## Advancing System-Level Analysis and Design of Specialized Architectures

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

<table>
<thead>
<tr>
<th>Citation</th>
<th>Xi, Likun. 2018. Advancing System-Level Analysis and Design of Specialized Architectures. Doctoral dissertation, Harvard University, Graduate School of Arts &amp; Sciences.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Citable link</td>
<td><a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:41121197">http://nrs.harvard.edu/urn-3:HUL.InstRepos:41121197</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
Advancing System-Level Analysis and Design of Specialized Architectures

A DISSERTATION PRESENTED
BY
Likun (Sam) Xi
TO
The School of Engineering and Applied Sciences

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy
IN THE SUBJECT OF
Computer Science

Harvard University
Cambridge, Massachusetts
July 2018
Advancing System-Level Analysis and Design of Specialized Architectures

Abstract

Over the course of the past decade, computation has increasingly spread to the cloud and mobile devices. With the growing computation demands placed by contemporary cloud and mobile workloads, architects have increasingly turned to hardware specialization. Once the niche of standardized computation like video decoding and audio processing, hardware accelerators have now expanded into many more fields. Fueled by the recent explosion of demand for deep neural networks, a huge amount of effort has been poured into advancing state-of-the-art accelerator architectures and designs.

However, current research into specialized hardware often overlooks evaluating the complete system, and this can often lead to non-optimal designs. For example, a large fraction of the total power consumed by a system-on-chip (SoC) is actually due to CPUs, but the accuracy of widely used CPU power models have not been thoroughly validated. Second, while we know how to design efficient accelerators in isolation, we have less understanding of how SoC integration impacts their performance and power. In addition, we have not explored how we can leverage SoC-accelerator interfaces to improve efficiency. Finally, architects have mostly explored “deep” acceleration, which focuses on compute-heavy workloads with hot functions, but we have largely ignored “broad” acceleration, which aims to accelerate common low-level routines present across a diverse set of workloads.
This dissertation presents the case for a holistic approach to accelerator design that accounts for the surrounding system’s constraints, both for “deep” acceleration and “broad” acceleration. First, it presents a comprehensive validation of McPAT, a widely used CPU power model, with a quantitative analysis of its sources of error. Second, it presents gem5-Aladdin, an complete SoC simulator that can model complex specialized SoCs and can run end-to-end accelerated workloads without the need to write any RTL. Third, this dissertation shows how considerations of system-level effects and SoC interfacing during accelerator design can dramatically improve its overall efficiency, with a deep dive into accelerating deep neural networks and vision pipelines. Finally, it leverages recent work in datacenter system-wide profiling to make a case for broad acceleration. It presents the design of an accelerator for dynamic memory allocation, a widely used programming paradigm that accounts for a significant fraction of total CPU cycles in a major cloud provider’s datacenters.

The work presented in this dissertation identifies both challenges and opportunities for extracting maximum performance from acceleration at the system level, both for traditional deep acceleration and broad acceleration in the cloud. We hope it will stimulate more interest and spur further research and development for holistic accelerator design.
Contents

1 INTRODUCTION .............................................. 1
  1.1 Thesis Contributions ............................... 4

2 BACKGROUND AND RELATED WORK ............... 8
  2.1 Architectural Power Modeling ..................... 8
  2.2 Accelerator Design Methodology ................ 11
  2.3 SoC-Accelerator Co-Design ....................... 13
  2.4 Broad Acceleration ............................... 17

3 QUANTIFYING SOURCES OF ERROR IN McPAT .... 18
  3.1 Introduction ....................................... 19
  3.2 Power Modeling Approaches ..................... 20
  3.3 Assessment Methodology .......................... 21
  3.4 Assessment Results ............................... 24
  3.5 Case Studies ....................................... 38
  3.6 Discussion and Guidelines ....................... 44
  3.7 Next Steps ......................................... 48

4 ENABLING SoC-Accelerator Co-Design using gem5-Aladdin 49
  4.1 Introduction ....................................... 50
  4.2 Motivation and Background ...................... 51
  4.3 Modeling infrastructure ......................... 55
# List of Figures

1.0.1 The number of specialized IP blocks on Apple SoCs over eight generations. Credit: Yu Wang and Sophia Shao, Harvard.  

3.3.1 POWER7 chiplet simplified floorplan. The decimal-floating unit (DFU) is omitted, and the wraparound L3 cache is split into two parts.  

3.4.1 The subset of DPM total POWER7 area and power that is represented by McPAT’s models.  

3.4.2 Power cumulative distribution function of each unit. Power from macros in the McPAT subset are in light green, and those not in the subset are in dark green. Each bar represents power consumed by a single macro plus the power from all macros consuming less power than it. Power is normalized to the total of the unit.  

3.4.3 Core power for 21 different benchmarks, normalized to DPM subset power. Each successive revision improves accuracy and thus brings McPAT’s estimates closer to DPM subset power. Note that the selected workloads from SPEC2006 have about the same power variance as those from SPEC2000.  

3.4.4 Core area, normalized to DPM subset. Each revision brings estimates closer to DPM subset area.
3.4.5 Core power breakdown by unit, normalized to DPM subset power. Power error relationships across the models are largely consistent regardless of the benchmark. .......................................................... 30
3.4.6 Instruction fetch unit area breakdown. ................................................. 31
3.4.7 Instruction fetch unit power breakdown. ............................................. 32
3.4.8 Instruction sequencing unit area breakdown. ...................................... 33
3.4.9 Instruction sequencing unit power breakdown. .................................... 33
3.4.10 Load-store unit area breakdown. ...................................................... 34
3.4.11 Load-store unit power breakdown. .................................................. 35
3.4.12 Fixed-point unit breakdowns. .......................................................... 36
3.4.13 Vector scalar unit breakdowns. See Section 3.4.1 for an explanation of VRF power on mcf_2k. ................................................................. 37
3.5.1 Steady state thermal distribution across the POWER7 chip in °C. MRo shows much greater variance across workloads than MRz and DPM, but MRo’s overall chip temperature is more accurate. 39
3.5.2 Overall chip temperature error with respect to DPM. Note that MRz error is always negative, but absolute value error is easier to visually compare. ................................................................. 40
3.5.3 Snapshot of a voltage noise trace for the three power models from gcc_2k. MRo shows considerably greater inductive noise than DPM, whereas MRz has much less error. .................................. 42
3.5.4 Distribution of voltage noise for all benchmarks and all power models with standard box and whisker heights. Consistent with Figure 3.5.3, MRz is a significant improvement over MRo. ...... 43
3.6.1 Covariance and ratio between subset and nonsubset components on four SPEC2006 benchmarks. The FXU and VSU were excluded because their subsets cover over 80% of the logic. .... 47

4.2.1 Design space exploration for stencil3d for both isolated and co-designed cases. ................................................................. 52
4.2.2 Data movement overheads on MachSuite. ........................................ 53
4.2.3 An example SoC that can be modeled using gem5-Aladdin. The table on the right shows the set of design parameters that we swept in this work and their values; this is just a small subset of what can be configured. ............................. 54
4.3.1 Error between Zedboard and gem5-Aladdin cycles. .............. 58
4.4.1 A demonstration of the DMA latency reduction techniques. ....... 62
4.4.2 Cumulatively applying each technique reduces the additional cycles spent on DMA, with some benchmarks seeing more benefit than others. After applying all techniques, increasing parallelism through loop unrolling reduces compute cycles until near-complete overlap is achieved, causing performance to saturate. 65
4.4.3 Effect of datapath parallelism on cache-based accelerator performance. ................................................................. 67
4.4.4 Power-performance Pareto curves for DMA- and cache-based accelerators. EDP optimal design points are shown as stars. Benchmarks are ordered left-to-right, top-down by preference for a DMA-based vs. a cache-based memory system. ...................... 68
4.5.1 Comparison of accelerator microarchitectural parameters across four design scenarios. The vertices of the Kiviat plots represent the number of datapath lanes, SRAM sizes, and local memory bandwidth, normalized to the isolated optimal design, shown on the upper-left corner, for each benchmark. ............................. 70
4.5.2 EDP improvement of co-designed accelerators in different scenarios, normalized to EDP of isolated designs. The design parameters of each optimal design point are illustrated in Figure 4.5.1. 74
5.3.1 The SoC platform used in this chapter’s experiments. The accelerators can use either DMA or ACP to access host memory. .... 84
5.3.2 Dataflow implemented by the accelerator. Apart from syntax, this is nearly the actual C code in SMAUG. ............................ 85
5.4.1 Overview of SMAUG execution flow. ............................... 92
5.4.2 Work tiling strategy used in this chapter. ................. 95
5.5.1 Performance and energy scaling behavior of the DNN accelerator with increasing local SRAM sizes (32KB, 48KB, 64KB, 128KB) for each integration scenario. Data is normalized to the baseline DMA scenario with 32KB SRAM. ................. 97
5.5.2 ACP decreases total DRAM traffic due to low overall miss rates in the LLC. ................................. 99
5.5.3 Each plot shows the relative per-program execution time when running two programs simultaneously: one DNN and one ISP accelerator. For example, the multiprogram mix of CF-VGG and spmv-cr is shown in the upper right corner. Within each integration scenario (A, B, C), each program's performance is compared to itself when running in isolation. ................. 102
5.5.4 Each plot shows relative per-program execution time of a four-program mixture: one DNN and all three ISP accelerators. For example, the top row individually shows DNN, stenc1l, fft, and spmv performance when all four run concurrently, compared to when each of them run in isolation. ................. 104
5.5.5 Each plot shows relative per-program execution time when running one DNN and one Halide program concurrently. ................. 105
5.5.6 Halide L2 cache miss rates rise significantly when running concurrently with a DNN compared to running in isolation. .......... 105
6.2.1 The costs of hits and misses in several allocation pools in TCMalloc vary by orders of magnitude. ......................... 114
6.3.1 The steps that an allocation requests goes through. The colored boxes represent the major operations on the fast path, which we aim to optimize. ......................... 115
6.3.2 Majority of time in malloc in CPU2006 is spent on calls taking less than 100 clock cycles. ......................... 117
6.3.3 Time spent in the three main components of the fast path accounts for ≈ 50% of cycles. ........................... 118
6.3.4 Size class lookup function. ................................. 119
6.3.5 Many benchmarks use a very small number of size classes. .... 120
6.3.6 Critical memory accesses on a free list push/pop. ............... 121
6.3.7 A malloc cache with example values. The cache is searched by
first an associative lookup over requested size and later by size
class. It stores the corresponding size class, and the first two free-
list elements for that size class. .................................. 122
6.3.8 Pseudocode for size class instructions. ......................... 123
6.4.1 Integration of size class instructions in an allocator. .......... 125
6.4.2 Pseudocode for linked list instructions. ......................... 126
6.4.3 Integration of linked list instructions in an allocator. .......... 127
6.6.1 Improvement of time spent in the allocator. ...................... 133
6.6.2 Improvement in time spent on malloc() calls (both fast and
slow paths). ......................................................... 134
6.6.3 Xapian sees a significant improvement on already-fast calls. .... 136
6.6.4 Xalan benefits both from latency reduction and cache isolation. 137
6.6.5 Effect of cache size on malloc speedup. ......................... 137
6.6.6 Fraction of time spent in the allocator. ......................... 139
Acknowledgments

These past five years of my Ph.D. have been the most transformative time of my life. I’m incredibly grateful to all the people that I’ve had the good fortune to work with, learn from, and grow with.

First, thanks so much to my advisors David Brooks and Gu-Yeon Wei. I remember, as if it were yesterday, when I was a graduating senior at Duke, agonizing over where to go for graduate school. I couldn’t have made a better choice to come to Harvard and join this dynamic group. They were always happy to meet and offer me advice and guidance at any time, and over the years they’ve taught me how to think critically, collaborate with other students, and shaped me into a competent computer architect and researcher. The passion they have for their work often takes me aback. I still remember David’s first words right after telling me my first paper was accepted: “Okay, now work hard on ISCA 🤟”. They knew how to drive me to work and also trust me as an adult to manage my own time, and I could not have asked for a better set of advisors.

I also want to thank to Stratos Idreos for working with me and serving on my qualifying and dissertation committee. I knew nothing about databases prior to taking his class, but the class and our research collaborations were really fun and rewarding. Thanks Greg Morrissett for serving on my quals committee. Finally, I couldn’t forget Benjamín Lee, Chris Dwyer, and Dan Sorin, my undergraduate research advisors at Duke who first taught me to how to do research, helped me publish my first papers, and advised me during my graduate school search.
I was lucky to have some incredible mentors in industry. Thanks to Bruce Khailany, Rangharajan Venkatesan, Steve Keckler, Joel Emer, and Michael Pellauer of NVIDIA Research for the mentorship and experiences during my two summer internships. I also thank Hans Jacobson and Pradip Bose of IBM T.J. Watson Research Center for their collaboration and support during the first years of my Ph.D.

My Ph.D. experience would not have been the same without the bright and driven students of our group. I will miss our long lunches, C++ Thursdays, incessant coffee runs, late nights during tapeouts, passion for speedup, and competition for group disk space. In particular, thanks to Sophia, Svilen, and Yuan for our awesome collaborations. Sophia and I worked together for nearly two years, and it was an intense period of brainstorming, prototyping, and discussion that I won’t ever forget. From Svilen, I learned about operating systems, datacenters, simulators, and more about esoteric C++ than I’d ever thought I wanted to know. And Yuan, thanks for teaching me some finer points about cache coherency and powering me through the last part of my Ph.D. Thanks to everyone else in the group for making these years so enjoyable: Brandon, Bob, Udit, Lillie, Kevin, Emma, Hyunkwang, Simon, Simone, Paul, Saekyu, Silvia, Rafael, Siming, Tao, Mario, Marco, Mark, Saketh, Thomas, Thierry, Glenn Ko, and Glenn Holloway.

A special thanks goes to my girlfriend Yiqun Zhang for her loving support and companionship over the past years. Meeting her was probably the luckiest thing to happen to me during my time in graduate school. Our times together, whether on vacation or just working together at school, were some of my happiest memories over the past three years. I can’t wait for the long distance to end and for us to continue sharing more fun times and adventures on the west coast.

Finally, thanks to my wonderful family for their support and love over the past five years. In particular, thanks to Mom and Dad for spoiling me with good food whenever I’m home for a long weekend, and thanks to my sister for ensuring that when she’s around, there’s never a boring day at home.
Introduction

Since the dawn of the semiconductor revolution, computer chip designs have steadily increased in size and complexity in the quest for greater performance. For many years, chip designers were able to achieve steady performance gains within the same power budget thanks to two trends: Moore’s Law and Dennard scaling. Moore’s Law predicted that due to economies of scale, the number of transistors on a chip would double every two years. Dennard scaling, a property of complementary metal-oxide semiconductor (CMOS) devices first discovered by Robert Dennard, further showed that as the size of a CMOS transistor shrank, it could switch at faster clock frequencies while keeping the power density constant [45]. Together, these two scaling trends powered the semiconductor industry for decades.

However, around 2004, Dennard scaling slowed down. Shrinking transistor feature sizes no longer allowed chipmakers to scale supply voltages down and clock frequencies up, so the traditional methods of delivering greater single-threaded
performance by adding pipeline stages and speeding up the clock would only increase power without providing any additional performance. As a result, the chip-makers moved away from building single-core processors with deeper pipelines and faster clocks, which burned more power, to multi-core processors with shallower pipelines and slower clocks. As a consequence of power-constrained design, performance per watt instead of raw performance became the primary metric by which chips were compared.

In order to take advantage of multi-core processors, software needed to be parallelized. Since the arrival of the first multi-core processors, many great strides have been made in extracting more thread-level parallelism out of workloads, spanning compiler support for OpenMP and auto parallelization [23, 130] to hardware features like simultaneous multithreading [172]. Yet all these advancements pertain to a homogeneous substrate of general purpose processors, and improvements to performance per watt are fundamentally limited by the overhead imposed by general purpose hardware, such as instruction decoding, limited execution units, memory bandwidth to the core, dynamic instruction scheduling, and more.

In order to alleviate the overheads of generality, we must turn to hardware specialization. Hardware specialization refers to building dedicated circuits called accelerators for specific workloads or tasks. Hardware accelerators employ customized datapaths with high computational and/or memory-level parallelism to achieve performance and power efficiency orders of magnitude higher than general-purpose CPUs. As a result, accelerators have now become first class design citizens for both academia and industry. In commercial chips, accelerators have been increasingly adopted for a variety of workloads. Nearly all processors now include accelerators for multimedia, network processing, and more. On power-constrained mobile systems-on-chip (SoCs), we now find that much of the die area is devoted towards accelerators. Using die photo analysis, Figure 1.0.1 demonstrates that over the past eight generations of Apple mobile SoCs, the number of specialized IP blocks has steadily increased. With device scaling slowing, the number of specialized hardware blocks is only expected grow with time to meet the steady demand for increased performance.
Indeed, hardware specialization is the future of the computing industry. Academia and industry alike have poured many years into researching and developing heterogeneous computing ecosystems, and in the process, we have accumulated a wealth of knowledge on accelerator design. However, most of the attention has devoted to designing the accelerator in isolation. This overlooks two important design considerations: the system in which they will be integrated and the context of the end-to-end workload that they will be used in. Accelerators designed in isolation are unlikely to achieve their total potential performance and energy efficiency once integrated into a chip due to system-level effects like total memory bandwidth, shared resource contention, and more. Furthermore, designing accelerators in isolation brings the focus onto computationally expensive kernels within a single workload, but it overlooks opportunities of accelerating smaller code regions common across applications. As the number of accelerators on chips grows, the need for a holistic, end-to-end, system-level approach to accelerator design and analysis becomes clear.
1.1 Thesis Contributions

This thesis describes the work I have done to motivate and enable architects to adopt a holistic system-level approach to accelerator design. I have validated and improved widely used power modeling tools in the academic community, built an integrated SoC simulator to enable power/performance/area modeling of end-to-end accelerated workloads, explored how to leverage SoC-accelerator co-design to improve a wide range of accelerators, built deep learning frameworks, and investigated opportunities for hardware specialization targeted at general purpose workloads in the datacenter.

1.1.1 Quantifying Sources of Error in McPAT

In the era of power-constrained design, robust power modeling tools are essential elements in the computer architect’s toolbox. For those seeking to perform high-level design space explorations, architectural power modeling tools are the preferred choice due to their high-level nature and evaluation speed. However, the outputs of these tools are only trustworthy if the tool has been thoroughly validated against real designs. In the area of processor design, McPAT [101] is the de facto power model, but the literature does not yet contain a careful examination of its modeling accuracy. In addition, the issue of how greatly power modeling error can affect subsequent architectural studies has not been quantified before.

In this project, we were the first to present the first rigorous assessment of McPAT’s core power and area models with a detailed, validated power modeling toolchain used in current industrial practice. We find that McPAT’s predictions can have significant error because some of the models are either incomplete, too high-level, or assume implementations of structures that differ from that of the core at hand. We demonstrate that large errors are possible when using McPAT’s dynamic power estimates in the context of voltage noise and thermal hotspots, but for steady-state properties, accurately modeling leakage power is more important. Based on our analysis, we are able to provide guidelines for creating accurate McPAT models, even without access to detailed industrial power modeling tools. We
conclude that in spite of its accuracy gaps, McPAT is still a very useful tool for many architectural studies, and its limitations can often be adequately addressed for a given research study of interest.

1.1.2 Enabling SoC-Accelerator Co-Design using gem5-Aladdin

For many years, accelerator design was done predominantly via writing RTL implementations. This low-level approach to accelerator design had many drawbacks. It is very slow to write correct RTL, synthesize the design, and run simulation. The time consuming nature of this design flow meant that major architectural design choices are made well before RTL writing begins, thereby making early-stage architectural design space exploration infeasible. This changed with the release of Aladdin [156], a pre-RTL accelerator power/performance/area modeling framework that enabled cycle-level simulation and rapid design space exploration from only a high-level C description of the workload. Aladdin was a major step forward in modern accelerator design methodology.

However, Aladdin can only model the accelerator’s characteristics in isolation. When accelerators are integrated into an SoC, the co-design of accelerator microarchitecture with the system in which it belongs is critical to balanced, efficient accelerator microarchitectures. For example, data movement and coherence management for accelerators are significant yet often unaccounted components of total accelerator runtime, resulting in misleading performance predictions and inefficient accelerator designs. In order to explore the design space of accelerator-system co-design, new research infrastructure is needed. Therefore, we developed gem5-Aladdin, an SoC simulator that captures dynamic interactions between accelerators and the SoC platform, and validate it to within 6% against real hardware. gem5-Aladdin is capable of running end-to-end workloads on complex SoCs that contain a mixture of CPUs, accelerators, memory systems, and more. Our co-design studies show that the optimal energy-delay-product (EDP) of an accelerator microarchitecture can improve by up to $7.4 \times$ when system-level effects are considered compared to optimizing accelerators in isolation.
1.1.3 Optimizing Multi-Accelerator Systems via SoC Interfaces

As accelerators become first-class citizens in contemporary heterogeneous SoCs, we must move beyond optimizing accelerators in isolation and consider how to overcome SoC integration challenges to deliver high end-to-end performance while also providing a practical programming model. Real applications increasingly span multiple algorithms and IP blocks, such as real-time image classification or continuous speech recognition. Thus, the interfacing of accelerators within the SoC has become a pivotal consideration for their overall performance and energy efficiency. DMA is the simplest interface to implement in hardware and is thus widely used today, but it leaves many problems for software management to solve, creating both significant overhead and a tedious programming model. With system-level design approaches, we can incorporate resources and features of the SoC into the accelerator to unlock greater efficiency. In this project, we explore a type of SoC-accelerator interface we call delegated coherency, in which an accelerator is directly connected to the last-level cache (LLC) of a CPU cluster. This technique provides coherent access to shared global memory for the accelerator without having to implement a cache inside the accelerator and without the overhead of a fully-coherent SoC bus master.

