoC perspective. *IEEE Micro*, 38(6):36–45, 2018.

[103] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In *Proc. of the Intl. Symp. on Hardware/Software Codesign ans System Synthesis (CODES+ISSS)*, pages 71–80, 2009.

[104] Yu-Chuan Guo, Wuu Yang, Jiunn-Yeu Chen, and Jenq-Kuen Lee. Translating the ARM Neon and VFP instructions in a binary translator. *Softw. Pract. Exper.*, 46(12):1591–1615, 2016.

[105] Anthony Gutierrez, Joseph Pusdesris, Ronald G Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D Emmons, Mitchell Hayenga, and Nigel Paver. Sources of error in full-system simulation. In *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 13–22, 2014.

[106] Nabil Hallou, Erven Rohou, Philippe Clauss, and Alain Ketterlin. Dynamic re-vectorization of binary code. In *Prof. of the Intl. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 228–237, 2015.

[107] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 1–13, 2016.

[108] 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 *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 37–47, 2010.

[109] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 184–195, 2009.

[110] John R. Hauser. The softfloat and testfloat packages, accessed Feb. 24, 2019.

[111] Kim Hazelwood. Dynamic binary modification: Tools, techniques, and applications. *Synthesis Lectures on Computer Architecture*, 6(2):1–81, 2011.

[112] Kim Hazelwood and Michael D Smith. Generational cache management of code traces in dynamic optimization systems. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, page 169, 2003.

[113] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *Proc. of the Intl. Symp. on Software Testing and Analysis (ISSTA)*, pages 248–258, 2014.

[114] Maurice Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, 1991.

[115] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[116] Jason D. Hiser, Daniel Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 61–73, 2007.

[117] Ding-Yong Hong, Chun-Chen Hsu, Cheng-Yi Chou, Wei-Chung Hsu, Pangfeng Liu, and Jan-Jan Wu. Optimizing control transfer and memory virtualization in full system emulators. *ACM Trans. on Architecture and Code Optimization (TACO)*, 12(4):47:1–47:24, 2015.

[118] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 104–113, 2012.

[119] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proc. of Principles and Practice of Parallel Programming (PPoPP)*, pages 267–276, 2011.

[120] Joel Hruska. Nvidia deeply unhappy with TSMC, claims 20nm essentially worthless. *ExtremeTech*, March 2012.

[121] Jing Huang, Yuanjie Huang, Olivier Temam, Paolo Ienne, Yunji Chen, and Chengyong Wu. A low-cost memory interface for high-throughput accelerators. In *Proc. of the Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 11:1–11:10, 2014.

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

[123] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation. *Web Copy*, 2010.

[124] Aamer Jaleel, Robert S Cohn, Chi-Keung Luk, and Bruce Jacob. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *Proc. of Workshop on Modeling, Benchmarking and Simulation (MoBS)*, pages 28–36, 2008.

[125] Xiao-Wu Jiang, Xiang-Lan Chen, Huang Wang, and Hua-Ping Chen. A parallel full-system emulator for RISC architure host. In *Advances in Computer Science and its Applications*, pages 1045–1052. 2014.

[126] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy

Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 1–12, 2017.

[127] Vinod Kathail, James Hwang, Welson Sun, Yogesh Chobe, Tom Shui, and Jorge Carrillo. SDSoC: a higher-level programming environment for Zynq SoC and Ultrascale+ MPSoC. In *Proc. of the Intl. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 4–4, 2016.

[128] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011.

[129] John H Kelm and Steven S Lumetta. HybridOS: Runtime support for reconfigurable accelerators. In *Proc. of the Intl. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 212–221, 2008.

[130] Angelos D Keromytis, Roxana Geambasu, Simha Sethumadhavan, Salvatore J Stolfo, Junfeng Yang, Azzedine Benameur, Marc Dacier, Matthew Elder, Darrell Kienzle, and Angelos Stavrou. The MEERKATS cloud security architecture. In *Intl. Conf. Distributed Computing Systems (ICDCSW) – Workshops*, pages 446–450, 2012.

[131] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. Dynamic compilation of data-parallel kernels for vector processors. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 23–32, 2012.

[132] Chad D. Kersey, Arun Rodrigues, and Sudhakar Yalamanchili. A universal parallel front-end for execution driven microarchitecture simulation. In *Proc. of Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*, pages 25–32, 2012.

[133] Asad Khan, Weiqiang Ma, Chris Wolf, and Bengt Werner. Multi-threaded simics systemc virtual platform. In *Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD)*, pages 373–379, 2015.

[134] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. Moonwalk: NRE optimization in ASIC clouds. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 511–526, 2017.

[135] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 211–222, 2002.

[136] Jinpyo Kim, Wei-Chung Hsu, and Pen-Chung Yew. COBRA: an adaptive runtime binary optimization framework for multithreaded applications. In *Proc. of the Intl. Conf. on Parallel Processing (ICPP)*, pages 25–25, 2007.

[137] Martha Mercaldi Kim, Mojtaba Mehrara, Mark Oskin, and Todd Austin. Architectural implications of brick and mortar silicon manufacturing. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, volume 35, pages 244–253, 2007.

[138] Vladimir Kiriansky, Derek Bruening, and Saman P Amarasinghe. Secure execution via program shepherding. In *Proc. of the USENIX Security Symposium*, volume 92, page 84, 2002.

[139] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Proc. of the Ottawa Linux Symposium*, volume 1, pages 225–230, 2007.

[140] Reid Kleckner. *Optimization of naïve dynamic binary instrumentation Tools*. PhD thesis, Massachusetts Institute of Technology, 2011.

[141] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals

for in-memory databases. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 468–479, 2013.

[142] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: a language and compiler for application accelerators. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 296–311, 2018.

[143] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. Stash: Have your scratchpad and cache it too. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 707–719, 2015.

[144] Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua. Automatic parallelization in a binary rewriter. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 547–557, 2010.

[145] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. Hardware acceleration in the IBM PowerEN processor: Architecture and performance. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 389–400, 2012.

[146] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[147] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 26(2):203–215, 2007.

[148] George Kurian, Srinivas Devadas, and Omer Khan. Locality-aware data replication in the last-level cache. In *Proc. of the Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2014.

[149] Christoph Lameter. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*, 2005.

[150] Robert E. Lantz. Fast functional simulation with parallel Embra. In *Proc. of Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2008.

[151] Dairsie Latimer. In Intel's arduous journey to 10 nm, Moore's law comes up short. *Top500*, August 2018.

[152] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, page 75, 2004.

[153] Hung Q Le, Guy L Guthrie, DE Williams, Maged M Michael, BG Frey, William J Starke, Cathy May, Rei Odaira, and Takuya Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8–1, 2015.

[154] Oded Lempel. 2nd generation intel core processor family: Intel core i7, i5 and i3. In *Hot Chips*, 2011.

[155] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 469–480, 2009.

[156] Joseph I Lieberman. White paper: National security aspects of the global migration of the US semiconductor industry. *Washington: Office of Senator Joseph I. Lieberman, Ranking Member, United States Senate Armed Services Committee*, 2003.

[157] Yu-Ping Liu, Ding-Yong Hong, Jan-Jan Wu, Sheng-Yu Fu, and Wei-Chung Hsu. Exploiting asymmetric SIMD register configurations in ARM-to-x86 dynamic binary translation. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 343–355, 2017.

[158] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux scheduler: A decade of wasted cores. In *Proc. of the European Conf. on Computer Systems (EuroSys)*, pages 1:1–1:16, 2016.

[159] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, page 180, 2003.

[160] 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 *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[161] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. ArMOR: defending against memory consistency model mismatches in heterogeneous architectures. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 388–400, 2015.

[162] Michael J. Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. The accelerator store: A shared memory framework for accelerator-based systems. *ACM Trans. Archit. Code Optim.*, 8(4):48:1–48:22, 2012.

[163] Yi-Hong Lyu, Ding-Yong Hong, Tai-Yi Wu, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. DBILL: an efficient and retargetable dynamic binary instrumentation framework using LLVM backend. In *Proc. of the Intl. Conf. on Virtual Execution Environments (VEE)*, pages 141–152, 2014.

[164] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. ASIC clouds: Specializing the datacenter. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 178–190, 2016.

[165] Peter Magnusson and Bengt Werner. Efficient memory simulation in SimICS. In *Proceedings of Simulation Symposium*, pages 62–73, 1995.

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

[167] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: A modular reconfigurable architecture. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 161–171, 2000.

[168] Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P Carloni. High-level synthesis of accelerators in embedded scalable platforms. In *Proc. of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 204–211, 2016.

[169] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proc. of the European Conf. on Computer Systems (EuroSys)*, pages 183–196, 2012.

[170] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[171] Milo MK Martin, Mark D Hill, and Daniel J Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, 2012.

[172] Milo MK 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, 2005.

[173] Paul E McKenney, Silas Boyd-Wickizer, and Jonathan Walpole. RCU usage in the linux kernel: one decade later. Technical report, 2013.