Using vision pipelines and deep neural networks (DNNs) as the driving workloads, we demonstrate how delegated coherency interfaces can improve performance by 20-40% and energy by up to 40% with no changes to accelerator datapaths. Furthermore, we show that these benefits are resilient to shared resource contention in the LLC from concurrently running applications, due to a software prefetching mechanism made possible by this interface. In order to study these end-to-end workloads, we also developed SMAUG, the first simulation-compatible DNN framework supporting custom accelerator models. This work shows the power of delegated coherency interfaces as a noninvasive way to unlock accelerator efficiency via the shared LLC.
1.1.4 Mallacc: Accelerating Memory Allocation

When identifying workloads amenable for acceleration, most of the attention has focused on the “deep” acceleration approach: find a “killer application” (e.g. deep neural networks), identify its most costly kernels, and map them to custom hardware. This approach has certainly seen its share of success, but the focus on deep acceleration means that only the killer applications benefit. For this reason, the proliferation of hardware accelerators has largely skipped server processors powering cloud workloads – they are simply too diverse and do not have any such killer applications or hotspot functions that can be optimized with a deep approach. However, a system-level analysis of cloud workloads across an entire fleet of datacenters has shown that there exists a collection of low-level routines common to cloud workloads called the “datacenter tax”. Although they individually contribute comparatively little time, across the entire fleet they comprise up to 30% of all cycles in Google’s datacenter [92]. This motivates a “broad” acceleration approach, where low level routines shared across a wide range of workloads are sped up to save total overall cycles.

Of the routines in the datacenter tax, dynamic memory allocation is the largest, consuming nearly 7% of all cycles in Google datacenters. With the trend towards increased specialization of hardware, we propose Mallacc, an in-core hardware accelerator designed for broad use across a number of high-performance, modern memory allocators. The design of Mallacc is quite different from traditional throughput-oriented hardware accelerators. Because memory allocation requests tend to be very frequent, fast, and interspersed inside other application code, accelerators must be optimized for latency rather than throughput and area overheads must be kept to a bare minimum. Mallacc accelerates the three primary operations of a typical memory allocation request: size class computation, retrieval of a free memory block, and sampling of memory usage. Our results show that malloc latency can be reduced by up to 50% with a hardware cost of less than 1500 μm² of silicon area, less than 0.006% of a typical high-performance processor core.
2 Background and Related Work

This section will discuss the rich history of past work in architectural power/performance/area modeling, accelerator research, SoC design and prototyping, and opportunities for specialization in the cloud.

2.1 Architectural Power Modeling

On modern processors, a large fraction of the total power is consumed by SRAM structures like large caches. Accordingly, one of the first and still widely used tools for power modeling is CACTI [181], a high-level SRAM modeling tool. CACTI began with support for only SRAM-based caches, but it has since evolved to support a variety of storage technologies, including DRAM and eDRAM, as well as a range of technology nodes. Twenty years since its original release, CACTI is now on its sixth version and is still widely used in the architecture community.
One of the first architectural CPU power modeling tools was Wattch [15]. Wattch provides parameterizable power models for many of the pipeline structures in an out-of-order CPU, such as register renaming tables, instruction window buffers, reorder buffers, TLBs, etc., as well as models for combinational logic structures (e.g. functional units), wires, and clock distribution. CACTI is used to model cache structures. Wattch was integrated into SimpleScalar [6], a widely used CPU simulator. The development of Wattch spurred a great deal of power-related research.

Over time, several trends spurred the development of a new architectural power modeling framework. First, as transistors shrank and supply voltages fell, leakage power became a growing fraction of total power, and Wattch did not support modeling of leakage power. Second, as multicore processors became the norm, architects needed to model interconnects, memory controllers, and other structures inherent to them. Finally, Wattch only modeled performance and power, but the rise of mobile computing meant that area had also become a design constraint.

This led to the development of McPAT, an integrated power, area, and timing framework for multicore processors [101]. McPAT provides a high-level interface to specify microarchitectural parameters of processors and the surrounding system. Under the hood, McPAT relies on CACTI to estimate power and area for array structures, and it supplies its own set of models for more irregular structures like functional units, memory controllers, network interfaces, and more.

2.1.1 Validation of Power Modeling

Despite the widespread use of architectural power models, comprehensive validations of them against real processors has been missing in the literature. Architectural power models compute total power by summing up the power from each CPU structure (e.g. reorder buffer, register files, functional units, etc), and while it is straightforward to measure total system power, measuring per-component power from real hardware is much more difficult. Per-core power can be measured by disabling all but one of the cores, but power from the individual core structures
usually cannot be measured in isolation. Faced with this dilemma, existing validation efforts have fallen back to three options: only validate core-level power, use RTL simulation when RTL is available, or attempt to infer power from other measurements.

**Validate core-level power**

McPAT itself takes this route. With the exception of an older processor for which more detailed power breakdowns have been published, all other validation data only compare total core power, crossbars, shared caches, global clock trees, etc. Furthermore, their validation is only able to compare peak power, assuming maximum switching activity factors, so this validation provides no workload-specific information. Other work have used hardware performance counters to measure power. For example, Zhai et al. [185] describe a power model called HaPPy that relies on a feature of recent Intel CPUs called Running Average Power Limit (RAPL) [79]. Although RAPL only provides total power for all cores on the chip, Zhai combines this information with performance counters to deduce information about hyper-threads and core-level activity. This technique could be extended to reveal additional unit-level data.

**RTL Simulation**

When RTL is available, post-synthesis and place-and-route extraction can be performed to obtain accurate power estimates. Naturally, this can only be done on open source hardware or by a semiconductor company on its own chips.

For open source hardware, Govindan et al. [59] validated a Wattch power model for a prototype TRIPS processor [21] against RTL simulations as well as real hardware measurements to categorize and quantify the types of modeling error observed. They find that the primary contributors to error are the clock tree, technology models, and unstructured logic. These are important general insights about the limitations of architectural power modeling. On the other hand, tools like Wattch and McPAT were not originally designed to model an EDGE architec-
ture, but rather a conventional load-store pipeline, and contemporary superscalar processors are much more complex than the TRIPS prototype they used.

For commercial chips, accurate early stage power modeling is also crucial. At IBM, Jacobson et al. [80] described an effort to develop a mathematically rigorous power model abstraction based on a small number of machine activity statistics. Their effort is focused on the IBM POWER7™ processor, a 16-core superscalar server processor. The result was an iterative process to select the $n$ utilization metrics most highly correlated with total core power and a methodology to build a first-order linear power model based on those activity statistics, with RTL simulation used as the baseline. This approach naturally produces unit-level power models as part of the overall per-core power model.

**Power inference from other sources**

Since power consumption directly causes the chip to emit heat, it is possible to correlate local chip temperatures with the local power draw. Mesa-Martinez et al. [120] presents a genetic algorithm that creates a power model by correlating a set of power equations to measured chip temperature. This requires removing the packaging from the chip, submerging the chip in a thermally transparent heat sink (an oil that transmits infrared light), and imaging the chip under various loads with an infrared camera. This is a very unique and specialized approach, but the extracted power model is validated only with a multimeter measuring total chip power. That being said, with specially crafted microbenchmarks, it could be possible to heat specific parts of the processor and infer localized power consumption.

### 2.2 Accelerator Design Methodology

This section will present an overview of design methodologies used for accelerators and SoC integration. In this section, I use the term “accelerator” to refer to a fixed-function application-specific hardware block, comprised of a number of custom datapath lanes and local memory structures. Other types of accelerators, such as GPUs and DSPs, are not discussed in this section.
Throughout much of its design history, accelerators were built from the start with the RTL (register transfer level) design flow. A designer uses an RTL language, such as Verilog or VHDL, to specify the behavior of the accelerator at every cycle using basic hardware building blocks like wires, registers, adders, multiplexers, etc. When the RTL is finished and verified in functional simulations, the designer then synthesizes the RTL into a netlist, which is an implementation of RTL using basic digital gates like AND and OR. During synthesis, if the RTL description is found not to be able to meet timing constraints imposed by the target clock frequency and technology library, synthesis will fail and the designer must address the problems. Finally, the netlist is placed and routed into a physical layout through semi-automated tools.

Clearly, this is a highly tedious and slow process at every step. RTL implementation is very verbose and error prone, and RTL simulation is so slow that large test cases often cannot be run in a reasonable amount of time. Synthesis and place-and-route can take many hours or even days to complete on a large cluster of machines, and failures require the designer to intervene, fix the problem, and restart the process. The slow process of evaluating a single design point with RTL precludes using this design flow to perform any kind of early-stage design space exploration. Instead, major architectural decisions are made using high-level analytical models and designer intuition.

The advent of high-level synthesis (HLS) brought about a major improvement in accelerator design flows. Instead of writing RTL and having to manually specify cycle-level behavior, the designer can describe the accelerator using a high-level language like C or C++. An HLS tool will transform this description into RTL, which can then be run through the standard backend flows. Certain microarchitectural parameters, like datapath parallelism, local memory partitioning, and loop pipelining, can be easily specified using HLS tool directives or code pragmas. Examples of such tools include Vivado HLS, Mentor Graphics Catapult, and Stratus HLS.

HLS greatly eases the task of designing hardware by raising the level of abstraction for the designer, but this benefit is not free. The quality of the generated RTL
often cannot match that of hand-written RTL, and the need to generate valid RTL in the first place means that HLS tools tend to be quite slow themselves. Also, their relative immaturity means that certain coding styles and structures are not well supported. For rapid early stage accelerator design space exploration, a faster alternative to HLS is needed.

This motivated the development of Aladdin, a pre-RTL accelerator power/performance/area simulator [156]. Aladdin takes a C description of the accelerator, generates an dynamic execution trace of the workload, and uses that trace to produce a dynamic data dependence graph. Then, it takes a set of user-specified hardware constraints, like local memory partitioning, loop unrolling, and more, and transforms the graph accordingly to respect those constraints. Finally, it simulates the accelerator by performing a breadth-first traversal of the graph. Like HLS tools, Aladdin uses a C description of the accelerator, but since it does not need to generate valid RTL, Aladdin can run orders of magnitude faster than HLS tools. Aladdin has been highly influential in the architecture community, and my contributions to the Aladdin family of tools form an integral part of this dissertation.

2.3 SoC-Accelerator Co-Design

Accelerators rarely exist in isolation. They are most often integrated into an SoC, and integration presents new challenges for architects. First, there are a variety of interfaces that can be used to connect the accelerator to the system, and each interface presents a different set of tradeoffs. Second, the accelerator may compete for shared resources with other agents in the system, and these system-level effects must be considered during accelerator design. Third, accelerators must compete for area and power resources on the SoC, so during integration, a holistic evaluation methodology that takes into account competing factors is needed. In order to evaluate solutions across all of these fronts, architects need robust SoC and accelerator modeling infrastructure.
2.3.1 SoC-Accelerator Interfacing

The logical interface between the accelerator and the SoC, at a high level, describes how much hardware management of cache coherency is involved during data exchange. The typical interface is called Direct Memory Access (DMA), in which software will handle the tasks of cache line flushing and invalidation and programming a hardware DMA engine to perform the data transfer. In this way, the accelerator can talk directly to main memory and not worry about coherence. DMA is highly efficient at bulk data transfer, but it does not suit fine-grained access patterns due to the high cost of software intervention. Bulk data transfers were perfectly suitable for the first accelerated workloads, like video decoding, but over time, the growth of interest in hardware acceleration meant that an easier programming model was needed, and some workloads also needed an efficient interface for fine-grained access patterns. This spurred a great deal of research and development into coherent accelerator cache interfaces. This section briefly summarizes some influential work in this field.

IBM Coherent Accelerator Processor Interface (CAPI): This is an interface introduced in the IBM POWER8 server processor to connect the CPUs with an accelerator running on an FPGA attached over PCIe [179]. It consists of two parts. The first is a coherent accelerator processor proxy (CAPP), implemented as a special unit on the POWER8 chip, that participates in cache coherency protocols on behalf of the accelerator. The second is a power service layer (PSL) on the FPGA that bridges the accelerator and the CAPP. The PSL contains a small cache that the accelerator can use as local memory and works in concert with the CAPP to provide address translation, so the accelerator can work with virtual memory.

ARM Accelerator Coherency Port (ACP): This interface on ARM SoCs allows an accelerator to directly access the last-level cache (LLC) of the host CPU as an alternative to DMA [112]. The LLC will handle all cache misses on behalf of the accelerator. This interface means that software no longer needs to flush and
invalidate cache lines, because accesses from the accelerator are fully coherent. However, ACP does not provide not full coherence – the CPU has no coherent visibility into the accelerator’s local memory – because the accelerator itself may not have a cache or any other way to gain exclusive ownership of a cache line.

**Heterogeneous System Architecture (HSA):** This is a set of specifications developed by a foundation of major semiconductor companies, such as AMD and ARM, that enable tighter integration between CPUs, GPUs, and other compute devices [147]. One of the major goals of HSA is providing unified virtual memory, which enables applications running on CPUs and GPUs to easily share data by passing pointers between compute devices instead of performing explicit data copies. This is enabled in hardware through the HSA memory model, compliant cache coherency protocols, pageable memory, and cross-device address translation. Other features of HSA include pre-emption and context switching on GPUs to hide long latency events, user mode work dispatching to avoid OS and driver overhead, a virtual ISA designed for parallel programming, and more.

**Specialized Coherency Protocols:** Cache coherency has been studied by architects for decades, but until recently, most of that work has targeted general purpose processors that must run a highly diverse set of workloads. However, accelerated workloads often lend themselves to specialized coherency protocols that require less on-chip storage for tracking, generate less traffic, reduce protocol complexity, and improve performance compared to a general purpose protocol. Recent work has focused on adapting timestamp coherency (TSC) to accelerators. Unlike traditional valid-invalid protocols that depend on message passing for state updates, TSC predicts the length of time a cache line is expected to be needed by a sharer in the system and self-invalidates the line when that time expires. TSC was first proposed by Nandy et al. [124]. Since then, it has appeared in the context of multiprocessors [109], GPUs [161], and fixed function accelerators [97].
Modeling infrastructure for accelerator-centric SoCs is relatively scarce. One example of an SoC simulator is PARADE [37]. PARADE integrates gem5, a widely used system simulator [12], with an accelerator model based on HLS. First, a user uses an HLS design flow to produce RTL for an accelerator, and then PARADE uses pipeline latency information from the HLS scheduling reports to estimate the number of cycles the accelerator would require for a given input data size. PARADE runs full system simulation in gem5, so in addition to being able to leverage the full flexibility of gem5, it can also explore OS effects. However, PARADE only supports traditional DMA-based data movement schemes. The accelerator is not able to issue external memory reads and writes during its execution; it can only access its local scratchpads. In addition to the inherent limitations imposed on memory access patterns by this constraint, this means that more complex local memory systems, such as hardware-managed caches, cannot be evaluated using PARADE.

There are several commercial development boards, readily available to researchers, that provide an attractive platform for studying specialized architectures. These boards often contain all the components essential to a modern mobile SoC, including a multicore CPU, a mobile-class GPU, Ethernet, and DRAM, with an FPGA. One such widely used board is the Digilent Zynq Zedboard, which has dual core ARM Cortex-A9 CPU, 512MB of DRAM, an HDMI transmitter, and a small FPGA [76]. The Xilinx Zynq Ultrascale MPSoC is the latest iteration of this family of SoC development boards. It contains a quad-core ARM Cortex A53 CPU, a dual-core ARM R5 real-time CPU, a Mali-400 GPU, 4GB of DDR4 DRAM, and a large FPGA with its own dedicated 512MB of DRAM [77]. These boards have seen widespread adoption for prototyping of hardware accelerators in a real system among academic researchers due to their relative ease of use and level of software support. For example, Sadri et al. performed an early investigation into the performance of the ARM Accelerator Coherency Port on the Zedboard [148]; Moreau et al. used the Zedboard to implement a prototype of SNNAP [122]; and Shao et al. used the Zedboard to validate a new SoC-simulator [158].
2.4 Broad Acceleration

The notion of identifying code hotspots within general-purpose applications for acceleration was first introduced by Venkatesh et al. with Conservation Cores (C-Cores) [174]. A c-core is a small hardware unit integrated into the pipeline of a CPU which is treated like a functional unit. To build a c-core, a workload is profiled, hot functions are identified, and an automated design flow maps the source code of that function into a custom datapath that is synthesized. The goal of a c-core is not to improve performance, but rather to match the original performance with much lower energy. Reconfigurable registers in the c-core and an exception handling mechanism allow the c-core to adapt to source code patches. On a set of SPEC2000 and PARSEC workloads, the hotspot functions often comprise between 20-50% of total execution time. A follow-up work, QsCores, expanded on this idea by searching for similar hot code sequences across a wide set of workloads rather than on a per-workload basis [175], so that fewer custom hardware blocks with more reconfigurability are needed.

The work on c-cores demonstrated that code hotspots for SPEC-like workloads exist and are potential targets for hardware acceleration. If such accelerators are to be deployed, they will most likely be deployed in the cloud, because ASIC development is very expensive and datacenter operators are among the few customers that can afford to pay such a premium for their hardware. However, it has long been known that SPEC workloads are unrepresentative of datacenter workloads. Recent work by Kanev et al. [92] showed that unlike SPEC, datacenter workloads are highly bottlenecked by the instruction cache, with the CPU being completely starved of instructions for 5-10% of all cycles. There are also no significant code hotspots within any application; however, across applications, they did find hotspots in a set of low-level library routines which are unique to running distributed workloads. These routines, called the “datacenter tax”, collectively consume up to 30% of all CPU cycles in Google datacenters. They include memory allocation, remote procedure calls, protocol buffer (de)serialization, and hashing, and they may be prime candidates for broad acceleration techniques.
Quantifying Sources of Error in McPAT

Architectural power modeling has enabled a tremendous amount of power-related research, but the value of the tool relies on the accuracy of its predictions. This chapter discusses a comprehensive validation of McPAT, a widely-used power modeling tool for multicore processors. Using the IBM POWER7™ server processor, we find that McPAT often only models a subset of the area and power for a given component, and its predictions can have significant error due to a variety of reasons. We demonstrate how we addressed McPAT’s sources of error, discuss why this error may not have been noticed previously, and provide guidelines on how to avoid these sources of error in the future.
3.1 Introduction

Architectural power modeling tools like Wattch [15] and McPAT [101] have enabled researchers to perform fast, integrated design space explorations of multicore and manycore CPU configurations. As the current de facto power modeling framework, McPAT has seen widespread adoption in the architecture community, but to date, a thorough validation of its area and power models for a contemporary high-performance processor does not exist in the literature. McPAT’s authors only had access to published data on peak power for the various cores they validated, so the validation of McPAT in the existing literature [101, 103] was very coarse-grained. For example, for three of the four cores examined, they validate total core peak power but not for units within the core. Also, peak power is not relevant for many architectural studies evaluating application-specific behavior. Therefore, a seemingly accurate result could be masking significant error canceling. More importantly, error in power modeling can greatly impact the conclusions drawn from modeling studies that rely on accurate power models, but it is unclear how significant this effect is.

There has been a significant body of work for creating performance simulators [11, 24, 90, 132, 150] and power models [15, 80, 100, 101, 120, 152], but validation of these models tends to be coarse-grained and emphasizes the methodology of creating the power model. Fine-grained validation requires access to detailed design data often not available in academia. In this chapter, we rigorously assess McPAT’s area and dynamic power models for the cores of a conventional general-purpose microprocessor, the IBM® POWER7™ server multicore chip. With privileged access to POWER7 design documentation, we construct three McPAT models that cumulatively improve accuracy with respect to a proprietary power model. Although we focus on dynamic power, we will touch on leakage too.

Our results show that McPAT’s power and area models can have significant error because they are either incomplete, too high-level, or represent an implementation of a structure which differs from that of the core at hand. Incomplete models result in McPAT only modeling a subset of the total area and power for a component.
The subset is mostly comprised of caches, CAMs, and other SRAM array-based structures and does not account for many examples of control logic. We do not introduce any new models in this report, but we show that fixing the other types of error resulted in significant improvements to power and area estimates. Only a few of the specific errors we found are POWER7-specific; in fact, most of them would affect a generic out-of-order superscalar CPU.

Power models like McPAT are used for other relevant studies, such as voltage noise and heat dissipation. We perform two simple case studies in these contexts to quantify their sensitivities to power error. Our results show that steady-state properties, like overall chip temperature and static IR drop, are quite resilient to dynamic power error. Spatial properties, like thermal hotspots, benefit from improvements to McPAT because average power overestimates have been mitigated. Temporal properties, like inductive noise amplitude, also benefit from improved average power estimates because these reductions simultaneously shrink overestimates of transient power swings. We conclude that leakage power accuracy is more important for steady-state modeling, but dynamic power accuracy is more important for temporal and spatial modeling.

Finally, we discuss why the inaccuracies we report may not have manifested in prior work using McPAT. We also provide specific guidelines that future studies using McPAT can observe to avoid being misled by power error. Our guidelines can be useful even without having proprietary power modeling tools. That being said, researchers will hugely benefit from having validated models released by industry, and we hope that this vision will come to fruition.

3.2 Power Modeling Approaches

Architectural power models are widely used to estimate the power consumption of a microprocessor, where high-level architectural and microarchitectural parameters (e.g. cache sizes, page size, and pipeline depth/width) and activity factors (e.g. cache accesses, total instructions) are specified to the power modeling tool, which abstracts away the underlying implementation details. These high-level ab-
stractions, which represent a tradeoff between detail and flexibility/ease of use, enable an architect to quickly evaluate design decisions and explore various design spaces. McPAT and Watch are two well known examples of these models.