[174] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[175] Charith Mendis, Ajay Jain, Paras Jain, and Saman Amarasinghe. Revec: program rejuvenation through revectorization. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, 2019.

[176] A. Mericas, N. Peleg, L. Pesantez, S. B. Purushotham, P. Oehler, C. A. Anderson, B. A. King-Smith, M. Anand, J. A. Arnold, B. Rogers, L. Maurice, and K. Vu. IBM POWER8 performance features and evaluation. *IBM Journal of Research and Development*, 59(1):6–1, 2015.

[177] Maged M Michael. ABA prevention using single-word instructions. *IBM Research Division, RC23089 (W0401-136), Tech. Rep*, 2004.

[178] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[179] Pierre Michaud. Demystifying multicore throughput metrics. *Computer Architecture Letters*, 12(2):63–66, July 2013.

[180] Luc Michel, Nicolas Fournel, and Frédéric Pétrot. Speeding-up SIMD instructions dynamic binary translation in embedded processor simulation. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, pages 1–4, 2011.

[181] 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 *Proc. of the Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[182] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.

[183] Gordon E. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[184] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proc. of the USENIX Security Symposium*, pages 18:1–18:18, 2007.

[185] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, 2007.

[186] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 284–295, 2005.

[187] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.

[188] Anh Quynh Nguyen and Hoang Vu Dang. Unicorn: next generation CPU emulator framework. In *Black Hat USA*, 2015.

[189] Tony Nowatzki, Vinay Gangadhar, Karthikeyan Sankaralingam, and Greg Wright. Pushing the limits of accelerator efficiency while retaining programmability. In *Proc. of the Intl. Symp. on High-Performance Computer Architecture (HPCA)*, 2016.

[190] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In *Proc. of the Intl. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 5–14, 2017.

[191] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy efficient architecture for graph analytics accelerators. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 166–177, 2016.

[192] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Proc. of the Design Automation Conference (DAC)*, pages 1050–1055, 2011.

[193] Christian Pilato, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. System-level memory optimization for high-level synthesis of component-based SoCs. In *Proc. of the Intl. Symp. on Hardware/Software Codesign ans System Synthesis (CODES+ISSS)*, pages 1–10, 2014.

[194] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking SIMD vectorization for in-memory databases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD)*, pages 1493–1508, 2015.

[195] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[196] Emerson W Pugh. *Building IBM: shaping an industry and its technology.* MIT press, 1995.

[197] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 13–24, 2014.

[198] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. Convolution engine: Balancing efficiency & flexibility in specialized computing. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 24–35, 2013.

[199] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 135–148, 2006.

[200] Nguyen Anh Quynh. Capstone: Next-gen disassembly framework. *Black Hat USA*, 2014.

[201] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite: benchmarks for accelerator design and customized architectures. In *Proc. of the Intl. Symp. on Workload Characterization (IISWC)*, pages 110–119, 2014.

[202] Vijay Janapa Reddi, Dan Connors, Robert Cohn, and Michael D Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 74–88, 2007.

[203] Pengju Ren, Mieszko Lis, Myong Hyon Cho, Keun Sup Shim, Christopher W Fletcher, Omer Khan, Nanning Zheng, and Srinivas Devadas. HORNET: a cycle-level multicore simulator. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 31(6):890–903, 2012.

[204] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net

[205] Mendel Rosenblum, Stephen A Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):34–43, 1995.

[206] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16 –19, Jan.-June 2011.

[207] Richard Sampson and Thomas F. Wenisch. ZCache skew-ered. In *Proc. of the Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2011.

[208] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling ways and associativity. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 187–198, 2010.

[209] Daniel Sanchez and Christos Kozyrakis. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 475–486, 2013.

[210] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 175–186, 2011.

[211] Eric Schnarr and James R. Larus. Fast out-of-order processor simulation using memoization. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 283–294, 1998.

[212] David Schneider. Deeper and cheaper machine learning [top tech 2017]. *IEEE Spectrum*, 54(1):42–43, 2017.

[213] K. Scott, N. Kumar, B. R. Childers, J. W. Davidson, and M. L. Soffa. Overhead reduction techniques for software dynamic translation. In *Proc. of the Intl. Parallel and Distributed Processing Symposium (IPDPS)*, page 200, 2004.

[214] Kevin Scott, Naveen Kumar, Siva Velusamy, Bruce Childers, Jack W Davidson, and Mary Lou Soffa. Retargetable and reconfigurable software dynamic translation. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 36–47, 2003.

[215] André Seznec. Bank-interleaved cache or memory indexing does not require euclidean division. In *Proc. of the 11th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2015.

[216] Yakun Sophia Shao and David Brooks. ISA-independent workload characterization and its implications for specialized architectures. In *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 245–255, 2013.

[217] Yakun Sophia Shao and David Brooks. Research infrastructures for hardware accelerators. *Synthesis Lectures on Computer Architecture*, 10(4):1–99, 2015.

[218] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. Co-designing accelerators and SoC interfaces using gem5-Aladdin. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 1–12, 2016.

[219] Tom Simonite. Intel puts the brakes on Moore's law. *MIT Technology Review*, March 2016.

[220] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proc. of the Intl. Conf. on Information Systems Security (ICISS)*, pages 1–25, 2008.

[221] Gabriel Southern and Jose Renau. Deconstructing PARSEC scalability. In *Proc. of the Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2015.

[222] Tom Spink, Harry Wagstaff, and Björn Franke. Hardware-accelerated cross-architecture full-system virtualization. *ACM Trans. on Architecture and Code Optimization (TACO)*, 13(4):36, 2016.

[223] Swaroop Sridhar, Jonathan S Shapiro, Eric Northup, and Prashanth P Bungale. HDTrans: an open source, low-level dynamic instrumentation system. In *Proc. of the Intl. Conf. on Virtual Execution Environments (VEE)*, pages 175–185, 2006.

[224] Sadagopan Srinivasan, Li Zhao, Ramesh Illikkal, and Ravishankar Iyer. Efficient interaction between OS and architecture in heterogeneous platforms. *ACM SIGOPS Operating Systems Review*, 45(1):62–72, 2011.

[225] J Stuecheli, B Blaner, CR Johns, and MS Siegel. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[226] Chen Sun, C-HO Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. DSENT - A tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proc. of the Intl. Symp. on Networks on Chip (NoCS)*, pages 201–210, 2012.

[227] Dean Takahashi. Intel: Moore's law isn't slowing down. *VentureBeat*, March 2017.

[228] Michael B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *Proc. of the Design Automation Conference (DAC)*, pages 1131–1136, 2012.

[229] Michael Bedford Taylor. The evolution of bitcoin hardware. *Computer*, 50(9):58–66, 2017.

[230] X. Tong and A. Moshovos. Qtrace: a framework for customizable full system instrumentation. In *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 245–255, 2015.

[231] Xin Tong, Toshihiko Koju, Motohiro Kawahito, and Andreas Moshovos. Optimizing memory translation emulation in full system emulators. *ACM Trans. on Architecture and Code Optimization (TACO)*, 11(4):60, 2015.

[232] Josh Triplett, Paul E McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proc. of the USENIX Annual Technical Conference (ATC)*, 2011.

[233] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–218, 2010.

[234] Huy Vo, Yunsup Lee, Andrew Waterman, and Krste Asanovic. A case for OS-friendly hardware accelerators. In *Proc. of the Annual Workshop on the Interaction amongst Virtualization, Operating Systems and Computer Architecture (WIVOSCA)*, 2013.

[235] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Enhancing cross-ISA DBT through automatically learned translation rules. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–97, 2018.

[236] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. A general persistent code caching framework for dynamic binary translation (DBT). In *Proc. of the USENIX Annual Technical Conference (ATC)*, pages 591–603, 2016.

[237] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. COREMU: A scalable and portable parallel full-system emulator. In *Proc. of Principles and Practice of Parallel Programming (PPoPP)*, pages 213–222, 2011.

[238] Zhe Wang, Jianjun Li, Chenggang Wu, Dongyan Yang, Zhenjiang Wang, Wei-Chung Hsu, Bin Li, and Yong Guan. HSPT: Practical implementation and efficient management of embedded shadow page tables for cross-ISA system virtual machines. In *Proc. of the Intl. Conf. on Virtual Execution Environments (VEE)*, pages 53–64, 2015.

[239] Vincent M Weaver and Sally A McKee. Are cycle accurate simulations a waste of time? In *Proc. of the Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, pages 40–53, 2008.

[240] Ryan Whelan, Tim Leek, and David Kaeli. Architecture-independent dynamic information flow tracking. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, pages 144–163, 2013.

[241] Emmett Witchel and Mendel Rosenblum. Embra: fast and flexible machine simulation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 24, pages 68–79, 1996.

[242] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: the architecture and design of a database processing unit. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 255–268, 2014.