Both of these tools are analytical, meaning that they use analytical equations of capacitance to model dynamic power. In contrast, empirical models, like Power-Timer [16] and ALPS [62], use pre-characterized power data and equations from existing designs. For structures like control logic that are difficult to model analytically, empirical models and/or fudge factors are often used instead. The differences between analytical and empirical models have been described in past work [17, 106].

The IBM power modeling tool, which we refer to as DPM (detailed power model) and use as the point of comparison for this work, is an empirical model that tips the balance towards painstaking detail. For example, it can track misaligned cache accesses to precisely calculate the extra energy required for the operation, and it can compute branch detection power by knowing how many branches were in the group of fetched instructions. Base energy values and power computation equations are manually updated by circuit and layout designers. Such detail not only enables very fine grained power modeling, but enables the tool to accurately compute clock-gating factors, which have been shown to be critical to accurate power modeling [80]. DPM has been validated against circuit simulations to within 5% accuracy. However, such detail also precludes high-level design space explorations because the model is so closely tied to a specific implementation. In this work, we quantify the error in architectural power models arising from this tradeoff.

3.3 Assessment Methodology

In our assessment, we created and compared three models of the POWER7 core, described below:

- **MRO**, a no-revisions model based on data published by Sinharoy et al. [163].

  This model mimics the typical McPAT use case where all parameters are
<table>
<thead>
<tr>
<th>Component</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-cache</td>
<td>32KB, 4-way set associative, 8-way banked</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>32K entry tournament predictor indexed by global history hash; 128-entry address cache for indirect branches; 16-entry jump return address cache</td>
</tr>
<tr>
<td>Frontend</td>
<td>Fetch up to 8, decode up to 6, issue up to 8</td>
</tr>
<tr>
<td>General purpose RF (GPR)</td>
<td>32 architected and 112 physical 64b registers, partitioned based on SMT mode</td>
</tr>
<tr>
<td>Vector RF (VRF)</td>
<td>64 architected and 172 physical 128b registers, partitioned based on SMT mode</td>
</tr>
<tr>
<td>ROB</td>
<td>20 instruction groups, each composed of up to 6 instructions</td>
</tr>
<tr>
<td>Register renaming</td>
<td>80 shared between GPR and VRF; 140 more for other types of registers</td>
</tr>
<tr>
<td>Issue queues</td>
<td>48-entry unified queue for FXU, LSU, and VSU instructions; 12-entry queue for branches; various others. Managed by an age tracking matrix</td>
</tr>
<tr>
<td>Load/store</td>
<td>Two pipelines; 32-entry load and store queues</td>
</tr>
<tr>
<td>D-cache</td>
<td>32KB, 8-way set associative, 8-way banked</td>
</tr>
<tr>
<td>D-TLB</td>
<td>128-entry 1st-level TLB; 512-entry 2nd-level TLB</td>
</tr>
<tr>
<td>Execution units</td>
<td>2 fixed-point, 4 floating-point, 1 vector, 2 load-store, 1 branch, 1 condition register, 1 decimal</td>
</tr>
</tbody>
</table>

Table 3.2.1: POWER7 core configuration [163, 178].

derived from published reports (see Table 3.2.1). The validation method in the original McPAT report [101] is an example of this use case.

- **MR1**, a revised version 1 model that represents the most accurate core configuration parameters possible. Parameters were available through privileged access to detailed design documentation. This level of detail is typically absent (and sometimes impossible) in other power modeling studies.

- **MR2**, a revised version 2 model that incorporates source code changes to fix modeling assumptions in McPAT which are incorrect for POWER7 (and pertain to generic out-of-order CPUs) and could not be fixed purely through the available parameters. These changes are intended to be applicable for any general purpose chip and are not in any way POWER7-specific.

This methodology lets us quantify how much improvement McPAT can show with the best configuration possible and how much more it could improve if source
Figure 3.3.1: POWER7 chiplet simplified floorplan. The decimal-floating unit (DFU) is omitted, and the wraparound L3 cache is split into two parts.

code were directly modified. In future work, one could create a hypothetical "MR3" that adds modeling for missing components, like datapath control logic. However, such a model would only apply to a specific implementation, and validating this model could depend on manually writing RTL; this task would only be compounded if this model were made parameterizable. As our goal was to keep modifications in the spirit of McPAT, we do not introduce any logic models in MR2, but we will briefly mention some preliminary work towards a POWER7-specific MR3 in Section 3.6.

In this report, we use the following abbreviations for the core units, shown in the POWER7 floorplan in Figure 3.3.1:

- IFU: Instruction fetch unit (includes decoder).
- ISU: Instruction sequencing (i.e. scheduling) unit.
- LSU: Load-store unit.
- FXU: Fixed-point unit.
- VSU: Vector-scalar unit (floating-point).

These three models are compared against DPM. Unlike DPM, McPAT does not model every macro of each unit. Therefore, McPAT’s predictions are compared
against the subset of the total DPM measurement for a unit representing components actually modeled by McPAT (henceforth referred to as subset area and power). Recall that McPAT primarily models caches, SRAM arrays, and CAMs; it does not account for many control logic elements. As an example, McPAT models storage components of the instruction issue queues, but it does not model logic that dispatches instructions to these issue queues. Only the former would be included in the subset, but both would be included in the total.

Performance statistics are generated by an IBM performance simulator for POWER chips called M1, described by Srinivas et al. [164]. Performance models like M1 target a goal of 2% error. M1 has been validated against RTL simulations and is used in a regression test suite, so it is continually maintained. It dumps thousands of statistics for both DPM and McPAT to use. $V_{dd}$ is set to 1.01V and $f_{clk}$ to 4.0GHz.

We evaluated these three models for area and power using 20 benchmarks selected from SPEC2006 and SPEC2000. We selected 14 from SPEC2006 using the guidelines described by Phansalkar et al. [134]. From SPEC2000, we selected six workloads that represent control-flow complexity, memory/compute-bound behavior, and a mixture of these qualities. A synthetic stressmark called vsx used by the POWER7 design team to measure the chip’s TDP is also included.

Area estimates are compared against actual areas measured from detailed floorplans. Dynamic power estimates are compared against those produced by DPM. Because DPM only models core units and the private L1 cache, the L2 and L3 caches are excluded from our validation, along with uncore components like the memory controllers and interconnect. For our case studies, we use an alternative method to account for L2/L3 cache and uncore power.

### 3.4 Assessment Results

In this section, we compare the power estimates from MR0, MR1, and MR2 against DPM for each macro modeled by McPAT, identify and categorize the sources of error, and either show how the error was addressed or explain why a
fix was not attempted. Units are broken down into macros modeled by McPAT, so for the rest of this section, we only compare McPAT with DPM subset area and power, as we can’t compare McPAT’s predictions for macros it doesn’t even model. Note that figures are organized by units instead of by error type.

Our investigation reveals two overarching problems in opposite directions with McPAT: McPAT only models a subset of the total core, but this subset is globally overestimated, creating significant error canceling. By “error”, we refer to any deviation of McPAT’s estimate from DPM’s. These errors can be divided into four categories, listed roughly in decreasing order of importance: abstraction error, which arises from incomplete or missing models; modeling assumption error, in which assumptions about the underlying implementation of a microarchitectural structure differ from that of the CPU at hand; input error, which arises from incorrectly specified parameters; and coding error, which are programming mistakes.
Figure 3.4.2: Power cumulative distribution function of each unit. Power from macros in the McPAT subset are in light green, and those not in the subset are in dark green. Each bar represents power consumed by a single macro plus the power from all macros consuming less power than it. Power is normalized to the total of the unit.

3.4.1 Model Abstraction Errors.

Abstraction errors in McPAT are usually due to one of two reasons: either the model for a structure is incomplete or missing, or the parameters are too high-level to capture important low-level details. Incomplete models create the subset vs. total problem, and insufficiently detailed models create error within the subset’s power estimates.

Incomplete/Missing Models.

Figure 3.4.1 shows that for the IFU, ISU, and LSU, subset area and power account for less than 40% of the totals in POWER7. FXU and VSU subset areas are high because McPAT accounts for most of those functions. Some of the specific unmodeled macros are listed in Table 3.4.1. Notice that the majority of these unmodeled macros can be classified as control logic. The important observation is that the fraction of functional blocks occupied by control logic is much greater than what might be suggested by an architectural block diagram and thus can have a major
Table 3.4.1: Major core macros that McPAT does not model in rough order of power cost (per unit). Only one of these macros is specific to POWER7. The FXU and VSU are not included because for the most part, only the globally unmodeled macros apply to them.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Top power-consuming unmodeled macros</th>
</tr>
</thead>
</table>
| IFU  | Branch control logic (e.g. history management)  
      | Special purpose registers  
      | Instruction cracking and fusion  
      | Hardware thread management  
      | I-cache prefetcher control logic |
| ISU  | Instruction age tracking logic  
      | Issue queue data management  
      | Instruction group tag register files  
      | Per-thread dispatch and execution state  
      | Instruction dispatch to issue queue control logic |
| LSU  | D-cache prefetcher control logic  
      | Load/store queue thread and data management  
      | Load/store queue age tracking  
      | Cache line replacement policy logic |
| Global | Automated built-in self testing |

impact on abstraction error. Fortunately, Figure 3.4.2 shows that out of the unmodeled macros (in dark green), simply accounting for about one third of them will bring the subset fraction up to 80%. In other words, McPAT only needs to model a few more macros to account for the large majority of the unit’s power.

Despite the small fraction constituted by the subset, McPAT tends to overestimate its power and area. Figures 3.4.3 through 3.4.5 show that MRo’s estimates almost always exceed the subset’s values, and at the unit and macro levels, overestimates can exceed an order of magnitude. MR1 and MR2 improve upon MRo by providing more accurate configuration parameters and fixing sources of error, which is why in Figure 3.4.3, each revision causes power and area estimates to decrease. In a few cases, fortuitous error canceling also leads to the surprising result that MRo is actually quite close to DPM total power. For instance, on omnetpp, bwaves, mcf _2k and swim, MRo’s core power is merely between 0.1% and 4% off of DPM total (data not shown). These were the only four workloads out of 21 to exhibit this behavior, so we have little reason to believe that this would be the case.
Figure 3.4.3: Core power for 21 different benchmarks, normalized to DPM subset power. Each successive revision improves accuracy and thus brings McPAT’s estimates closer to DPM subset power. Note that the selected workloads from SPEC2006 have about the same power variance as those from SPEC2000.

in general for other workloads. In addition, overestimation has important ramifications for the voltage and thermal case studies that we later perform. As a whole, these results emphasize that proper power modeling validation must be done at the unit level or lower. Simply validating total core and/or chip power, the general approach taken by much previous work [73, 90, 101, 120], is misleading because it hides a large amount of internal error. These figures also clearly show that MR1 is in general not a huge improvement over MR0. That is, most of the error observed in MR0 is not input error. Thus, directly addressing the other modeling errors through source code changes in MR2 is necessary.

Figure 3.4.3 shows that many of the benchmarks exhibit similar error relationships between the three models regardless of the workload, so we will present macro-level analysis using only 7 of the 21 workloads surveyed. These workloads were selected to span the entire range of power variance observed.

INSUFFICIENT MODELING DETAIL.

While incomplete models create the subset problem, insufficient modeling detail creates power error within the subset itself. The two primary examples of this are McPAT’s assumptions for read and write ports and perfect clock-gating and data-gating.

**Read/write port errors.** The register files are good examples. McPAT defines
a parameter called peak_issue_width, which specifies the maximum number of instructions that can be issued from the issue queues in a cycle\textsuperscript{1}. POWER7 can issue up to six instructions per cycle from the unified issue queue, so McPAT defines an integer register file with twelve read ports and six write ports, a worst-case approximation. McPAT’s assumption is that every instruction issued per cycle needs two of its own read ports on the register file, but this is not necessarily true (for instance, some instructions only need one register operand for execution if there is an immediate). Also in POWER7, instruction dispatch and issue rules limit the number of simultaneous accesses to such structures. Thus in MR2, we manually specified the number of read and write ports. This change contributed to a large fraction of register file area and power reduction, seen in Figures 3.4.12\textsuperscript{a} and 3.4.12\textsuperscript{b}. For processors that have separated issue queues, setting this parameter to the maximum of any individual queue may be a simpler solution.

Another such parameter is number_instruction_fetch_ports. McPAT uses this one parameter to set the number of read/write ports for the I-cache and instruction buffer, among others. But in POWER7, the I-cache and instruction buffer have different port configurations because of their different physical structures. In MR2, we added additional instruction buffer parameters to resolve this issue.

\textsuperscript{1}This interpretation of “peak issue width” agrees with how this parameter is used by the example models in McPAT.
Figure 3.4.5: Core power breakdown by unit, normalized to DPM subset power. Power error relationships across the models are largely consistent regardless of the benchmark.

**Perfect clock-gating and data-gating.** Clock-gating prevents switching activity on clock signals from drawing power while a macro is not in use, and data-gating does the same for data signals in combinational logic. Assuming perfect clock- and data-gating means that any components not in use at any given cycle consume zero dynamic power. This assumption is reasonable for structures like SRAM arrays and makes some sense for McPAT since the vast majority of McPAT’s modeled structures are caches, arrays, or CAMs. However, logic circuits cannot always be perfectly clock- and data-gated due to design complexity or timing reasons, and it has been shown that accurately capturing clock-gating factors is critical to accurate power models [80]. As a result, McPAT tends to underestimate dynamic power for components that include nontrivial amounts of logic. Even though POWER7 components do consume close to zero dynamic power when not being accessed, McPAT’s estimates in these scenarios are smaller still. This causes underestimates for the branch target buffer and vector register file on gcc and mcf. In our exe-
 execution traces, _mcf_ rarely uses the VRF, so all three models estimate effectively no VRF dynamic power on this benchmark (Figure 3.4.13). The only exception to the perfect data-gating assumption is the pipeline latch model, because McPAT models both switching and holding power for flip-flops in pipeline latches.

3.4.2 **MODELING ASSUMPTION ERRORS.**

Modeling assumption errors occur when the CPU’s implementation of a microarchitectural structure differs from McPAT’s modeled implementation. McPAT’s SMT and control logic models are the main contributors to this error type.

McPAT’s SMT model.

McPAT provides a set of assumptions regarding what hardware structures are shared, partitioned, or duplicated [103]. However, if some of those assumptions are incorrect for the CPU at hand, there isn’t always enough flexibility to compensate with the provided parameters. POWER7 is a four-way SMT design that shares some of the resources McPAT assumes are duplicated, so deduplicating hardware where appropriate in MR2 always resulted in at least a fourfold reduction in area for that component.

These corrections do not necessarily translate to a fourfold improvement in power for the same unit. One of two reasons can usually explain this:
Figure 3.4.7: Instruction fetch unit power breakdown.

- For a cache structure, McPAT sometimes duplicates the area of the element without increasing its energy per access (which may be the desired effect).

- The element that was duplicated accounts for only a small fraction of the total unit power.

The register renaming unit is a good example of both reasons (Figure 3.4.9). McPAT’s SMT model assumes separate renaming tables per thread [103] whereas POWER7 uses shared renaming tables. Duplicated renaming tables in McPAT are modeled by computing area and energy per access for a single renaming table, whose area is then multiplied by the number of hardware threads. In MR2, we deduplicated hardware, reducing area error greatly, but power was unaffected. It turns out that MR2 estimates the renaming table power to within 10%, but the renaming unit includes an analytical logic model for dependency checking logic that overestimates power nearly tenfold. This logic, which is not duplicated, has large
errors because the POWER7 implementation greatly differs from McPAT’s model implementation.

One way to address this duplication problem without modifying source code is to specify $1/N$ of the actual number of entries, where $N$ is the number of hardware threads. When McPAT multiplies area by $N$, it is effectively modeling an $N$-way physically partitioned structure. There are cases where this structure is true or approximately true in POWER7 (such as the load/store queues), but this is not always the case. Consider the register renaming free lists: the number of entries is set by the number of physical registers, so this method would mean specifying $1/N$ of the total number of physical registers. There are two problems with this. First, this solution assumes that the free lists, and by extension the physical register files, are composed of $N$ array instances. This is not true for POWER7 and not neces-
Necessarily true in general. Second, specifying fewer physical registers would reduce the width of the register renaming table entries.

To illustrate these problems, we took this $1/N$ approach for the general purpose register file (GPR) and vector register file (VRF) in MR0 and MR1. As shown in Figure 3.4.12, the GPR area is $7 \times$ too high, but the power estimate is accurate. VRF results are similar (Figure 3.4.13): area is $3 \times$ too high, but power is more accurate. MR2 was able to achieve good area and power estimates for both the GPR and VRF by modeling them as being built with one array per pipeline, eliminating the duplication per hardware thread. In contrast to the renaming tables, physical register files are duplicated per integer/ floating-point pipeline, and McPAT models this correctly.

Control logic and arithmetic logic.

These are known to be difficult to model, especially at the architectural level [59, 101]. Since the vast majority of McPAT’s modeled structures are caches, array-based structures and CAMs, the subset area is merely 25 - 50% of the total area of the IFU, ISU, and LSU, as shown by Figure 3.4.4. McPAT does use analytical models for some control logic, such as dependency checking logic, instruction decoder, and aspects of instruction issue selection. As in Watch [15], the decoder is modeled as a $n$ to $2^n$ bit decoder, which does not represent functions like instruction cracking or group formation/instruction fusion, so area and power error is large for MR0, MR1, and MR2.
Like other analytical power models [15, 156], McPAT models the ALU [118] and FPU [125] empirically. Base power and area is obtained from published data and scaled for the activity factors, technology node, and operating point. Since these models are based on older designs from different architectures, MRo's predictions show significant inaccuracies for POWER7 (Figures 3.4.12 and 3.4.13). MR2 simply replaced the base area and power values with ones measured from detailed floorplans and microbenchmarks, respectively. As expected, area error dropped to 1% and power error to within 20%. Of course, these changes would not be appropriate for ALUs that don’t look like those in POWER7. One possible solution would be to have a library of base area and power values for different ALU designs. We discuss this problem further in Section 3.6.

**POWER7-specific details.**

A few modeling assumption errors are due to details specific for POWER7. For instance, the global completion table (GCT) tracks instruction groups by tags and other metadata instead of storing entire decoded instruction (or μop) words. Also, McPAT assumes that each entry in the instruction buffer stores peak_issue_width instructions (six in POWER7) whereas POWER7 stores
four instructions per entry. Adding the appropriate additional parameters in MR2 improved area and power estimates.

3.4.3 Input Error.

When creating MR0, we often needed to guess the values of certain parameters. These values were corrected in MR1 and made significant improvements in a few cases, but input error is not a major contributor to overall error. This can be seen in that MR1 is often only slightly different from MR0, but much different compared to MR2.

The IFU is a good example of this error category. Published data stated the I-cache was 16-way banked when it was actually 8 [163]. This cut I-cache area and power error in half for most benchmarks. The branch target buffers turned out to be 2-way set associative, which improved area errors to 5%, but power estimates are greatly underestimated due to clock- and data-gating assumptions. The I-TLB
Figure 3.4.13: Vector scalar unit breakdowns. See Section 3.4.1 for an explanation of VRF power on mcf_2k.

was also 2-way set associative instead of fully associative, but this was a rare case where fixing parameters increased both area and power error. We were unable to pinpoint the cause.

3.4.4 CODING ERRORS AND CACTI ISSUES.

There were only two instances where coding errors in McPAT caused large power modeling errors. The physical register files were intended to be modeled as shared components [103] but were actually modeled as duplicated components. Also, the register renaming tables stored 33 Bytes for each entry instead of 33 bits. Fixing these bugs in MR2 reduced register file and renaming table error significantly. Both were present in McPAT v1.2 and have been reported to the developers.

We had to manually tweak CACTI’s optimization weights in order to reduce I-cache and D-cache power and area error while staying close to the cycle time time target.
These weights represent tradeoffs between energy and delay. The presented data are the best results we were able to achieve. This is not a “bug”, but it is still worth mentioning.

Finally, we note that CACTI’s 45nm technology model is for planar bulk devices, but POWER7 is built on an SOI process. As there is no ITRS model for 45nm SOI or 32nm bulk, we ran our analysis on CACTI’s 32nm SOI model and compared it with the 45nm bulk results. Despite changing both technology feature size and device topology, average power only changed about 20-30%. We believe that this does not impact any of our conclusions given the magnitude of the errors we observed and the fact that nothing we’ve discussed thus far pertains to a specific technology.

3.5 Case Studies

Accurate power modeling is critical because many studies rely on it to evaluate other systems of interest, such as power-aware scheduling, or properties of a chip like thermal hotspots and voltage noise. We demonstrated in the previous sections that McPAT’s power and area models possess a significant amount of error unless carefully tuned for the target platform. How much power modeling error can these studies tolerate until the conclusions drawn become wrong or misguided?

In the following case studies of thermal hotspots and voltage noise, we show that error in McPAT’s dynamic power predictions only has a small effect on a chip’s steady-state properties like average chip temperature and static IR drop magnitude, suggesting that accurately modeling leakage power may be more important for these metrics. However, dynamic power error has a much larger effect on temporal properties, like the amplitude of transient inductive noise, and spatial properties, like the locations of thermal hotspots and greatest IR drop. For such properties, dynamic power error can result in even larger error in the studies, potentially leading to wrong conclusions. For these studies, it is important to carefully tune McPAT’s dynamic power models in order to obtain accurate results.
Figure 3.5.1: Steady state thermal distribution across the POWER7 chip in °C. MR0 shows much greater variance across workloads than MR2 and DPM, but MR0’s overall chip temperature is more accurate.