[243] Qiang Wu, Margaret Martonosi, Douglas W Clark, Vijay Janapa Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 271–282, 2005.

[244] Efe Yardimci and Michael Franz. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *Proc. of the Conf. on Computing Frontiers*, pages 127–138, 2006.

[245] Reza Yazdani, Albert Segura, Jose-Maria Arnau, and Antonio Gonzalez. An ultra low-power hardware accelerator for automatic speech recognition. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 1–12, 2016.

[246] Matt T Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 23–34, 2007.

[247] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. PEMU: A Pin highly compatible out-of-VM dynamic binary instrumentation framework. In *Proc. of the Intl. Conf. on Virtual Execution Environments (VEE)*, pages 147–160, 2015.

[248] Weifeng Zhang, Brad Calder, and Dean M Tullsen. An event-driven multi-threaded dynamic optimization framework. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 87–98, 2005.

[249] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. HERMES: A fast cross-ISA binary translator with post-optimization. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, pages 246–256, 2015.

Appendix

Implementation Adoption

Contributions to Upstream QEMU

At the time of this writing, the main intellectual contributions implemented in Pico and Qelt (Chapters 3 and 4, respectively) are part of QEMU[1]. The only exception is Qelt's instrumentation layer, which is currently under review by the QEMU community.

QEMU v2.7, released in September 2016, includes (1) Pico's improved hashing for the TB block hash table (commit `42bd32287`) and (2) Pico's implementation and use of QHT for scalable TB lookups (`909eaac9`, `518615c6`). QEMU v2.8 includes: (1) the use of the host's atomic instructions to emulate the guest's atomics (e.g., x86 in `37b995f6`, ARM in `354161b3`, Aarch64 in `1dd089d0`; merge commit `5929d7e8`), and (2) the necessary work to safely support multi-threaded execution (`53f5ed95`, `3359baad`; merge commit `c640f284`). Qelt's indirect branch improvements (`a0d4aac7`) are in v2.10, parallel code generation (`33836a73`) is in v3.0, and Qelt's FP emulation enhancements (`ec3c927f`) as well as dynamic TLB sizing (`3a183e33`) are in v4.0, released in April 2019.

[1]http://www.qemu.org/

Lessons Learned

The integration of the above features totals more than 300 non-merge commits in QEMU. The required integration effort involved a substantial time investment, yet came with two long-term benefits: (1) it ensures that the features will be maintained, which maximizes the number of eventual users, and (2) the resulting code quality is higher, since the code went through several review iterations.

Navigating the upstream process of a popular open-source project is not trivial. Based on the lessons learned from our interactions with the QEMU community, we now reflect on four principles that an external contributor would do well to follow.

Trust. Maintainers are responsible for maintaining code after it is merged, and are therefore judicious in what code they merge in the first place. Proposing novel features without a clear use case is guaranteed to fail; maintainers will inevitable ask the question *"What is it what you are trying to do?"* Answering this question with a practical use case and with technical depth is a mandatory step towards building some trust from the maintainers. Only after this question has a satisfactory answer a contributor should move on to a submission via the project's contribution mechanisms (e.g. QEMU's submission guide), to then clear the bar imposed by the maintainers in terms of code clarity and efficiency.

Empathy. Project maintainers are deeply knowledgeable and—perhaps as a result—extremely busy. They not only write code, they also review code submissions from others and guide the direction of the project. Thus, a contributor should use the maintainers' time wisely and always remain patient when waiting for feedback. A contributor should address all concerns raised by code reviewers in a timely fashion, and be prepared to expand the scope of a contribution if so required (for example, by adding tests when modifying some code, even though no tests for that code existed previously).

Engagement. Regardless of the medium of communication used, a mature open-source project is inevitably a team effort, and the more a contributor can contribute to that collective effort, the easier it will be for the contributor to get their changes merged. A simple way to engage with the community is for the contributor to also review other people's code submissions, as well as monitor the bug tracking system; this increases good will from other members of the community, as well as improves the contributor's understanding about potentially unfamiliar parts of the code base. Another productive way of engaging with the community is to attend one of their regular in-person events, e.g. KVM Forum in case of QEMU, which is a yearly gathering of QEMU developers.

Persistence. It is not unusual to go through more than a dozen review iterations for a submission, particularly when landing a large feature. This can be discouraging, but persistence has a reward: the resulting (merged) code will be of higher quality. Being quick in incorporating review feedback is also strongly advised; reviewers will have a fresher state when submitting an improved version, which will likely result in a quicker turnaround.