3.5.1 Thermal Hotspots

Heat dissipation is an important design consideration for modern microprocessors. Excessive heat can result in reduced performance as the chip tries to stay within its thermal budget, and over time, reliability of the chip can suffer. Past thermal studies have investigated temperature induced reliability degradation [165], thermal-aware task scheduling [10, 44], DVFS for mitigating thermal emergencies [48], as well as optimal floorplanning across a chip [71, 151]. Here, we examine
Figure 3.5.2: Overall chip temperature error with respect to DPM. Note that MR2 error is always negative, but absolute value error is easier to visually compare.

thermal hotspots on the POWER7 chip to quantify thermal error deriving from power error.

We constructed a full chip model using HotSpot [74], with parameters derived from the POWER7 chip and package. We selected four representative SPEC2000 workloads, simulated selected regions of each for 100 million cycles, then duplicated the resulting performance traces 40 times for a total of 4 billion cycles. Transient power was computed using MR₀, MR₂, and DPM. This single threaded power trace is duplicated across all eight cores (a multiprogrammed SPEC workload) during thermal modeling. We chose MR₀ over MR₁ because it represents the kind of model most users will have – MR₁ requires proprietary data and Section 3.4 showed that MR₀ and MR₁ are similar because the overall effect of input error is small. Some important details about our methodology are mentioned below:

- When computing chip temperature with DPM data, we use total power instead of subset power, because subset power is such a small fraction of total power that the computed temperature would be unrealistically small.

- Based on our experiments, we find that McPAT’s leakage power estimates are approximately 2× smaller than DPM’s. Leakage power is primarily a
function of technology, but accounting for technology parameters is beyond the scope of this project, so we normalize all McPAT power results by substituting its leakage numbers with those from DPM.

- DPM does not model power for the L2 and L3 cache or any uncore components. For these components, we use leakage power obtained from circuit simulations of the synthetic stressmark. This means that overall chip temperatures will be lower than what would be observed in practice.

Figure 3.5.1 demonstrates temperature characteristics consistent with the power models we have described. As a whole, MRo appears more accurate than MR2; Figure 3.5.2 shows that MRo exhibits only 5% mean error for three of the four benchmarks, while MR2’s error is consistently higher (≈10%). The exception is art, because the total core power predicted by MRo on this benchmark is dominated by the LSU. As a whole, this suggests that if leakage power is accurately modeled, McPAT can be used to predict average chip temperature well.

However, MRo invariably identifies the LSU as the thermal hotspot because its LSU power estimates are 3-6× greater than DPM’s (Figure 3.4.11). In contrast, DPM shows that no single unit within the core is primarily responsible for heat production. MR2 agrees with DPM in this regard, even if its estimates are universally much smaller because MR2 has eliminated a lot of error canceling. Furthermore, MRo exhibits significant variation across workloads, whereas MR2 and DPM are far more consistent. This is an important qualitative error that was addressed by MR2’s cumulative changes.

In summary, even though MRo can possess over 200% total core dynamic power error, it only results in 10% overall chip temperature error. Some of this error is suppressed because of accurate leakage data in the power trace and heat diffusion from the core to the surrounding uncore area. Nonetheless, we can conclude that while dynamic power modeling accuracy may not be critical for estimating average temperature, it is much more important for analyzing spatial properties.
Figure 3.5.3: Snapshot of a voltage noise trace for the three power models from gcc_2k. MR0 shows considerably greater inductive noise than DPM, whereas MR2 has much less error.

3.5.2 VOLTAGE NOISE

Voltage noise, comprised of static IR drop and transient inductive noise, is an important phenomenon in contemporary chips, because aggressive power and clock gating can produce large fluctuations in supply current that can then induce fluctuations in supply voltage. Significant voltage drops can result in timing violations for logic circuits. To mitigate effects of voltage noise, researchers have proposed various runtime strategies [60, 63, 84, 99, 144, 186], optimal placement of available C4 pads [176], and more. In this case study, we quantify the amount of error in voltage noise that derives from power error.

To evaluate voltage noise characteristics, we constructed an on-chip power distribution network and package model using VoltSpot [186], with parameters derived from the POWER7 chip and package. Because VoltSpot is a very fine grained modeling tool and the C4 pads on POWER7 are very densely packed, we only model a single chiplet - core, L2 and local L3 cache - rather than the full chip like in the thermal study. The power grid and C4 pad specifications are approximations of the actual structure and layout in POWER7. Note that although PDN parameters are derived from the physical hardware, it is beyond the scope of this work to
correlate voltage noise computed by VoltSpot with those from actual hardware.

We used the same four workloads from the thermal study and simulated representative regions of them for 40 million cycles each. Transient power was computed using MR0, MR2, and DPM. Like the previous study, we compare MR0 and MR2’s results with DPM total instead of subset because subset voltage noise is unrealistically small.

**Transient voltage behavior**

Transient voltage noise is produced when periodic current swings trigger localized \( \frac{dI}{dt} \) resonance as well as chip-wide \( LC \) resonance if the current swings occur near the global resonance frequency of the PDN. Here, we assess the accuracy of transient voltage noise predictions using power traces produced by McPAT.

Figure 3.5.3 shows a snapshot of a voltage trace from gcc that includes two distinct phases of the application with distinct transient characteristics. The snapshot demonstrates a trend in both phases that persists throughout all benchmarks: MR0’s power estimates result in huge supply voltage swings, whereas MR2’s transient voltage noise is much more muted. From Figure 3.5.4, we see that in all cases,
MRo exhibits the greatest variance in voltage noise amplitude by far. Based on the whisker heights, MRo predicts anywhere from 20 to 70% swing, whereas MR2 ranges from 8-24% and DPM from 4-14%. MR2’s maximum predictions are merely 6% Vdd higher than that of DPM, compared to 5.4% for MRo.

These results raise a question: MR2’s power estimates were tuned to match DPM \textit{subset} rather than \textit{total} power, so why does MR2 produce accurate voltage noise results when compared with DPM total power? The reason is that \textit{transient} power fluctuation, not average power, creates inductive noise. While MR2’s average power is smaller than average total power, MR2’s transient power swing amplitude is much more accurate. This case study demonstrates that for inductive noise studies, accurately capturing transient power, rather than average power, is much more significant.

\section*{Static IR Drop}

Static IR drop is caused by the impedance of the power delivery network, and in today’s systems, the primary solution is the use of a voltage guardband. IR drop is effectively static on the time scale of processor activity due to the presence of the power grid itself and decoupling capacitance, but sustained nonuniform activity and current draw can create significant variations across a chip. We did analyze IR drop for POWER7, but we found the exact same conclusions as we did for thermal hotspots: MRo is slightly more accurate than MR2 for overall IR drop but much worse for spatial properties. Therefore, we omit the data for brevity.

\subsection*{3.6 Discussion and Guidelines}

Despite the amount of existing work using McPAT, few (if any) have mentioned modeling inaccuracies like the ones we describe. One possible reason is that many past studies have used cores that did not trigger some errors we observed. For example, older and simpler cores like Atom and Penryn have lower issue widths, so studies using them [50, 61, 149, 176, 186] avoid read/write port overestimates, one of the major error sources we observed. Penryn cores also do not support
SMT, eliminating the duplication of hardware error. This being said, POWER7 is not unusual from a power modeling perspective. Table 3.6.1 shows the high-level microarchitectural parameters that define three different 45nm server-class processor cores. From such a view, there is no fundamental reason why one should not use McPAT to model a POWER7-like chip. Furthermore, server-class cores, like Haswell, are becoming more complex in order to meet single-thread growth targets while simultaneously attaining lower power budgets for mobile applications. Therefore, without knowledge about the modeling gaps in McPAT, power modeling studies using this tool are likely to become progressively more inaccurate over time.

As mentioned in Section 3.2, one method of analytically modeling hard-to-model control logic is to assume that its power is correlated with that of a cache structure and add a fudge factor to compensate. McPAT does not use this method in that it does not have explicit fudge factors to account for missing control logic models.\(^3\) It is unfair to argue that these fudge factors are implicit through the subset (caches/arrays) overestimates because CACTI is not meant for logic modeling.

We investigated correlation between subset and nonsubset power on POWER7

\(^3\)The only exception to this is that McPAT estimates pipeline latch, common data bus, and layout overhead with fudge factors.
as a first step towards creating MR3. For this analysis, we used power traces from four SPEC2006 benchmarks. As shown by Figure 3.6.1, the covariance between the power traces is close to +1 for all four benchmarks, indicating that subset and nonsubset power are highly correlated and that the fudge factor method does have merit. However, this method is difficult to use because these fudge factors likely require RTL simulations to ascertain. Indeed, Figure 3.6.1 shows that the ratio of subset to nonsubset power varies as much as 0.4 between units and up to 0.3 within a unit between benchmarks. The error bars denote standard deviation of the ratio over the time series power trace, so variation of up to 0.3 exists even during a single benchmark’s execution.

Based on our power model error analysis, we suggest the following solutions for improving power model accuracy:

1. **Abstraction error**: Users of McPAT must specify important parameters like read/write ports as accurately as possible with the available data. In general, we need to build more detailed models of microarchitectural structures. Alternatively, having more collective experience for generalizing fudge factors for key units would be valuable.

2. **Modeling assumption error**: Users of McPAT should take care to correctly model shared resources for SMT when appropriate. Modeling of control and arithmetic logic is difficult at the architectural level, but we believe there is potential in building semi-empirical models by characterizing hardware from open source projects like RISC V [177], FabScalar [32], chip generators [56], and OpenSPARC [154].

3. **Input error**: Users should carefully specify modeling parameters, but it is comforting to find that in our study, input error was not a big contributor to overall power error.

In addition, we emphasize two guidelines about how power models can and should be validated when tools like DPM are not available:
Figure 3.6.1: Covariance and ratio between subset and nonsubset components on four SPEC2006 benchmarks. The FXU and VSU were excluded because their subsets cover over 80% of the logic.

1. **Validate at unit level using measured power.** We demonstrated that validating a power model at the core or chip level hides a large amount of internal error at the unit level and below. Targeted microbenchmarking is a well-known technique for characterizing fine-grained power [9, 85, 152].

2. **Validate leakage power.** Leakage is primarily a function of area, technology, and voltage/frequency, so it can be captured by more detailed analytical models and/or circuit simulations. Power-gating factors are important too, but researchers will usually have to settle for educated guesses.

Finally, the academic community would greatly benefit from the availability of validated power models for contemporary commercial chips and/or assistance with power/performance validation studies as described in this chapter. There have been some industrial performance and power simulators that have been re-
leased, like Turandot [123] and PowerTimer [16] from IBM and XTREM from Intel [39], but many of these tools were designed for processors that are greatly outdated by current standards, and the release of updated core models would be extremely helpful. The power models developed in this project can be downloaded at http://vlsiarch.eecs.harvard.edu/mcpat. Note that these models should only be used for core dynamic power analysis, not for uncore or leakage studies.

3.7 Next Steps

This project taught us many lessons about how to build accurate power modeling tools for complex systems. However, McPAT is mostly used for general-purpose multicore processors, not for accelerator-centric SoCs. In the next chapter, we show how we build an end-to-end SoC simulator that supports both performance and power modeling for complex heterogeneous SoCs, enabling a new field of holistic SoC-accelerator co-design research.
Enabling SoC-Accelerator Co-Design using gem5-Aladdin

In surveying the field of accelerator research, we observed that most of the research focuses on the design of the accelerator’s core datapaths and local memory systems. Comparatively little attention is given to concerns about integration into an SoC. This project makes the case that system-level effects like data movement, coherency management, and shared resource competition are significant yet often unaccounted components of total accelerator runtime. To enable system-level accelerator research, we build gem5-Aladdin, an SoC simulator that can model complex accelerator-system interactions. gem5-Aladdin is validated against real hardware to less than 6% error. We find that with co-design, we can improve the optimal energy-delay product (EDP) of accelerators by up to $7.4 \times$ compared to that of accelerators optimized in isolation.
4.1 Introduction

Accelerators are often designed as standalone IP blocks that communicate with the rest of the system using a Direct Memory Access (DMA) interface. This modularity simplifies IP design and integration with the rest of the system, leaving tasks like data movement and coherency management to software device drivers. As a result, the costs of these overheads are hard to predict and accommodate for at accelerator design time. Our detailed characterization of accelerator behavior shows that the combination of just these two effects can occupy over 40% of the total runtime. Hence, when it comes to accelerator design, architects must take a holistic view of how they interact in the overall system, rather than designing them in isolation.

Fundamentally, all systems should be designed in a way that balances the bandwidth of the memory interface with the amount of compute throughput. An overly aggressive design will have more computational units than the memory interface can supply, leading to wasted hardware and additional leakage power. We identify three major system-level considerations that strongly affect accelerator design: local memory interface, cache coherency management, and behavior under shared resource contention.

The typical local memory interface is DMA, a push-based system that requires software to setup bulk transfers and manage coherency. An alternative is to embed a hardware-managed cache with the accelerator design, leading to a fine-grained, pull-based memory system that loads data on-demand and transparently handles coherency state. Despite these conveniences, caches are rarely used in accelerators due to hardware overheads leading to power and area penalties. However, there has been growing interest from industry in providing coherent accelerator cache interfaces [18, 127, 168] for the ease of programmability. We investigate the system-level considerations for both approaches to understand when each is preferable.

Such studies require detailed simulation infrastructure for heterogeneous accelerator-rich platforms like SoCs. There is a wide selection of CPU simulators [12, 24, 91] and standalone accelerator simulators like Aladdin [156]. However,
existing SoC simulators are unable to model dynamic interactions between accelerators and the memory system [37]. In this chapter, we introduce gem5-aladdin, which integrates the gem5 system simulator with the Aladdin accelerator simulator to enable simulation of SoCs with complex accelerator-system interactions. We validate gem5-aladdin against the Xilinx Zynq platform and achieve less than 6% error.

We demonstrate that co-designing accelerators with system-level considerations has two major ramifications for accelerator microarchitectures that are not yet fully understood in the literature. First, datapaths should be less aggressively parallel, which results in more balanced designs and improved energy efficiency compared to accelerators designed in isolation. Second, the choice of local memory interfaces is highly dependent on the dynamic memory characteristics of the accelerated workload, the system architecture, and the desired power/performance targets. We show that accelerator-system co-design can improve energy-delay-product by up to 7.4× and on average 2.2×.

4.2 Motivation and Background

In this chapter, we use the term “accelerator” to refer to an application-specific hardware block. These accelerators are comprised of multiple customized datapath lanes, and customized local memories. Each lane is a chain of functional units controlled by finite state machines. When the local memory is comprised of scratchpads, each scratchpad can be partitioned into smaller arrays to increase memory bandwidth to the lanes. Such accelerators are representative of recent academic proposals [33, 50, 66, 107, 138, 174, 182] and commercial designs [13, 78, 96].

4.2.1 Co-design: A Motivating Example

To demonstrate the differences between isolated vs. co-designed accelerators, we perform a design sweep exploration for both scenarios on a 3D stencil kernel. We sweep compute parallelism and scratchpad partitioning. Compute parallelism is
Figure 4.2.1: Design space exploration for stencil3d for both isolated and co-designed cases.

described by the number of datapath lanes. Figure 4.2.1 shows these two design spaces.

We consider an accelerator designed in isolation to be one that focuses design optimization on the computation phase. This design space (blue circles) leans towards more parallel, power-hungry designs, as exemplified by the isolated energy-delay-product (EDP) optimal design point. But if we account for effects like initial data movement, the design space (green triangles) shifts dramatically towards the lower right, preferring less parallel designs at lower power. If we take the isolated EDP optimal design and then apply these system effects, we find that it is quite different from the co-designed EDP optimal point. Unaccounted data movement becomes a significant part of total runtime, making aggressively parallel datapaths unnecessary.
Figure 4.2.2: Data movement overheads on MachSuite.

4.2.2 Typical CPU-Accelerator Communication

The existence of the difference between the two design spaces is due to how CPUs and accelerators traditionally communicate data. In this typical flow, DMA is the transfer mechanism, but typical DMA implementations can only access main memory or LLC, so the CPU first flushes all input data from private caches and invalidates the region used to store return data [183]. Then it programs a DMA transaction into the DMA engine and initiates the transfer. The accelerator begins execution after receiving all the data and streams its output data via DMA back to main memory when it is done. The CPU, having invalidated that memory region
Figure 4.2.3: An example SoC that can be modeled using gem5-Aladdin. The table on the right shows the set of design parameters that we swept in this work and their values; this is just a small subset of what can be configured.

From its caches, can now access the return data correctly.

For many benchmarks, this flow works quite well. DMA is quite efficient at copying large blocks of data, and accelerators whose compute-to-memory ratios are large are well served by DMA. However, for other workloads with more irregular memory access patterns, this flow can impose severe overheads, because the accelerator must wait to receive all the data before it can begin computation. As an example, Figure 4.2.2a shows the execution timeline for a 16-lane implementation of an m<sub>cd</sub>–knn accelerator (a k-nearest-neighbor molecular dynamics), running on a Xilinx Zynq platform. As shown, the accelerator’s computation only occupies about 25% of the total cycles, with the rest of the time spent on preparing and moving data. We expanded this study in simulation for all the MachSuite benchmarks and find that about half of them are compute-bound and the other half data-movement-bound, as shown in Figure 4.2.2b.

Clearly, DMA is not an optimal solution for some workloads. One alternative, as mentioned earlier, is to replace push-based DMA with pull-based hardware-managed caches. In recent years, the scope of workloads that we desire to acceler-
ate has widened from dense computational kernels to more irregular applications which could benefit from a less rigid memory system. Although caches have seldom been used for accelerators, the increased workload diversity motivates a more comprehensive study of new CPU-accelerator communication strategies.

4.3 Modeling Infrastructure

Figure 4.2.3 shows an example of an SoC, including general-purpose cores, memory controllers, a DMA engine, and different types of fixed-function accelerators, all of which are connected through the system bus. In order to understand how system-level effects impact the behavior of accelerators, we need simulation infrastructures that can model these heterogeneous systems. In this chapter, we integrate Aladdin with the gem5 system simulator [12], a widely-used system simulator with configurable CPUs and memory systems.

gem5-aladdin models interactions between accelerators and CPUs, DMA, hardware-managed caches, and virtual memory. All of these features have implications on how the accelerator behaves and in the following sections, we describe how each is modeled.

4.3.1 Overview

For the experiments in this chapter, we run gem5-aladdin in syscall emulation mode because it is sufficient to capture the effects of our system-level considerations on performance and power. Full-system simulation would enable us to model operating system effects, but most are beyond the scope of this study. Some interactions with the operating system, such as device driver to hardware interactions, are characterized through real hardware measurements and analytically included in our models. Finally, syscall emulation is much faster than full system simulation, easing rapid design space exploration.
4.3.2 Accelerator Modeling

The Aladdin accelerator simulator \cite{156} takes a first step towards modeling the power, performance, and cycle-level activity of standalone, fixed-function accelerators without needing to generate RTL. Aladdin is a trace-based accelerator simulator that profiles the dynamic execution of a program and constructs a dynamic data dependence graph (DDDG) as a dataflow representation of an accelerator. The vertices in the DDDG are LLVM IR instructions, and the edges represent true dependences between operations. Aladdin then applies common accelerator design optimizations and schedules the graph for execution through a breadth-first traversal, while accounting for user-defined hardware constraints. Aladdin was validated to be within 7\% accuracy compared to standalone, RTL accelerator designs.

However, Aladdin only focuses on the standalone datapath and local memories. It assumes that all data has been pre-loaded into the local scratchpads. This skips the modeling of any interactions between accelerators and the rest of the system in which they belong.

4.3.3 DMA Engine

DMA is a software managed mechanism for transferring bulk data without CPU intervention. To set up a transaction, the programmer constructs a DMA transfer descriptor that contains the source and destination memory addresses along with the size of the transfer. Multiple descriptors can be constructed and connected through a linked list. When all descriptors are ready, the programmer initiates the transfer by writing the address of the head of the descriptor linked list into a hardware DMA engine’s control register. The DMA engine then fetches and services these descriptors one by one. Meanwhile, the CPU is free to perform other work.

In gem5-Aladdin, accelerators can invoke the DMA engine already present in gem5. To do so, a programmer inserts calls to special \texttt{dmaLoad} and \texttt{dmaStore} inside the accelerated function with the appropriate source, destination, and size arguments. When the function is traced by Aladdin, Aladdin will identify these calls as DMA operations and issue the request to the gem5 DMA engine. As part
of the DMA engine, we include an analytical model to account for cache flush and invalidation latency, using the measured numbers mentioned in Section 4.4.2.

4.3.4 Caches and Virtual Memory

For the accelerator caches, we use gem5’s classic cache model along with a basic MOESI cache coherence protocol. When Aladdin sees a memory access that is mapped to a cache, it sends a request through a cache port to its local cache. Aladdin will receive a callback from the cache hierarchy when the request is completed. To support virtual memory, we implement a special Aladdin TLB model. We do not use gem5’s existing TLB models for two reasons. First, the existing TLB models are tied to particular ISAs, which do not pertain to accelerators [155]. Second, as a trace-driven simulator, the trace address that Aladdin originally uses does not directly map to the simulated address space that CPU is accessing. To maintain correct memory access behavior, our custom TLB model translates the trace address to a simulated virtual memory address and then to a simulated physical address. TLB misses and page table walks are modeled with a pre-characterized miss penalty.

4.3.5 CPU-Accelerator Interface

On the CPU, a simulated user program can invoke an attached accelerator through the ioctl system call, a system call widely used in practice for arbitrary communication with devices. In the ioctl emulation code, we assign a special file descriptor value for Aladdin and use command numbers to refer to individual accelerators. When the accelerator finishes, it writes to a shared pointer between the CPU and the accelerator. The CPU will see the update due to cache coherence. After invoking the accelerator, the CPU can either spin wait for the status to update or continue to do other work, periodically checking the status variable to see if the accelerator is completed.

Sharing virtual memory between CPUs and accelerators means that any mismatches in memory consistency models must be resolved. In our experiments, we
Figure 4.3.1: Error between Zedboard and gem5-Aladdin cycles.

handle this by strictly limiting pages of memory that the accelerator may access and enforcing mutual exclusion on these pages. For our simple synchronization primitive, the accelerator issues an mfence before signaling to the CPU that it is finished through the shared pointer.

4.3.6 Performance Validation

We have validated gem5-Aladdin’s performance models using the Zynq Zedboard for a subset of the MachSuite benchmark suite. For each benchmark, we implement the AXI4-Stream interface to transfer data via Xilinx’s DMA IP blocks. Accelerator RTL is generated using Vivado HLS 2015.1. To maintain a consistent view of the model, we use HLS without specifying any additional design optimizations, so Vivado HLS generates a default design whose parameters we then match in Aladdin.

The complete system (including the DMA engine, accelerators, crossbars, etc.) is implemented in in Vivado Design Suite 2015.1. Software running on the CPU first initializes all devices in the system and generates the accelerator input data. Then it performs the necessary cache flushes and invalidates and starts the DMA transfer. The accelerator automatically begins computation when the DMA transfer is complete.
To measure performance, we instrument this code using cycle counters on the A9 CPUs. Because we cannot directly measure the DMA transfer time, we include logic analyzers in the synthesized system to capture waveforms using Xilinx tools during execution. Most benchmarks were implemented on a 10ns clock; a few used slower clocks for timing reasons.

The results of our validation are shown in Figure 4.3.1. Our DMA performance model achieves 6% average error across this suite of benchmarks, while Aladdin achieves 5% average error, and the flush and invalidation analytical model achieves 5% average error. These results demonstrate the ability of gem5-Aladdin to model a wide range of accelerator workloads accurately for both the accelerated kernels and important system-level considerations.

Validation omissions

Our validation focuses on the features required by our DMA techniques: cache flushes and invalidates, DMA transfer time, and accelerator runtime. In general, we validated as much of the new additions as we could. Below are the components this work does not validate and our reasons for omitting them.

- CPU performance models: Existing work by Gutierrez et al. has already produced an accurate gem5 CPU model for the ARM A9 core[65], and gem5-Aladdin uses that validated model.

- Power model: All power results represent only the accelerator power. We do not account for CPU power in any of our results. We use the same validated Aladdin's power models with TSMC 40nm technology.

- Cache: To the best of our knowledge, there is no existing IP block available on Zynq such that we could implement a cache controller on the programmable fabric. Furthermore, we never modified gem5’s cache models.
4.4 MEMORY SYSTEM OPPORTUNITIES

In this section, we will discuss the primary design considerations when deciding whether to use a DMA- or cache-based memory system for an accelerator. Because baseline DMA leaves much room for improvement, we will also apply two optimizations to DMA. We will then describe design considerations specific to cache-based accelerators. Finally, we will evaluate the performance of both memory systems for a set of representative benchmarks.

4.4.1 PRIMARY DESIGN CONSIDERATIONS

First, we compare and contrast DMA and caches across the three system-level considerations mentioned earlier: push vs. pull, data movement granularity, management of coherency, and behavior under shared resource contention.

Push vs. Pull: DMA is designed for efficient bulk data transfer where the data requirements of the program are well known a priori. This works well for streaming applications and applications with high compute-to-memory ratios. However, applications with more irregular memory access patterns, such as indirect memory accesses, can suffer without an on-demand memory system like a cache. In addition, because caches have the feature of automatic cache line replacement, a cache can often afford to be smaller than a scratchpad that must hold all the data.

Data Movement Granularity: Because DMA is software controlled, the overheads of setting up a transaction are usually amortized over a large bulk transfer. In contrast, caches pull in data at cache line granularity, enabling fine-grained overlap between compute and data movement. Although fine-grained DMA is possible, each new transactions adds additional overheads. On the other hand, caches must perform tag comparisons, replacements, and address translations, which make them inefficient for bulk data movement.

Cache Coherence Management: DMA engines typically can only access main memory or last level cache. Therefore, the programmer must conservatively flush any data the accelerator may read out of private caches. Figure 4.2.2b shows that on average, accelerators employing traditional DMA spend 20% of their total
cycles on cache flushes. The flush is typically performed by software because DMA engines rarely participate in coherency (although there have been exceptions, like IBM Cell [94]). In contrast, hardware-managed caches handle all of this complexity transparently at the cost of additional hardware.

**Shared Resource Contention:** In a real scenario where resources like the main system interconnect and main memory are shared across multiple agents, invariably a DMA operation or cache fill will stall to allow another process to make progress. A coarse-grained mechanism like DMA will be affected much more by shared resource contention because the accelerator usually waits for the entire transfer to complete. In comparison, fine-grained memory accesses like cache fills are less likely to contend due to their smaller size, and hit-under-miss allows other independent operations to proceed even while an earlier cache load or store missed.

### 4.4.2 DMA Optimizations

In this section, we improve the baseline DMA method by overlapping various stages of the process. We will examine two DMA latency optimizations: pipelined DMA and DMA-triggered computation, which are depicted in Figure 4.4.1.

**Pipelined DMA**

Pipelined DMA reduces latency by dividing the flush and DMA operations into page sized blocks and overlapping the DMA of block \(b\) with the flush of block \(b + 1\). We choose page size granularity to optimize for DRAM row buffer hits. In the best case, we can hide all but 4KB of the flush latency. Note that the correctness of this optimization is ensured by never starting a DMA block before its flush has completed.

Cache line flush latency varies across ISAs and implementations. For example, we characterized the flush throughput on the Zedboard’s Cortex A9 CPU to be one cache line per 56 cycles at 667MHz. To achieve optimal pipelining and avoid bubbles, we want to match the flush and DMA latencies of a 4KB transaction. On
the Zedboard, this is achieved with an accelerator clock frequency of 100MHz, which is why we use this frequency for the rest of our experiments.

Breaking up a large flush and DMA operation introduces additional overheads. The DMA engine must fetch new metadata from main memory for every block, and the CPU must synchronize flushes with dependent DMA operations. For this, we add a fixed 40 cycle delay to every DMA transaction, also based on characterization. At 100MHz, this accounts for metadata reads (4 cycles), the one-way latency of initiating DMA from the CPU (17 cycles), and additional CPU cycles spent on housekeeping actions.

**DMA-Triggered Computation**

Even with pipelined DMA, the accelerator still must wait for the entire DMA transaction to finish before it can start. To overcome this, we augment our accelerators with full/empty-bits, which are often used in producer-consumer situations to in-
dicate that data is ready [115]. In our designs, we track data at cache line granularity to be consistent with the preceding flush operations (which operate on cache lines). Full/empty bits are stored in a separate SRAM structure and indexed by a slice of the load address. With full/empty bits, the accelerator immediately begins computation without waiting for DMA to complete until it reaches a load. A load accesses both the full/empty bit arrays and the data arrays in parallel and returns the data if the full/empty bit is 1. If not, the control logic stalls the datapath until the DMA engine eventually fills that data and sets the full/empty bit. Note that double-buffering could be implemented in this scheme by tracking the granularity of data transfer at half the array size instead of cache line size, without any manual intervention. If an accelerator has multiple datapath lanes, other lanes are free to proceed even while some are blocked.

4.4.3 DMA Evaluation

To quantify the performance improvements from each of the techniques described, we start from the baseline design and cumulatively apply our DMA optimizations. From execution traces, we break down the runtime into four parts based on how cycles are spent: flush-only time, DMA/flush time, compute/DMA time, and compute-only time. Flush-only and compute-only are self-explanatory; compute/DMA time includes all cycles when compute and DMA are overlapped, while DMA/flush includes all cycles when DMA and flush but not compute are running.

Increasing the parallelism of accelerator datapaths through additional datapath lanes and memory partitioning is a widely used and effective way to achieve higher performance at the cost of greater area and power. However, the presence of memory movement imposes an upper bound on achievable speedup, and our DMA optimizations will affect realized speedup as well. To understand how parallel an accelerator must be in order to approach this upper bound, we take all the optimizations, sweep the parallelism of the accelerator datapath, and analyze the speedups realized.
Performance gains from DMA optimizations

The performance improvements from each optimization are shown in 4.4.2a. For brevity, we only present a subset of benchmarks whose DMA times spans the range shown in Figure 4.2.2b. We fix the parallelism of all accelerators to four datapath lanes.

We immediately observe that in the baseline design, flush-only time is a significant fraction of the total execution time. Pipelined DMA is thus shown to be very effective, almost completely eliminating flush-only time for all the benchmarks shown. This is because the benefits of pipelined DMA are only dependent on the amount of data transferred and not on the memory characteristics of the application.

DMA-triggered computation is able to improve performance even more, but its effectiveness clearly varies across workloads. It is most effective when a benchmark exhibits some level of streaming behavior. For example, stenc112d uses a 3x3 kernel and thus only requires the first three rows of the input matrix to arrive before it can start computation, so ready bits recover a significant amount of performance. A similar logic applies to md-knn – in fact, ready bits are so effective here that with just four datapath lanes, we achieve 99% compute/DMA overlap. This is in contrast to fft-transpose, where each unit of work requires eight loads strided across the entire input arrays. This is not a streaming memory access pattern and so DMA-triggered compute is ineffective.

Impact of parallelism on DMA optimizations

The results of sweeping accelerator parallelism, while applying all the DMA optimizations, is shown in Figure 4.4.2b. This figure demonstrates two points.

First, on several workloads, if there is enough parallelism, the entire computation can be overlapped with DMA. This means that without reducing flush or DMA time, no more speedup is achievable. Benchmarks without this property either have very little data to transfer to begin with (aes) or are so serial that they don’t benefit from data parallelism in the first place (mv).
(a) Performance improvements from each technique.

(b) Effect of parallelism on performance gains.

Figure 4.4.2: Cumulatively applying each technique reduces the additional cycles spent on DMA, with some benchmarks seeing more benefit than others. After applying all techniques, increasing parallelism through loop unrolling reduces compute cycles until near-complete overlap is achieved, causing performance to saturate.

Second, increased parallelism has no effect on the amount of compute-DMA overlap. This is due to the serial data arrival effect: no matter how parallel a datapath is, DMA will always copy data sequentially starting with the first byte, and until that critical first byte of data arrives, no compute can start. As our DMA engine already fully utilizes the available bus bandwidth, this data cannot arrive any faster, and therefore compute also cannot be overlapped any more.

In conclusion, these sweeps show that memory movement, not compute, has become a significant bottleneck, and only accelerating computation will quickly bring diminishing returns. In fact, Figure 4.4.2b shows that for many benchmarks, we can achieve the upper bound performance with relatively fewer datapath lanes. As a result, to continue to get better performance, we must somehow further overlap computation with data by overcoming the serial data arrival effect, motivating the study of fine-grained, on-demand memory systems.
4.4.4 Cache-Based Accelerators

In a cache-based accelerator, one of the most important questions is how to handle variable latency memory accesses in a statically scheduled datapath. The simplest way is to stall the entire datapath until the miss resolves, but this significantly hurts performance. Techniques like multithreaded accelerators have been proposed in the CAD community to hide cache miss latency [75, 171], but these require additional resources to store thread contexts.

We choose a simpler cache miss handling scheme. Accelerators are typically designed with multiple parallel lanes. When a cache miss happens in one of the lanes, only that lane is stalled until the miss resolves. Other lanes are free to continue. We include MSHRs to enable hit-under-miss and multiple outstanding misses. Any lane with a dependence on a result from a blocked lane is also blocked via control logic mechanisms. This scheme lets independent computation proceed while waiting for the missed data to be returned without requiring storage for thread contexts. When lanes are finished executing, they must wait and synchronize with all other lanes before the next iteration can begin.

Another important design choice is what data is cached. In our experiments, only data that must be eventually shared with the rest of the system is sent through the cache, and local scratchpads are used for private intermediate data. For example, nw uses an internal score matrix to align DNA sequences. This matrix is kept in local scratchpads.

4.4.5 Cache Evaluation

In this section, we will analyze the impact of datapath parallelism on cache-based accelerator performance. We decompose total execution time into processing time, latency time, and memory bandwidth time, using a similar technique as Burger et al. [20]. Each component is the additional execution time after applying a realistic constraint to a memory system parameter. To briefly summarize:

1. Processing time: assume memory accesses are single-cycle and always hit.
2. Latency time: allow memory accesses to miss in the cache, but the system bus has unlimited bandwidth to service cache fills.

3. Bandwidth time: constrain the system bus width to 32 bits, thus limiting the rate at which cache fill requests can be serviced.

**Impact of Datapath Parallelism**

Figure 4.4.3 shows how the performance of cache-based accelerators scales with datapath parallelism. In this set of experiments, we first sweep cache sizes to find the smallest cache at which performance saturates for each benchmark. This is labeled at the top of each group of bars. The datapath parallelism sweep is performed with this cache size per benchmark.

Naturally, we observe that processing time decreases with increased parallelism, as expected. However, parallelism also improves latency time, which is in contrast to the DMA experiments where parallelism did not affect flush or DMA time. This
is because caches are a fine-grained pull-based memory system, and increased dat
apath parallelism also increases memory-level parallelism (more memory accesses per cycle). Furthermore, the fine granularity more effectively masks cache miss lat
tency with computation, thereby decreasing latency time.

On the other hand, more parallelism does not improve bandwidth time due to increased memory bandwidth pressure. In fact, bandwidth time becomes a larger
er fraction of total execution time as we go to increasingly parallel designs. For ex
ample, the performance of \texttt{spmv-cr5} and \texttt{md-knn} is eventually bottlenecked by bandwidth, even though the increased memory level parallelism improves both
processing and latency time. Accelerators that are designed without considera
tion of the available memory bandwidth in the SoC are likely to be over-designed,
provisioning more functional units than can be fed with data by the system.

4.5 Accelerator Design Choices

Thus far, we have discussed in detail how the performance of accelerated work
loads changes when connected to two different memory systems, scratchpad with
DMA and hardware-managed caches. However, it has been unclear when to select
one over the other. Performance is not the only goal as well; accelerator designers especially must balance performance targets against power and energy constraints. It is also unclear how differently one must think about designing accelerators when system-level effects like data movement and its mechanisms are considered. In this section, we will shed light on the DMA vs. cache question as well as illustrate that without consideration for data movement, accelerator designers are highly to over-provision and underutilize their accelerators.

4.5.1 DMA vs. Caches

One of the earliest decisions a designer needs to make is decide whether private scratchpads with DMA or hardware-managed caches is a better fit for the application at hand. In this experiment, we performed a comprehensive design space sweep for all the parameters listed in Figure 4.2.3 for all of the MachSuite benchmarks. We show the resulting Pareto optimal design curves, distinguished by memory system type, in Figure 4.4.4. For brevity, we only show eight benchmarks that span the range of design space characteristics observed. The energy-delay-product (EDP) optimal design point for each memory system is labeled with a star of the corresponding color. All DMA design points apply all the optimizations discussed in Section 4.4.2.

This experiment shows that some benchmarks unambiguously prefer scratchpads with DMA (on the left), some clearly are better with caches (on the right), and several work equally well with either (in the middle). We will briefly discuss each benchmark’s behavior in turn.

aes-aes and nW-nW:

These two benchmarks always both perform better and use less power with DMA than with caches. They have have very regular access patterns, and importantly, they only require a small amount of data before computation can be triggered. In contrast, a cache-based memory system will first experience a TLB miss followed by cache misses, causing significant performance slowdown.
**Figure 4.5.1: Comparison of accelerator microarchitectural parameters across four design scenarios.** The vertices of the Kiviat plots represent the number of datapath lanes, SRAM sizes, and local memory bandwidth, normalized to the isolated optimal design, shown on the upper-left corner, for each benchmark.

**gemm-ncubed:**

This benchmark, unlike the previous two, is actually able to match its DMA counterpart in performance. However, due to the various overheads of caches (tag lookups, TLB lookups, etc.), more power must be expended to reach this performance.

**stencil-stencil2d:**

Although DMA can always outperform the cache system on this benchmark, a cache-based design can actually achieve same performance with lower power. This is because the cache system can capture enough locality to use a smaller cache, whereas the scratchpad design must fit the entire data set into local memory.

**stencil-stencil3d:**

70
The 3D stencil kernel distinguishes itself from its 2D counterpart because the cache system can outperform the optimized DMA system at the cost of additional power. This is because the kernel’s three-dimensional memory access pattern creates nonuniform stride lengths, which are gracefully handled by the on-demand nature of a cache. In contrast, even the most optimized DMA design spends half of its execution time waiting for DMA and flush operations. The cost of this performance is \(2\times\) to \(3\times\) increased power.

**md-knn:**

*md-knn* is a very compute intensive application. In this benchmark, there are 12 FP multiplies per atom-to-atom interaction, so the power consumption of this benchmark is dominated by functional units rather than memory. Also, the optimized DMA system is able to fully overlap compute with data movement because full/empty bits are very effective in this benchmark. Figure 4.4.4 shows that the Pareto curves for cache and DMA designs largely overlap, demonstrating that either memory system can be an acceptable choice.

**spmv-crs:**

On this benchmark, a cache system is able to outperform a DMA system with lower power as well. This is due to the indirect memory accesses inherent to sparse matrix multiply algorithms, where the first set of loads provide the memory addresses for the next set that actually returns the data. Full/empty bits may not be effective on this benchmark if the data pointed to by a matrix index has not yet arrived, since DMA sends data sequentially, but a cache can fetch arbitrary memory locations. Caches thus eliminate most of the idling time, leading to better performance. Lower power on caches is achieved by being able to use a smaller cache than the scratchpads.

**fft-transpose:**

*fft-transpose* also performs better with caches than DMA but for slightly different reasons. There are no indirect memory accesses in this benchmark. Instead, the parallel implementation of this benchmark possesses a stride length of
512 bytes, meaning that each loop iteration (aka datapath lane) only reads eight bytes per 512 bytes of data. As a result, even with full/empty bits, a DMA system must supply nearly all of the data before the computation can begin, whereas this is not a problem for the cache system. Again, lower power is achieved by a smaller cache than scratchpads.

4.5.2 Design Decision Comparison

In addition to deciding the type of memory system to use, accelerator designers must also select local design parameters like the datapath parallelism and local memory size and bandwidth. In this section, we show that when system-level effects are considered, these parameters can change considerably compared to when an accelerator is designed in isolation.

To illustrate how optimal design parameters are affected by system-level effects, we consider the following design scenarios:

1. Baseline: design accelerators in isolation.

2. Co-designed DMA: use DMA to transport data over a 32-bit system bus.

3. Co-designed cache: use a hardware-managed cache for the accelerator’s local memory.

4. Co-designed cache with 64-bit bus: Same as above, but we double the width of the system bus.

We focus our comparisons on three accelerator microarchitectural parameters: datapath lanes, local SRAM/cache size, and local memory bandwidth to datapath lanes. As before, we select the EDP optimal points from each design scenario for comparison.

Figure 4.5.1 shows the differences in these three dimensions for each benchmark under the four design scenarios. For each benchmark, the triangle on the upper-left corner shows microarchitecture parameters for isolated optimal designs. The colored triangles, in turn, represent optimal design choices for DMA
with 32-bit bus, cache with 32-bit bus, and cache with 64-bit bus. To show differences between isolated optimal and co-designed optimal choices, we normalize all the designs to the design parameters of the isolated design.

**Isolated vs Co-Designed Microarchitecture**

It is immediately apparent that accelerators designed in isolation over-provision accelerator resources. In Figure 4.5.1, almost every colored triangle is smaller than the baseline triangle, showing that isolated designs tend to over-provision computational resources, and more balanced designs can be found by accounting for system-level effects.

This over-design is most pronounced in local memory bandwidth and SRAM size for cache-based designs. Isolated designs attempt to parallelize computation as much as possible, requiring very high internal memory bandwidth, but in a more realistic environment, the need to move data from system to accelerator imposes a upper bound on performance that makes internal memory-level parallelism less critical. For example, on spmv-cr and md-knn, both DMA- and cache-based designs require much lower local memory bandwidth than the isolated design. In addition, because caches have the feature of automatic data replacement, they can be sized smaller than scratchpads which must hold all the data, resulting in energy improvements.

In general, caches tend to prefer more parallel datapaths than DMA, as shown in md-knn and fft-transpose, since their fine-grained nature allows more parallel memory accesses. In fact, gemm-nibled an example where a co-designed cache-based accelerator is more parallel than both the isolated design and a DMA-based one.

**Impact of System Bus Bandwidth**

As a proxy for resource contention in a loaded system, we vary the system bus width to modulate the bus bandwidth available to accelerators. If we compare accelerators designed with a 64-bit bus to those designed with a 32-bit bus (or-
Figure 4.5.2: EDP improvement of co-designed accelerators in different scenarios, normalized to EDP of isolated designs. The design parameters of each optimal design point are illustrated in Figure 4.5.1.

ange and red triangles in Figure 4.5.1, respectively), we see that accelerators designed with lower bus bandwidth tend to provision fewer datapath lanes (md-knn, spmv-crs) and local memory bandwidth (nw, stencil2d, and spmv-crs). These effects happen for the same reasons co-designed accelerators are leaner than isolated accelerators.

EDP IMPROVEMENT

Figure 4.5.2 shows the improvements in EDP when accelerators are co-designed, compared to how an accelerator designed in isolation would behave under a more realistic system. This is the same analysis as Figure 4.2.1, but applied to more benchmarks and three different design scenarios. Overall, average EDP improves by 1.2×, 2.2×, and 2.0× for accelerators with DMA, caches with 32-bit system bus, and caches with a 64-bit bus, respectively.

The EDP improvements for co-designed cache-based accelerators is higher than that for DMA-based accelerators because an overly aggressive design for a cache-
based accelerator results in a large, highly multi-ported cache, which are much more expensive to implement than partitioned scratchpads. Furthermore, we see that on average, improvements are greater for cache-based accelerators with a 32-bit system bus than a 64-bit bus. In other words, co-design is even more important for contended systems than uncontended systems.

4.6 Next Steps

In this chapter, we argued that a holistic approach to accelerator design must account for the system-level effects and presented an SoC simulator that can model dynamic interactions between the SoC and accelerators. Our case studies exploring datapath parallelism and local memory systems is merely a first step in co-design opportunities. In the next chapter, we will show how a multi-accelerator system can leverage SoC interfaces and existing resources to improve overall performance and efficiency without requiring any changes to accelerator datapaths or local memory systems.
5

Optimizing Multi-Accelerator Systems via SoC Interfaces

This chapter extends the work of the previous chapter to present a deep dive into SoC-accelerator interfaces. Today’s complex SoCs contain many specialized IP blocks to accelerate specific workloads. However, real applications, such as real-time image classification or continuous speech recognition, increasingly span multiple algorithms and IP blocks. Thus, the interfacing of accelerators within the SoC has become an pivotal consideration for their overall performance and energy efficiency. In this chapter, we study an accelerator interfacing scheme we call delegated coherency, in which the accelerator is directly connected to the shared last-level cache (LLC) of the CPU cluster. This allows the accelerator to access coherent memory without the hardware overhead of implementing its own local cache. Using vision pipelines and deep neural networks (DNNs) as the driving workloads,
we demonstrate how delegated coherency interfaces can improve performance by 20-40% and energy by up to 40% with no changes to accelerator datapaths. Furthermore, we show that these benefits are resilient to shared resource contention in the LLC from concurrently running applications, due to a software prefetching mechanism made possible by this interface. In order to study these end-to-end workloads, we also developed SMAUG, the first simulation-compatible DNN framework supporting custom accelerator models.

5.1 Introduction

All modern computing platforms from datacenters down to mobile devices are power limited. To meet computational demands within power constraints, the industry has increasingly turned to heterogeneous systems-on-chip (SoCs) with many specialized accelerators. For highly demanding workloads, dedicated hardware accelerators are the most efficient approach and have been heavily promoted in commercial products and research.

In recent years, we have observed a trend we expect to continue: the emergence of new workloads from the composition of existing ones, therefore spanning multiple IP blocks. Notable examples include real time image classification, speech recognition and translation, facial recognition as a biometric security mechanism, and more. Such workloads will by necessity need to span multiple independent hardware accelerated pipelines; for example, real time image classification requires composing the camera pipeline with a deep neural network (DNN) accelerator. As a result, the interfacing of accelerators and SoCs will become an increasingly important architectural component to these workloads, in addition to the accelerators themselves.

Fixed-function accelerator designs typically use private scratchpads for local storage and communicate with the outside world through a software-managed DMA interface. But the overhead and awkward programming model of this interface has driven researchers to investigate alternative interfacing options, such as hardware-managed caches [3, 97, 128, 129]. While cache coherency for pro-
grammable accelerators like GPUs have been extensively studied [5, 69, 135–137], only in recent years have academia and industry started investigating the potential use of caches for fixed-function accelerators and FPGA platforms. Existing work on accelerator caches have largely assumed that the accelerator has its own private cache. However, not all accelerators that want access to coherent memory also need a local cache, which costs additional area, energy, and complexity.

In this work, we explore an interface called delegated coherency that occupies a middle ground between SW-managed and fully hardware-managed coherency. Here, the accelerator is connected directly to the last level cache (LLC) of the CPU cluster. It issues coherent memory requests to the LLC, which handles all the coherency traffic on the accelerator’s behalf. This enables the accelerator to reap the benefits of coherency without adding more area and complexity requirements to the IP block and integration.

With the recent intense interest in deep learning, in this chapter we explore the optimization of delegated coherency interfaces, with vision pipelines and DNNs as the driving workload. First, we show that if a DNN accelerator is the primary process in a system, these interfaces can improve performance by 20% and energy by up to 40%. Then, we demonstrate the impact of shared resource contention when image processing workloads are simultaneously running and sharing the LLC. Finally, we show how delegated coherency interfaces enable software assist mechanisms which provide highly effective mitigation for interference from shared resource contention, enabling all workloads to achieve within 10% or better of their standalone performance. Critically, all of the results in this work were accomplished without changing the accelerator’s core datapath.

In order to rapidly explore and evaluate different accelerators and SoC integration scenarios, we use simulation. However, a comprehensive system-level study of DNN accelerators requires end-to-end workload analysis, not just per-layer analysis, and existing DNN frameworks like TensorFlow/PyTorch do not support integrating custom hardware accelerator models and running a full network in simulation. To address this gap in research infrastructure, we develop SMAUG: Simulating Machine Learning Accelerators Using gem5-Aladdin. gem5-Aladdin is
an SoC simulator that supports modeling of complex heterogeneous SoCs [158], and SMAUG is the first simulation-compatible DNN framework, with a modular architecture that allows easy plug-and-play of new hardware blocks for DNNs and analysis of accelerator interfacing implications. To spur research in this area, SMAUG will be open sourced.

In summary, the contributions of this work are:

- An analysis of the performance and energy efficiency gains that delegated coherency interfaces provides for hardware-accelerated DNN and image processing workloads.
- SMAUG, the first end-to-end DNN framework compatible with existing SoC simulators enabling rapid DNN accelerator prototyping and SoC integration.
- An evaluation of software-based mechanisms to mitigate the impact of shared resource contention with delegated coherency interfaces.

5.2 Motivation and Background

As the number of specialized IP blocks on an SoC grows with time, understanding trade-offs of interfacing becomes more and more important. Oftentimes, these IP blocks are not running completely independently, but rather as part of a larger pipeline dedicated to an important workload. Here, we motivate the problem of accelerator interfacing in the context of one of the most important and compute intensive pipelines in current mobile systems: the camera subsystem, upon which many vision-oriented applications are built. Then, we provide an overview of the various types of interfaces used in modern SoC design.

5.2.1 Example: Camera Visual Pipeline

The camera pipeline is a long series of spatial linear and non-linear filters and transforms to transform the raw output of the image sensor into a final picture. The sensor itself sits behind a Bayer color filter, so each individual photodiode only
captures light from one primary color (RGB). As a result, the output of the sensor is an array of pixel values, each representing the intensity of a single color. The process that estimates the original color of each pixel from this raw image is called demosaicing. The subsequent image processing then proceeds through many more processing steps. White balance correction adjusts the relative intensity of each color to preserve the correct balance of colors in the image. Gamma correction adjusts the intensity of each pixel based on a power law to store more information in color/intensity regions that human eyes are more sensitive to. Other steps include color space conversion, denoising/sharpening, chromatic aberration correction, chroma subsampling, and more. Finally the image is compressed using the JPEG format, preserving low frequency detail that human eyes are sensitive to while removing imperceptible high frequency information \([54]\). Because this pipeline is highly compute intensive, the image signal processor (ISPs) of modern camera systems are typically implemented as fixed function ASICs \([2]\), either as part of the camera module or in an IP block on the SoC.

It may seem that this pipeline is self-contained within the ISP itself, but this is hardly the full story. End-to-end camera applications involve not just image capture, but also streaming a live view to a frame buffer to view on an LCD display. When recording video, the output of the image processor must also be encoded and written to persistent storage, while an audio stream is separately recorded, processed, synced with the video, and stored \([145]\). Contemporary computational photography methods often rely on capturing a rapid burst of full resolution images and sending them to an application processor for processing \([70]\). And recently, it is becoming increasingly common to attach deep learning models at the end of these pipeline to perform tasks like real-time image classification \([133]\), text recognition and translation (e.g. Google Translate), and more.

All these heavy image processing workloads create significant load on the memory system and system interconnects. Today, these separate IP blocks communicate predominantly through DRAM, orchestrated by the CPU. As time passes, computation and communication demands will only increase (e.g. higher image/video resolution). To enable these future workloads, in addition to build-
<table>
<thead>
<tr>
<th>Type</th>
<th>Benefits</th>
<th>Drawbacks</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No hardware overhead</td>
<td>Large software management overhead</td>
<td>DMA</td>
</tr>
<tr>
<td>IO</td>
<td>Low hardware cost in IP</td>
<td>No data sharing, high access latency</td>
<td>ACE-Lite [7, 167]</td>
</tr>
<tr>
<td>Full</td>
<td>Little software management</td>
<td>Higher hardware overhead</td>
<td>ACE [7, 167], TileLink [40]</td>
</tr>
<tr>
<td>Delegated</td>
<td>IO coherency without additional bus master</td>
<td>No data sharing, additional HW in LLC</td>
<td>ACP [111], RoCC [51]</td>
</tr>
</tbody>
</table>

Table 5.2.1: Examples of interface options. ACE = ARM Coherency Extensions, ACP = Accelerator Coherency Port.

ing faster individual accelerators, we must think of new ways to connect them, such as through direct links to reduce interconnect bandwidth use or via shared on-chip storage to reduce DRAM traffic. Understanding these problems in SoC-accelerator interfacing and identifying opportunities to increase overall system performance at the level of interfacing is the focus on this chapter. Our experiments include a mixture of DNN inference on a variety of networks, blurs, stencil pipelines, and FFTs, running on both CPUs and hardware accelerators, as well as a complete camera pipeline implemented in Halide.

5.2.2 Interfacing Overview

There are several approaches to interfacing an accelerator with an SoC, based on various degrees of cache coherency support in hardware. Table 5.2.1 summarizes these interfacing schemes in terms of their benefits and trade-offs and practical examples of protocols implementing each type. As an overview, we discuss all the major types briefly.

Software coherency is the simplest approach to sharing data from a hardware point-of-view. Software alone is responsible for explicitly invalidating and flushing cache lines that the accelerator is going to read and/or write, resulting in a challenging programming model where the programmer must correctly manage complex coherency operations in a heterogeneous, multicore system. On the other end of
the spectrum is full coherency, where a coherent interconnect between the CPU cluster and accelerator ensures coherent shared memory operations. While this is the ideal programming model, the hardware is much more expensive, and this generally requires the accelerator to maintain a cache, which may not actually suit the kernel being accelerated. Much prior work has focused on this type of interface [1, 3, 38, 97, 136]. IO coherency is a middle-ground solution between these two ends of the spectrum. Here, the accelerator can snoop caches of other coherent masters (e.g., a CPU cluster), but it does not become a sharer of that data because it lacks a cache. This maintains the illusion of coherent shared memory from the accelerator’s perspective with less complexity. Finally, delegated coherency takes the concept of reducing overhead in the accelerator even further. Here, the accelerator attaches directly to the last-level cache (LLC) of the CPU and issues memory requests to the LLC, which handles all the coherency tasks on behalf of the accelerator. More about the benefits of delegated coherency are discussed in Section 5.3.2.

The most common type of interface varies by the type of accelerator. Contemporary integrated GPUs are typically fully coherent masters [89, 113], but discrete GPUs typically offer IO coherency due to the high latency of the PCIe interconnect (however, recent NVIDIA GPUs on IBM POWER systems can become fully coherent through CAPI [43, 168]). On the other hand, fixed function accelerators, the class of accelerators our work focuses on, typically only offer software coherency and private scratchpads for storage (see Section 5.6 for related work on accelerator caches).

In this work, we compare delegated coherency, which provides low latency access to a large pool of shared memory with no hardware cost, with software coherency, the standard baseline. An SoC integration engineer can influence the interface to an accelerator, but may have limited influence over the accelerator’s datapath itself. Therefore, the possibility of increasing accelerator performance and energy efficiency, via the interface alone and without requiring any additional hardware, is extremely attractive and deserving of close investigation.
5.3 System Design

To evaluate the implications of various accelerator integration scenarios, we developed a hypothetical mobile SoC containing a set of accelerators for DNNs and image processing tasks. The workloads we run include both hardware-accelerated and optimized software implementations. First, since DNNs are the most compute and memory intensive workload studied in this work, we evaluate its performance and energy when it runs as the only program in the system. Once we understand the potential gains from delegated coherency on DNNs, we run a mixture of DNN and ISP workloads concurrently across multiple CPUs and accelerators. This is a multiprogrammed workload that models a camera-based application feeding images to a DNN. In this section, we describe our experimental design and workloads. In the next section, we’ll describe the SMAUG framework which runs the DNNs.

5.3.1 SoC Architecture

Figure 5.3.1 shows the SoC used in this work. The parameters of the CPU cluster and memory hierarchy are listed in Table 5.3.1. In addition, there are two accelerator complexes in the SoC. The first is a DNN accelerator, based upon the Nvidia Deep Learning Accelerator (NVDLA) design [42]. We choose this design because it is an open-source project backed by a major hardware vendor, and this work is about accelerator SoC integration, rather than new accelerator design. The second is an ISP accelerator complex, composed of a set of fixed function units for computing common ISP functions such as spatial filters (stencil) and transforms (FFT). Both accelerator complexes run at 1GHz.

DNN Accelerator Microarchitecture

As shown on the left side of Figure 5.3.1, the DNN accelerator design is based around a convolution accelerator consisting of eight PEs. Each PE consists of a 32-way multiply-accumulate (MACC) array and operates on a different output feature
map. The dataflow is described in Figure 5.3.2. Inputs and weights are 16-bit fixed point, while outputs are accumulated in 32-bit fixed point and reduced to 16-bit before being written to the scratchpad. In the emerging vernacular used to describe DNN dataflows, this dataflow is L0 weight-stationary (weights are reused at the register level within a MACC array), and L1 input/output stationary (on every cycle, inputs are reread from the SRAM and outputs are accumulated in-place in the SRAM). It is backed by three distinct SRAMs, one each for inputs, weights, and outputs. We only model the core datapath and dataflow of NVDLA, not other specific features like its convolution buffer.

In addition to convolution, the accelerator also supports inner products by mapping them onto the convolution hardware. To reduce memory bandwidth re-
BUFFER IN[IN_R][IN_C][IN_H];
BUFFER WGT[NUM_PES][WGT_R][WGT_C][IN_H];
BUFFER OUT[NUM_PES][OUT_R][OUT_C];
parallel for (pe = 0 to NUM_PES)
  for (kr = 0 to WGT_R - 1)
    for (kc = 0 to WGT_C - 1)
      for (cb = 0 to IN_H/32 - 1) {
        // cb = channel block
        // Each PE has its own weight reg.
        BUFFER wgt_reg[0:31] = WGT[pe][kr][kc][cb:cb+31];
        // Now iterate over the input rows and cols.
        for (r = 0 to OUT_R - 1)
          for (c = 0 to OUT_C - 1)
            parallel for (h = 0 to 31) {
              // 32-way spatial reduction in channel dimension.
              OUT[r][c][pe] += IN[r+kr][c+kc][cb*32+h] *
                wgt_reg[h];
            }
      }
  }

Fig. 5.3.2: Dataflow implemented by the accelerator. Apart from syntax, this is nearly the actual C code in SMAUG.

requirements, the accelerator supports reading sparse compressed fully-connected weights, but they are first decompressed into the internal scratchpads before running the inner product. The compression format is based on the CSC format described by Han et al. [68]. Weights compression for convolutional layers is not supported, since much research in the machine learning and architecture community has observed that they are harder to prune than fully-connected layers [68, 131]. Max and average pooling and batch normalization are natively supported, as are a range of activation functions.

ISP Accelerators

The ISP accelerator complex consists of accelerators for 3D stencils, FFTs, and sparse matrix-vector multiply. They were selected to cover a breadth of target applications: stencils are at the core of many ISP algorithms, FFTs are widely used for spectral analysis and filtering, and sparse matrices are found in image compres-
sion and sparse representations [64]. We use implementations from the Mach-Suite benchmark suite [142]. Microarchitectural parameters like the datapath parallelism and SRAM banking were obtained from a design sweep identifying the EDP optimal design point.

5.3.2 SoC Interfacing

Both accelerator complexes are connected to the SoC via two interfaces. The first is a standard DMA interface, which provides a 256-bit non-coherent connection to the system bus. The second is called the accelerator coherency port (ACP)\(^1\), which implements the delegated coherency interface. The LLC accepts accelerator read/write requests, invalidating other copies of the cacheline in other private caches and merging partial line writes as necessary. This is one-way coherency, so the CPUs will not see any writes that the accelerator makes to its internal SRAMs except when the accelerator issues an ACP write. Note that these two ports are merely gateways to external memory; the accelerator can access its internal scratchpads at will without generating DMA or ACP traffic. For both interfaces, physical addresses are used; this means that the ACP does not require an internal TLB or IOMMU for address translation, but pages must be pinned in memory.

In our experiments, we select one of these two interfaces to load all required input data from host memory into the accelerator SRAMs. The two interfaces need not be mutually exclusive; it is possible for the accelerator to use both simultaneously. As the data is arriving, the accelerator will begin computing if possible. During compute, the accelerator will write and update output values into its output scratchpad, and when finished, the accelerator copies the finished output values back to host memory, either via ACP (in which case it will be stored in the LLC initially) or via DMA (in which case it will go straight to DRAM).

\(^1\)Although we refer to this port as “ACP”, none of the experiments should be taken as ARM-specific.
Benefits and Drawbacks of Delegated Coherency

There are two major benefits of delegated coherency that other interfaces do not provide. First, it opens the entire capacity of the shared LLC for accelerators to access with relatively low latency. For workloads that are highly sensitive to on-chip storage, this can make a big difference, as we will show in Section 5.5. Second, through the LLC, the CPU and accelerator are more tightly coupled than just being connected to the system bus. In this work, we exploit this coupling by using the CPU to prefetch data the accelerator will need next into the LLC. In other words, a CPU is able to actively assist the accelerator. Without delegated coherency, accelerators would need to give up some scratchpad space to store the prefetched data, which is the tradeoff that double buffering requires. The primary drawback is that since LLC is a shared resource, multiple memory-intensive workloads running simultaneously will compete with each other, potentially causing performance degradation all on sides. As we will show in Section 5.5.2, shared resource contention can be effectively mitigated with software assistance.

5.3.3 Workloads

We evaluate variety of DNN and image processing workloads, across both hardware accelerators and CPUs. On the DNN front, we investigate three image classification tasks: MNIST, CIFAR10, and CIFAR100. For each dataset, we use two different networks, taken either from published literature or trained by ourselves. Whenever possible, we use real weights and inputs. Table 5.3.3 summarizes the networks used. The goal was to cover a diverse set of network topologies that still map well to the accelerator’s dataflow, which is optimized for convolution shapes deep in input/output feature maps. All DNN workloads are hardware accelerated. Due to simulation constraints, we cannot currently run larger inputs (e.g. ImageNet), but we anticipate that ongoing improvements to simulation speed and sampling will eliminate this constraint.

For ISP workloads, we use three MachSuite benchmarks - stencil-stencil3d, fft-transpose, and spmv-crs - on the ISP ac-
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>MachSuite</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fft-transpose</td>
<td>512-point 1D complex FFT</td>
<td>512 windows of complex signal, each 512-pt</td>
</tr>
<tr>
<td>stencil-stencil3d</td>
<td>48 chained 3D second-order stencil</td>
<td>32x32x32 input, 32-bit float</td>
</tr>
<tr>
<td>spmv-crs</td>
<td>Sparse matrix-vector multiply</td>
<td>5 copies of 2048x512 matrix, 5% density</td>
</tr>
<tr>
<td>Halide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fft</td>
<td>16x16 2D forward and inverse FFT</td>
<td>160 distinct 16x16 complex signals</td>
</tr>
<tr>
<td>stencil-chain</td>
<td>Eight chained 2D stencils, 5x5 each</td>
<td>256x256 color image</td>
</tr>
<tr>
<td>blur</td>
<td>3x3 blur kernel</td>
<td>640x480 color image</td>
</tr>
<tr>
<td>camera-pipeline</td>
<td>Hot pixel suppression, deinterleave, demosaic, white balance, sharpen</td>
<td>512x375 color image</td>
</tr>
</tbody>
</table>

Table 5.3.2: Image processing workloads. Halide workloads run on the CPU-only. MachSuite workloads are all hardware accelerated, with some workload input sizes increased from their original sizes.

accelerator complex. Since ISP programs must run on CPUs too, we also use optimized Halide implementations of three core ISP kernels [141]. For a more complex workload that combines many kernels and processing stages, we use a Halide implementation of a camera pipeline, which transforms raw data from a camera sensor into a usable image. These workloads are described in Table 5.3.2. All Halide workloads were built with gcc-5.4.0 using ahead-of-time compilation without auto-scheduling, so we only measure the performance of the kernels and not the JIT.

With this set of programs, we measure per-program performance in several multiprogrammed settings. Although we don’t integrate Halide or MachSuite with the DNN as a unified application, our multiprogrammed experiments are still able to capture the behavior of such an application in steady state.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Name</th>
<th>Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>Minerva</td>
<td>4 FC layers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>784 ( \times ) 256 ( \times ) 256 ( \times ) 10</td>
</tr>
<tr>
<td>MNIST</td>
<td>LeNet5</td>
<td><strong>Conv</strong>: 3x3x32, <strong>Conv1</strong>: 3x3x32, <strong>Pool</strong>: 2x2, <strong>FCo</strong>: 128, <strong>FC1</strong>: 10</td>
</tr>
<tr>
<td>CIFAR10</td>
<td>CF-Simple</td>
<td><strong>Conv0</strong>: 3x3x32, <strong>BN</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv1</strong>: 3x3x32, <strong>Pool0</strong>: 2x2, <strong>BN Conv2</strong>: 3x3x64, <strong>BN</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv3</strong>: 3x3x64, <strong>Pool1</strong>: 2x2, <strong>BN FC0</strong>: 512, <strong>FC1</strong>: 10</td>
</tr>
<tr>
<td>CIFAR10</td>
<td>CF-VGG</td>
<td><strong>Conv00</strong>: 3x3x64, <strong>Conv01</strong>: 3x3x64, <strong>Pool0</strong>: 2x2</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv10</strong>: 3x3x128, <strong>Conv11</strong>: 3x3x128, <strong>Pool1</strong>: 2x2</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv20</strong>: 3x3x256, <strong>Conv21</strong>: 3x3x256, <strong>Pool2</strong>: 2x2</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv30</strong>: 3x3x512, <strong>Conv31</strong>: 3x3x512, <strong>Pool3</strong>: 2x2</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>FCo</strong>: 512, <strong>FC1</strong>: 10</td>
</tr>
<tr>
<td>CIFAR100</td>
<td>ELU-o</td>
<td><strong>Conv00</strong>: 5x5x192, <strong>Pool0</strong>: 2x2</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv10</strong>: 1x1x192, <strong>Conv11</strong>: 3x3x240, <strong>Pool1</strong>: 2x2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv20</strong>: 1x1x260, <strong>Conv21</strong>: 2x2x260, <strong>Pool2</strong>: 2x2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv30</strong>: 1x1x260, <strong>Conv31</strong>: 2x2x280, <strong>Pool3</strong>: 2x2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv40</strong>: 1x1x280, <strong>Conv41</strong>: 2x2x300, <strong>Pool4</strong>: 2x2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv50</strong>: 1x1x300, <strong>Conv51</strong>: 1x1x100</td>
</tr>
<tr>
<td>CIFAR100</td>
<td>ELU-1</td>
<td><strong>Conv00</strong>: 3x3x384, <strong>Pool0</strong>: 2x2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv10</strong>: 1x1x384, <strong>Conv11</strong>: 2x2x384, <strong>Conv12</strong>: 2x2x640,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv13</strong>: 2x2x640, <strong>Pool1</strong>: 2x2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv20</strong>: 1x1x640, <strong>Conv21</strong>: 2x2x768, <strong>Conv22</strong>: 2x2x768,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv23</strong>: 2x2x768, <strong>Pool2</strong>: 2x2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv30</strong>: 1x1x768, <strong>Conv31</strong>: 2x2x896,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv32</strong>: 2x2x896, <strong>Pool3</strong>: 2x2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv40</strong>: 3x3x896, <strong>Conv41</strong>: 2x2x1024,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv41</strong>: 2x2x1024, <strong>Pool4</strong>: 2x2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv50</strong>: 1x1x1024, <strong>Conv51</strong>: 2x2x152, <strong>Pool5</strong>: 2x2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv60</strong>: 1x1x152, <strong>Pool6</strong>: 2x2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conv70</strong>: 1x1x100</td>
</tr>
</tbody>
</table>

Table 5.3.3: Datasets and networks used in this chapter. All convolutional/pooling layers use stride 2, and all layers use RELU except for the last two, which use ELU. As several of these networks organize layers into logical stacks, layers labeled “ConvXY” indicate stack X, sub-layer Y. See 5.3.4 for more details on network properties.
Table 5.3.4: Properties of networks used in this chapter. Since our accelerator only supports decomposition of sparse weights for FC layers, we only prune FC weights.

<table>
<thead>
<tr>
<th>Name</th>
<th>Density (% nonzeros)</th>
<th>Total weights (total nonzero)</th>
<th>Accuracy</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minerva</td>
<td>2%, 2%, 2%, 0%</td>
<td>335,114 (9,197)</td>
<td>97%</td>
<td>[143]</td>
</tr>
<tr>
<td>LeNet5</td>
<td>FCo: 5%</td>
<td>600,810 (34,655)</td>
<td>98%</td>
<td>[98]</td>
</tr>
<tr>
<td>CF-Simple</td>
<td>2% on FCo</td>
<td>2,168,618 (112,568)</td>
<td>85%</td>
<td>N/A</td>
</tr>
<tr>
<td>CF-VGG</td>
<td>5% on FCo</td>
<td>8,689,482 (7,690,125)</td>
<td>90%</td>
<td>[184]</td>
</tr>
<tr>
<td>ELU-0</td>
<td>Not pruned</td>
<td>1,666,384</td>
<td>71.25%</td>
<td>[47]</td>
</tr>
<tr>
<td>ELU-1</td>
<td>Not pruned</td>
<td>39,316,096</td>
<td>77.72%</td>
<td>[47]</td>
</tr>
</tbody>
</table>

5.3.4 SIMULATION ENVIRONMENT

The SoC and accelerators are modeled using gem5-Aladdin [158]. The simulator already supports most of our required functionality, but it does not have support for delegated coherency. To model the accelerator coherency port (ACP), we take a standard MESI cache coherence protocol implemented in gem5’s Ruby modeling framework and augment it with an ACP controller. This controller is connected to an accelerator’s memory interface and generates requests to the L2 cache on behalf of the accelerator. Unlike a cache controller, it does not implement a cache and leaves ownership of the relevant cache lines with the L2 cache rather than the accelerator itself. Using Verilog simulation of an ARM Cortex A53 CPU, we measure ACP hit latency to be 20 cycles, which we set as the LLC latency.

To obtain area and power estimates, we take a multi-pronged approach. We characterize various 16-bit functional units for power and area in a commercial 16nm FinFET process and plug them directly into Aladdin. To model accelerator local scratchpads, we build and characterize a variety of SRAM blocks through a commercial memory compiler in the same technology node. LLC power estimates are obtained from CACTI 7 [87], and DRAM power is modeled by DRAM-
Power [93], with timing and power parameters taken from a commercial LP-DDR4 product datasheet [121].

While simulation of MachSuite and Halide workloads is straightforward, simulating end-to-end deep neural networks is highly nontrivial, due to the lack of simulation-compatible DNN frameworks. We address this issue in the next section.

5.4 SMAUG: Enabling Research of DNN Accelerator Integration

Although there are many popular deep learning frameworks in use today, none of them present an easy way to evaluate SoC-accelerator integration because they are not amenable to simulation, an experimental methodology widely used in computer architecture. As result, evaluation of end-to-end DNN workloads generally requires FPGA or silicon implementation, making early-stage design space exploration infeasible or slow. As a result, most prior DNN accelerator work evaluates performance on a layer-by-layer basis, which ignores overhead from data layout transformation, memory management, kernel launch, and more.

To address the lack of simulation-compatible DNN frameworks, we built SMAUG: Simulating Machine Learning Accelerators Using gem5-Aladdin. The goal of SMAUG was not to replace existing frameworks like TensorFlow or PyTorch, but rather to enable DNN accelerator researchers to quickly evaluate a DNN accelerator design in the context of a complete SoC running end-to-end workloads. As the name suggests, SMAUG is compatible with the gem5-Aladdin SoC simulator [158] and its associated LLVM-based toolchains.

SMAUG supports a wide range of commonly used operators in a linearly stacked network architecture, with support for more complex network organizations coming. It supports inference only. Networks are specified through a simple configuration file format. In addition to the core kernels (matrix-vector multiply, convolution, etc.), SMAUG also supplies a library of supporting functions, like data layout transformations, sparse data handling, and tensor blocking, which are
critical for running end-to-end DNN models. Finally, SMAUG is multithreaded, enabling users to exploit task-level parallelism across multiple compute elements.

Figure 5.4.1 shows the execution flow of SMAUG. The user provides SMAUG with the model configuration, a model parameter file, a target backend, and a set of parameters describing specific features of the backend that he/she wants to use (e.g. selectively enable or disable various hardware blocks). Based on this information, SMAUG will first perform a series of offline preprocessing steps. For example, if the user specified to use reduced precision arithmetic, SMAUG will quantize the appropriate input/weights data. Other preprocessing steps include sparse data compression, data layout transformations, and more, depending on the target backend. Then, SMAUG invokes a tiling optimizer to compute an efficient way of breaking up a large operator into smaller pieces that the backend can run. Finally, SMAUG dispatches each tile of work to the appropriate compute element, potentially multiple at a time if multithreading is enabled.
5.4.1 SMAUG Tiling Optimizer

Due to the limited amount of local storage easily accessible by an accelerator, individual layers of DNNs often have too many weights and/or inputs to run at once, therefore requiring work tiling. Whenever tiling is required, redundant data movement is likely necessary, so identifying efficient tiling schedules (also called “loop nests”) that maximizes data reuse and minimizes data movement between different levels of the memory hierarchy is critical to achieving high performance. However, the general problem of finding the optimal solution is combinatorial in dimensionality and tiling factors, and a solution is beyond the scope of this work.

In SMAUG, we circumvent a general solution by implementing a specialized tiling optimizer for each backend. The reason is that any particular instantiation of an accelerator implements at most a few dataflows to exploit particular parallelism patterns. For example, the NVDLA convolution engine reduces partial products in the channel dimension, so it benefits from a tiling schedule that maximizes the channel dimension of the input and weight tiles. But such a schedule does not make sense for an accelerator that computes 1D or 2D convolutions, like the row-stationary dataflow [29]. By specializing the optimizer for a specific backend, we restrict the search space to a narrower set of possibilities that can be exhaustively explored. The final schedule is one that maximizes both the utilization of the local scratchpads and the functional units.

There are two major steps to the tiling optimizer: 1) Determine the sizes of the tiled input and output data, and 2) Determine the order/nesting in which they are processed. As an example, consider the dataflow implemented by the NVDLA convolution engine (see Figure 5.3.2), which we use in the experiments in this chapter. Without loss of generality, assume in this example that we use same padding and stride 1 (so that the spatial dimensions of the input and output are equal). Because the innermost loop of the dataflow reduces partial products along the channel dimension, the tiling optimizer maximizes the number of channels in each data tile. Let the inputs have dimensions $H \times W \times C$, the weights have dimensions $M \times K_h \times K_w \times C$, and the outputs have dimensions $H \times W \times M$, and
the accelerator has three SRAMs for inputs, weights, and outputs. Then the tiling optimizer will block the data like so:

- **Input tiles** are row-wise slices of the input volume. They have dimensions \( B_i \times W \times C \), where \( B_i = \lfloor \text{SRAM}_{\text{input}}/(W \times C) \rfloor \)

- **Output tiles** are row-wise slices of the output volume. They have dimensions \( H \times W \times B_o \), where:

\[
B_o = \min(K_{\text{max \_filters}}, K_{\text{max \_ofmaps}}) \\
K_{\text{max \_filters}} = \lfloor \text{SRAM}_{\text{weights}}/(Kf \times Kc \times C) \rfloor \\
K_{\text{max \_ofmaps}} = \lfloor \text{SRAM}_{\text{outputs}}/(Bi \times W) \rfloor
\]

- **Weight tiles** are \( B_o \) complete convolutional filters, with dimensions \( B_o \times Kf \times Kc \times C \).

After determining the tile sizes, the tiling optimizer considers all the implemented loop nest orderings and selects one to use for the current layer with an analytical model to estimate the total cost of data movement. Accelerators often have a hierarchy of local memory, with small buffers close to the PEs and globally shared buffers farther away. The analytical model considers this memory hierarchy and accounts for different access latencies based on where the data may come from. For example, one loop nest might look like this:

```c
// Input stationary.
for (it = 0 to numInputTiles)
  // Copy data over ACP or DMA.
  memcpy(inputSRAM, inputTiles[it]);
  for (wt = 0 to numWeightTiles)
    ot = calcOutputTileIdx(it, wt);
    memcpy(weightSRAM, weightTiles[it]);
    CONV_KERNEL(inputSRAM, weightSRAM, outputSRAM);
    memcpy(outputTiles[ot], outputSRAM);
```

94
If the accelerator has an L2 scratchpad or cache, which can fit all the weights, this loop nest may work well, but if not, then cache thrashing can occur. Instead, if the inputs are smaller than weights, the tiling optimizer may prefer a loop nest that reverses the order of the inputs and weights loops, or one that further blocks the problem into groups of tiles. All of these loop nests are implemented separately for each backend, as is typical of high performance DNN backends [30, 41].

5.4.2 Runtime Scheduler

Once the tiling schedules have been generated for each layer, the runtime scheduler dispatches each tile to the appropriate compute element. If there are multiple compute elements and task-level parallelism exists, SMAUG can dispatch independent work to separate compute elements. Any operators that are not supported in the backend hardware accelerators are executed on the CPU instead. The scheduler allocates host memory to store results from the accelerator for post-processing (e.g., zero-padding for data alignment, re-shuffling of data into the target layout, etc). Because the syscall-emulation mode of gem5 does not have a thread scheduler, SMAUG implements a custom thread pool with a work queue, enabling other CPUs in the system to be assigned independent work.

The runtime scheduler is also responsible for sampling during simulation. Modern DNNs are very deep and compute intensive, often taking billions of operations,
and because Aladdin is a trace based simulator, it may be infeasible to store and simulate a complete forward pass. However, since DNN computation is so regular, a sampling approach works well. SMAUG implements sampling at the tiling level. Based a user’s specified sampling factor, the runtime scheduler identifies a representative subset of input/output tiles and only executes those during simulation. Afterwards, we upsample all of the relevant performance statistics. To ensure that we capture the memory behavior of the original workload, we implement a traffic generator module in gem5 that imitates the memory access patterns of the accelerator and is invoked during sampling. Sampling error is measured to be less than 10% due to the regularity of DNNs.

5.4.3 SMAUG BACKENDS

The backend is responsible for computing the convolutions/inner products/etc required by the model. SMAUG supports optimized CPU backends, such as Intel MKL-DNN [41], as well as accelerated backends, like NVDLA, which are modeled with Aladdin. One of the key aspects of SMAUG is the ease with which a new HW accelerator model can be implemented and integrated into the framework. For example, apart from syntax, Figure 5.3.2 is almost identical to the code that models the convolution engine. In fact, merely 5% of the code is (currently) used to model the various hardware blocks, with the rest devoted to computing tiling schedules, memory management, data movement, cache coherency, and work coordination. This design is meant to let researchers spend more time on studying hardware-software interactions and less time on low-level hardware modeling.

5.5 EXPERIMENTS AND RESULTS

In this section, we present experiments that explore the impact of SoC interfacing on performance and energy efficiency of accelerated workloads. First, since the DNNs are the most computationally and memory intensive workloads in this paper, we start by evaluating benefits of delegated coherency when each DNN (Table 5.3.3) is the only workload in the system, so we can measure the maximum impact
of the interface. Then, we combine it with compute-heavy ISP workloads, running on both accelerators and CPUs, to model realistic multiprogrammed workloads on today’s mobile SoCs. We measure the impact of shared resource contention and demonstrate the effectiveness of a SW prefetch mechanism enabled by delegated coherency.
5.5.1 DNNs and Delegated Coherency

In this section, we explore the impact of delegated coherency interfaces on DNN workloads as the only process in the system. We evaluate three different integration scenarios. The baseline scenario uses DMA (software coherency) to connect the accelerator and SoC. The second scenario adds the accelerator coherency port (ACP), but the DMA interface still remains as an option. The third scenario augments the second by using a helper thread to prefetch accelerator data into the LLC. In addition, we also sweep the allocation of local accelerator memory. DNN accelerators are highly sensitive to local memory size, and since delegated coherency exposes another pool of storage otherwise not available to the accelerator, it potentially enables trading private memory for shared resources.

Figure 5.5.1 shows performance and energy for each scenario. We break down execution time into four categories: Accel: time spent in the hardware accelerator; ACP/DMA transfer: time required to transfer inputs/weights data prior to starting computation; DMA overhead: software coherency management time, if any; Data layout: time spent on any data format manipulations the CPU must run between tiles or layers.

As expected, performance and energy improve with larger local scratchpads, since there is less redundant data movement caused by tiling. More importantly, we find that by attaching the DNN accelerator over ACP, DNN performance improves by between 10%-20% and energy consumption drops by between 25 to 40%, depending on the scratchpad size. All these gains were achieved without any changes to the core datapaths. They come from three effects: turning expensive DRAM accesses into LLC hits, a software-assisted prefetching scheme possible only through delegated coherency, and elimination of software coherency management overhead.

Turning DRAM Accesses into LLC Hits

Using ACP, many expensive DRAM accesses are converted into cheaper LLC hits. For the three smaller networks (Minerva, LeNet5, CF-Simple), the performance
Figure 5.5.2: ACP decreases total DRAM traffic due to low overall miss rates in the LLC.

cost of data movement is low to begin with, so ACP does not substantially improve data transfer time, but by not needing to flush data to DRAM only to read it back again, all three networks see a significant reduction in DRAM energy. On two of the three larger networks (ELU-0 and ELU-1), having low latency access to the LLC lets the cache-aware tiling optimizer select an alternative loop nest which iterates over the smaller input tiles on the innermost level rather than the larger weights. This loop nest avoids cache thrashing because the inputs activations are much smaller than the weights and can often fit into cache. This is primarily responsible for the dramatic performance improvement and energy reduction on our largest network, ELU-1, with 48KB of per-buffer SRAM. On the whole, Figure 5.5.2 shows end-to-end DRAM traffic is reduced by as much as 60-80% on the small networks and up to 50% on the biggest networks. The only outlier is CF-VGG, which sees less than 5% and 8% in performance and energy improvements, respectively, because both tiling schedules that SMAUG generates for this network cause cache thrashing. Improving tiling schedules is an area of constant improvement.

Software Prefetch Assist

Because delegated coherency creates a tighter coupling between a CPU and an accelerator, the CPU is able to assist the accelerator through their shared cache.
We observed that on the three larger CNNs, simply using the ACP only provided performance gains of between 5-15%, because the tiling schedules can sometimes cause cache thrashing when the inputs or weights of a layer don’t fit in L2 cache. To mitigate this problem, we use helper threads to prefetch data into the L2 cache for the next tiles. Prefetching does not generate significant LLC capacity pressure because each data tile is at most as large as the accelerator’s SRAM buffers, which are small relative to the LLC. However, if not timed correctly, it could compete for cache or DRAM bandwidth while accelerator is loading data, which ends up hurting rather than helping performance. To avoid this scenario, we synchronize the helper thread with the accelerator using spinwaits and delay the prefetch until the accelerator reports that its data loading phase has completed. This is similar in spirit to double buffering, but it requires no hardware changes and lets the accelerator use all of its local memory, whereas double buffering usually reserves half of the memory for itself.

The results of the software prefetch assistance are shown as the rightmost group of bars in Figure 5.5.1. On larger networks, it is remarkably effective, improving performance by another 10-20%. In fact, with this optimization, an accelerator using ACP with the least local memory (64KB) can outperform the same accelerator using DMA with the most local memory (128KB). While the tradeoff between limited private resources and larger shared resources is widely applied, we are not aware of any prior work that exploits this tradeoff in the context of SoC integration of accelerators. Because a prefetched cacheline is not guaranteed to be referenced before the line gets evicted, this results in 2-3% additional energy (Fig 5.5.1b) and a slight increase in DRAM traffic (Fig. 5.5.2).

Software Coherency Overhead

Software managed coherency (DMA) is expensive in terms of performance. It accounts for 20% of total execution time on LeNet5 and CF-Simple and 10% on ELU-o. At iso-SRAM sizes, an accelerator using ACP always outperforms or matches the DMA baseline, regardless of prefetching, because the overhead of
manual cache invalidation and flushing is so great. The precise amount of this overhead is proportional to the ratio of model parameters to activations of the network. During inference, weights are read-only data, so a single cache flush at the very beginning of execution is sufficient to ensure coherency. However, intermediate activations may be modified by either the accelerator or the CPU at any time, so they must incur this overhead between every layer. Our networks exhibit a wide range of weights-to-activations ratios, ranging from 3:1 on ELU-0, to 13:1 on LeNet5 and CF-Simple, to 27:1 on ELU-1. In turn, this range creates varying amounts of software coherency overhead.

5.5.2 Integrating Camera Pipelines and DNNs

In this section, we demonstrate that most of the improvements just described are still attainable in a more realistic scenario, when multiple programs are concurrently running on the SoC. We evaluate three multiprogrammed mixtures: 1) One DNN + one ISP accelerator; 2) One DNN + all ISP accelerators; and 3) One DNN and one Halide program.

DNN with ISP HW accelerators

For multi-accelerator workloads, in theory we could test every combination of interface for every accelerator. However, the previous section demonstrated that using the ACP provides performance and energy benefits to the DNN accelerator across the board, so we don’t use DMA with it again. For simplicity, we restrict the ISP HW to all use the same interface. This creates three different integration scenarios:

A. DNN uses ACP with SW prefetch. ISP uses DMA.
B. DNN and ISP both use ACP without SW prefetch.
C. DNN and ISP both use ACP with SW prefetch.

In these experiments, we run all programs until the DNN (the longest running) has finished one input image. If any other program finishes first, it is restarted.
Simulations are warmed up for 1-5B cycles, depending on the size of the model parameters and inputs. Performance is measured in average cycles per input.

We begin this section by studying pairwise combinations of DNNs and ISP accelerators. Results across the three scenarios are shown in Figure 5.5.3, with DNNs in blue and ISPs in orange. For each program, its performance in isolation is shaded dark, and under contention, it is shaded light. DNN data is normalized to the first DNN bar and likewise for the ISP accelerator. We summarize the per-scenario observations:

Scenario A: Unsurprisingly, the DNN performs the best when it does not share the LLC. Under contention, its performance does not suffer much. However, the ISP accelerator performs the worst due to DMA overhead, with a slight slowdown under contention.

Scenario B: When the two accelerators share LLC without SW prefetching enabled, cache contention clearly penalizes DNN performance. However, the ISP accelerator is always faster than its corresponding perfor-
mance in scenario A. Different ISP kernels see varying speedup (from 5% on stencil3d to 20% on spmv-crs), because each has a different ratio of compute and communication time, and interfaces only affect data communication.

Scenario C: If both accelerators apply SW prefetching when sharing the cache, the DNN can recover most of its lost performance from Scenario B. For example, on stencil3d and fft-transpose, prefetching effectively erases any effects of contention on the DNN altogether, no matter which ISP program it is run with. ISP performance either improves (spmv-crs) or remains the same.

stencil3d is the only HW ISP accelerator not to benefit from prefetching (Scen. C), because it is a stencil pipeline whose input is the output of the previous stage, so the input data is likely already in the cache.

These results show that in multiprogrammed settings, sharing of the LLC is actually necessary to maximize total system performance. The remarkable effectiveness of SW prefetching for the accelerator holds under contention as well, helping the DNN achieve near-standalone performance and speeding up the ISP accelerators across the board. While prefetching could be implemented with additional hardware, this shows that a software-only solution is just as effective.

To see if these observations hold under heavier load, we next run a bigger multiprogrammed workload that contains a DNN and all three ISP accelerators. This mimics a complex visual application with multiple pipeline stages and distinct per-stage characteristics. Figure 5.5.4 shows that the prior observations still hold. In scenario B, the DNN slowdown is much more pronounced, reaching as high as 100% on CF-VGG compared to isolated performance. But scenario C shows that even under higher levels of contention, the DNN accelerator can achieve within 10% of its baseline performance. Meanwhile, the ISP accelerators continue to benefit from delegated coherency. stencil3d performance actually degrades in scenario C because as mentioned earlier, prefetching does not help this workload and creates additional L2 traffic. Overall, these results demonstrate that the
Figure 5.5.4: Each plot shows relative per-program execution time of a four-program mixture: one DNN and all three ISP accelerators. For example, the top row individually shows DNN, stencil, fft, and spmv performance when all four run concurrently, compared to when each of them run in isolation.

benefits of delegated coherency extend to realistic mobile SoC workloads. Applying software-assist mechanisms appropriately can effectively shield workloads running in systems under load from shared resource contention.

DNNs with ISP software

Halide is a language and compiler for writing high performance image processing code [141]. It has been shown to outperform many hand optimized implementations of core kernels like stencils, blurs, and FFTs as well as longer image processing pipelines. Owing to its success, it has seen wide adoption in commercial products. High performance ISP software is important across the spectrum of mobile devices, but it is of particular importance to the large number of low-end mobile devices, which often lack specialized hardware features present in high-end chips.

In this section, we run Halide programs together with DNNs to examine the interactions between ISP software and DNN hardware and compare these effects to the multi-accelerator results of the previous section. Since Halide programs are not HW-accelerated, we only evaluate scenarios B and C (without prefetching
for Halide since it doesn’t apply). And since camera-pipe represents a complex multi-kernel benchmark, we only run pairwise combinations of programs. Details about Halide programs are listed in Table 5.3.2.

Figure 5.5.5 shows the impact of shared resource contention. In contrast to the earlier study, interference effects are asymmetric. The CPU workload has very little effect on the DNN accelerator because they consistently exhibit L1 dcache hit rates of 99%, so they rarely affect the LLC (the shared resource). However, the DNN’s larger memory footprint causes the CPU’s LLC miss rate to jump from 2-10% to 15-50%, depending on which DNN is also running. Of the four DNNs, CF-VGG exhibits the worst cache behavior (Section 5.5.1) and therefore generates the most interference, as shown by the high L2 miss rates for this network. Halide slowdown
ranges from 10% (fft) to 50% (blur).

In summary, delegated coherency unlocks significant speedup and energy savings for accelerators compared to the standard DMA interface, without changing the accelerator datapath. Even when the system is under heavy load with multiple accelerators and CPUs sharing the LLC, a software prefetching mechanism enabled by the delegated coherency interface recovers most of the performance loss from contention, with most accelerators achieving within 10% or better of their standalone performance under all workload mixtures.

5.6 RELATED WORK

DNN ACCELERATOR DESIGNS There has been an incredible amount of interest in DNN hardware acceleration. Broadly speaking, the architecture community has focused on on designing efficient core datapaths and dataflows to maximize local reuse of data and functional unit utilization [4, 29, 31, 46, 86, 110, 114], exploit model sparsity and data quantization [67, 88, 131, 143], or explore technologies for analog computation [153]. Although these issues are highly relevant, they do not address integration challenges and system optimizations. Both of these receive scant attention in the research community and are the focuses of this chapter. In fact, only very recently have researchers began to investigate SoC co-design in the context of visual deep learning applications [19, 187], but none investigated delegated coherency.

SoC-ACCELERATOR INTERFACING Over the years, there have been a few publications investigating SoC-accelerator interfacing. Chung et al. proposed CoRAMs, an FPGA memory architecture composed of actively managed memory blocks that interact with external memory [34]. In contrast, delegated coherency is not FPGA-specific and could in fact be used as the external memory interface that CoRAMs abstract away. Sadri et al. investigated the performance of the ARM ACP on the Xilinx Zynq platform and found that a CPU and accelerator can cooperatively run an FIR filter on images faster and with less energy using it than DMA
[148]. However, this study only looked at simple filters, and the physical platform limited the ACP bandwidth to 2GB/s, which is far too low for contemporary vision workloads. Moreau et al. also used the ACP when building SNNAP because of lower round-trip latency [122], but they also used the Zynq platform and would also be bandwidth-limited.

**Shared Cache Management for Accelerators** The management of shared caches in multicore processors has been studied extensively by architects. Broadly, prior work falls into four categories: cache replacement policies [81, 140], cache partitioning schemes [108, 139, 169], cache bypass policies [49, 83], and cache prefetching [27, 126, 173]. Many of these prior efforts are designed for homogeneous multi-core processors and do not necessarily perform well in today’s highly heterogeneous systems. Rhu et al. proposed vDNN, a runtime memory management system to virtualize GPU memory for DNN training [146]. Our work differs in that we study on-chip SoC interfaces (rather than off-chip) across a range of multiprogrammed workloads, not just DNNs. We are unaware of any prior work that explores using software prefetching for on-chip accelerators through a shared cache.

More recently, with the proliferation of integrated CPU-GPU SoCs, researchers have investigated management of LLC sharing between CPUs and GPUs. Mekkat et al. proposed a mechanism to dynamically identify GPU memory accesses that are latency-insensitive and thus can bypass the LLC without significantly impacting performance [119]. Chen et al. combines cache bypass and GPU warp scheduling to minimize congestion for shared resources due to the sheer volume of memory accesses from the GPU [28].

Most of the work on shared cache management has focused either on homogeneous multicore processors or CPU-GPU systems, whereas we focus on less programmable accelerators. Such accelerators are often not even designed to tolerate fine-grained variable latency memory accesses and do not have a concept of thread/warp scheduling; it is more common to see fixed pipelines of data flowing through functional units. The rigidity of these accelerators removes many of
the degrees of freedom available to architects optimizing for CPU-GPU systems, thereby imposing new challenges.

Cache Coherence for Accelerators Research has extensively investigated cache coherence for heterogeneous systems. Topics of interest include optimization of data communication in CPU-GPU systems [72, 160, 162] and the integration of heterogeneous coherency protocols [3, 136], specialized coherency protocols for cooperating accelerators [97], and ensuring host system security with coherent accelerator caches [129]. Most of this past work has assumed full coherency in which the accelerator has a local cache. In contrast, our work exploits existing cache resources to bring the benefits of coherent memory to workloads that don’t necessarily want a private cache, thus avoiding the cost of an expensive and complex memory structure (tag arrays, TLBs, coherency complexities, etc).
Broad Acceleration in the Cloud

The previous chapters have been devoted to system-level analysis of specialized architectures running “killer applications”, wherein architects target the most costly kernels that consume the majority of total execution time for acceleration. This is a “deep” acceleration approach, and it is the approach taken by much of the existing literature on hardware specialization. In this chapter, we make the case that in the cloud, we can also accelerate multiple low-level routines that appear in many applications as a “broad” acceleration approach to collectively save CPU cycles across a fleet of datacenters. One promising candidate for broad acceleration is `malloc`: dynamic memory allocation. We show that a tiny in-core hardware unit can reduce `malloc` latency by up to 50% at merely 1500 μm² of silicon area.
6.1 Introduction

Despite the tremendous momentum in accelerator adoption in mobile SoCs, where the majority of silicon area in current designs is dedicated to specialized blocks [157], the server chips powering cloud workloads remain predominantly general-purpose. A major reason for this omission is that modern datacenter workloads are simply too diverse, without any opportunities for deep acceleration. Not only do they run thousands of different applications, but the individual workloads themselves have also been shown to not have significant hotspots that can be optimized with deep approaches [92]. This does not mean hardware acceleration in datacenters is infeasible. Characterization studies show that a large fraction of cycles is spent on the so-called “datacenter tax” – low-level routines like remote procedure calls, data serialization and memory allocation. While each individual component of this tax is a relatively mild hotspot (in the 2-8% range), together they can comprise up to 30% of all cycles in Google datacenters [92].

The ubiquity of the datacenter tax suggests a broad approach to acceleration: speeding up multiple shared low-level routines that appear in many applications. This approach may not provide the 10× application speedups typically associated with hardware specialization. But accumulating several instances of such several-percent optimizations can save significant amounts of CPU cycles, especially when deployed broadly across the hundreds of thousands of servers that cloud providers operate. Borkar refers to this approach as “10 × 10 optimization” [14] and argues that it is a necessity for continued performance increases in the era of dark silicon.

Of the components that comprise the datacenter tax, perhaps the most familiar one is malloc: dynamic memory allocation. malloc is such a popular programming paradigm that many collective developer-years have been spent researching and optimizing allocation strategies and techniques. For example, a typical malloc call takes only 20 CPU cycles on a current-generation general-purpose processor, setting the bar high for potential hardware implementations. malloc exemplifies the unique set of challenges facing broad acceleration: because calls to these routines tend to be very frequent, fast, and interspersed in-
side other application code, accelerators must be optimized for latency rather than throughput, and because each such accelerator brings a limited amount of overall application speedup, overheads must be kept to a bare minimum.

In this chapter, we present the design of Mallacc, a memory allocation accelerator that meets these constraints. Mallacc is a tiny in-core hardware block which accelerates the three primary operations of a typical memory allocation request: size class computation, retrieval of a free memory block, and sampling of memory usage. Mallacc is designed not for a specific allocator implementation, but for use by a number of high-performance memory allocators commonly found in datacenters today. Our goal is to make the already fast (20-30 cycle) malloc calls even faster, because they are so frequent, and Mallacc achieves that goal. It can reduce malloc latency by up to 50% while occupying less than 1500 \( \mu \text{m}^2 \) of silicon area. As we will show, Mallacc far exceeds the “1% speedup for 1% area” mantra that has informally guided processor development over the past decades.

6.2 Background

Dynamic memory allocation has been studied for decades. In this section, we place our work in the context of past literature. We discuss historical research on allocators, general techniques and structures that are still used in modern allocators, and factors that drove evolution of allocators over the decades.

At a very high level, a dynamic memory allocator sits between an application and the operating system (often as a part of the platform’s standard library). It requests continuous blocks of memory from the OS and distributes chunks of them, with different sizes, to call sites in the application that explicitly request them. Allocators are judged on both the speed with which they satisfy a request and their memory fragmentation, which measures how much memory is requested from the OS vs. how much memory the application actually uses.

In the very early days, main memory was expensive and scarce, so allocator design focused on minimizing memory fragmentation and overhead. Starting from the 1960s, researchers studied data structures for storing free objects, notably
linked lists [36] and trees [166]. Various strategies for searching through free lists of memory blocks to identify the right object to return were examined: such as returning the first block large enough (“first fit”), the exact size (“best fit”), and many more [35]. Techniques for efficiently splitting and coalescing free memory objects were also studied; one notable example is the buddy system, in which a free object can split into two “buddy” objects for small allocations, but can only be merged with that same “buddy” when a large allocation is needed [95]. The notion of size classes – allocating memory from a set of specific sizes – was also conceived decades ago [170].¹ Many of these techniques and data structures are still used in today’s allocators.

Over time, two trends motivated significant changes in allocator design. First, main memory costs dropped and densities increased exponentially thanks to Moore’s Law. However, unlike CPU speeds, main memory access latencies stagnated. The increasing gap between CPU and memory speeds shifted the focus from memory fragmentation to speed. Second, the rise of multi-core processors and multithreaded applications in the last decade motivated allocator designs that were fast and efficient in the face of problems like lock contention, false cache sharing, and memory blowup with large numbers of threads. Modern allocators like Google’s tcmalloc [57], FreeBSD’s jemalloc [52], Hoard [8], and others were all designed to support robust multithreaded performance.²

Modern multithreaded allocators like the ones listed above all share a common set of design principles. First, they logically organize available memory in a hierarchical fashion. The top level is a pool of memory that can only be accessed by a limited number of threads (often just one) to mitigate the cost of synchronization. These pools are highly optimized in software such that a hit in one is likely to be considered “fast enough”. They are backed by lower-level pools, which are shared among threads. Memory is migrated back and forth as necessary. Second, they select a set of size classes and round requested sizes to the next nearest size class,

¹Research in allocators has been especially prolific – for a significantly more complete survey of early approaches, refer to Wilson et al. [180].

²Similarly, Ferreira et al. [55] provide a succinct overview of the structure of modern allocators.
which simplifies splitting and coalescing of larger memory blocks and reduces the amount of metadata needed. Third, they use different methods to allocate “small” and “large” chunks of memory (though they differ on the exact thresholds of considering a chunk small). Finally, they ensure that memory can migrate from thread to thread to avoid memory blowup in scenarios where one thread allocates memory and another thread frees memory.

Within this framework of common design principles, modern allocators can differ significantly in their implementations. For example, size classes are selected based on different upper bounds of memory fragmentation. Heuristics for determining when to migrate memory from lower to upper levels, as well as how many blocks to move, vary greatly too. Lower level pools tend to store larger blocks of memory that are then sliced into smaller chunks for top level pools, which is a time-consuming process that requires synchronization. Similarly, at some point additional memory must be requested from the operating system, which requires a costly system call. Developers must balance the frequency of these requests with the overall memory usage and consider various allocation patterns from different applications. Therefore, the parameters of these procedures tend to change relatively frequently as developers seek out new optimizations and tradeoffs.

Compared to the effort spent on software optimization and tuning, creating custom hardware for allocators has received next to no attention. We are only aware of one feasibility study [105] and several variations of the buddy technique [22, 25, 26, 104], which show that it easily maps to purely combinational logic. While buddy allocation has been available for decades, modern allocators have converged to simpler techniques in their highest-level pools (most frequently, first-fit free lists), most likely due to buddy systems’ reported high degrees of fragmentation [180] and relative complexity.

This presents an opportunity for hardware designers looking to accelerate allocation. Rather than design a whole new algorithm from scratch to simplify hardware implementation, they can speed up the common elements of modern allocators – the “fast enough” top-level pools – and allow different allocator algorithms to tune the details on the lower levels in software for their own workload assump-
Figure 6.2.1: The costs of hits and misses in several allocation pools in TCMalloc vary by orders of magnitude.

The rest of this chapter, we demonstrate the feasibility of this approach by optimizing the top-level pools of TCMalloc [57]. While TCMalloc makes for a good anchor point to demonstrate gains – it is mature, robust and among the faster allocators [55] – the optimizations we propose can easily be used by other modern allocators.

6.3 UNDERSTANDING TCMALLOC

We start by describing how TCMalloc allocates and deallocates memory and compare and contrast it with other multithreaded allocators. We profile the costs of several allocator code paths and find that the fast path is an overlooked area for potential optimization.

6.3.1 TCMALLOC OVERVIEW

Allocation pools Like many other allocators, TCMalloc allocates memory from a hierarchy of memory pools. At the top are thread caches assigned to each thread of a process, and meant to service small requests (< 256KB). Each cache contains many singly-linked free lists – lists with addresses to free chunks of mem-
Figure 6.3.1: The steps that an allocation requests goes through. The colored boxes represent the major operations on the fast path, which we aim to optimize.

ory of the same size. There is one free list per size class. TCMalloc currently has 88 size classes, a relatively large number picked to keep memory fragmentation low. When a free list is not empty, a small allocation can be satisfied by simply popping the head off the list. Since these caches are thread-local, no locks need to be acquired and a thread-cache hit is relatively fast. jemalloc’s thread caches were inspired by TCMalloc [53], and their size class organization is quite similar.

If a free list is empty, the allocator must first fetch blocks into a thread cache from a next-level pool. In TCMalloc, it either attempts to “steal” some memory from neighboring thread caches, or gets it from a central free list. Both approaches require locking, and are orders of magnitude slower than hitting in a thread cache. Should both of these sources be empty themselves, TCMalloc allocates a span (a contiguous run of pages) from a page allocator, breaks up the span into appropriately sized chunks, and places these chunks into the central free list and the thread-local cache. Large requests (> 256KB) go directly to spans and bypass the prior caches. Should the page allocator also be out of memory, TCMalloc then requests additional pages of memory from the operating system.

Figure 6.2.1 illustrates the cycle costs associated with hitting or missing in several of these pools for 400 perlbench from SPEC CPU2006. It is a simulated distribution (details on our methodology follow in Section 6.5) of time spent in each malloc() call over the call’s duration in cycles. The three major peaks correspond to hitting in a thread cache, missing in a thread cache and hitting in the central free list, and grabbing a span. Missing in a thread cache has a cost at least three orders of magnitude higher than that of a hit. Because of the high costs, too many
misses in the highest-level pool can be detrimental to allocator (and application) performance. TCMalloc employs several heuristics to transfer chunks of memory between the various pools in an effort to maximize thread cache hit rates. These heuristics (and the particular implementation details of the lower-level pools) are what distinguishes different modern allocators from one another. Note, however, that despite their very low per-call cost, thread cache hits represent a significant chunk of allocator cycles overall for 400_parbench. We will come back to this observation in the following section.

**MEMORY DEALLOCATION**  Deallocation follows a similar path. When memory is being freed, TCMalloc first determines the size class of the freed object. If that object is small, it gets pushed to the top of the appropriate thread cache free list, and if that free list now exceeds a certain size (1MB), TCMalloc returns unused objects back to the central free list. If the freed object is large, pages of memory get returned back to the page allocator.

**6.3.2 Time spent in the allocator**

As discussed in the prior sections, research in allocator design has focused on the lower-level memory pools because of their potentially catastrophic effects on performance. This is also partially because the fast paths – those that hit in thread caches – are already considered sufficiently optimized. Microbenchmark experiments often support such a hypothesis. For example, our tp_small microbenchmark (described later) achieves an average malloc() latency of only 18 cycles.

However, we find that for a range of applications, time spent on the fast path is not only a significant, but also a major fraction of time spent in the memory allocator. Figure 6.3.2 shows this property for the four SPEC CPU2006 benchmarks that actually call the system allocator. In the cumulative distribution of malloc() time, more than 60% of time is spent on calls that take less than 100 cycles. For _xapian_, an open source search engine, we see an even higher fraction. This need not be the case for all workloads: for example, the performance tests of _maas tree_, a key-value store, never free any memory and end up continuously getting more
from the page allocator (which eventually goes to the operating system). A real
deployment of masstree does free memory and has much better thread-cache
hit rates, but even such corner-case behavior spends more than 30% of allocator
time on the fast path.

There are two main reasons for the high fraction of fast-path time that we ob-
erved. First, while individually cheap, fast-path calls can be very frequent – a clas-
sic “death by a thousand cuts” scenario. This is especially true for applications
that allocate and deallocate at similar rates so that their requests almost never have
to reach the other memory pool levels. Second, thread caches are very cheap in
microbenchmarks, but can get significantly more expensive when the requesting
application itself is cache-heavy. In that case, the application’s memory accesses
evict the allocator’s data structures from the CPU’s caches, and a cheap 18-cycle
fast-path call can turn into a hefty 100-cycle stall on main memory.

Thus, we believe the fast path of memory allocators presents an overlooked op-
opportunity for optimization and focus the rest of the chapter on speeding it up with
specialized hardware. For that, we need a detailed understanding of the work done
during fast path calls, and the costs associated with it.
Figure 6.3.3: Time spent in the three main components of the fast path accounts for $\approx 50\%$ of cycles.

6.3.3 Analysis of the Fast Path

By definition, the fast path is a memory request satisfied by a thread cache free list. And by design, an access on the fast path has little work to do. For TCMalloc compiled with GCC 6.1, the fast path is only $\approx 40$ static x86 instructions, and can take 18-20 cycles, assuming cache hits. It contains a few conditional branches that are easy to predict and no loops. Microbenchmarks with back-to-back allocations and deallocations can achieve an IPC of 3.0 on a 4-wide Intel Haswell core. In other words, it has been heavily optimized. Thus, speeding it up further is an exercise in performance microscopy and in reducing the latencies of the different steps of a fast allocation. Figure 6.3.1 illustrates these steps in the context of an incoming allocation request: 1) finding the appropriate size class for the requested size, 2) potentially sampling the request, and 3) satisfying it by popping the head of the corresponding free list. In the rest of the section, we go into more detail about the computation in each step and its cycle costs.

Note that we can (and do) estimate these costs, even if they are caused by only several instructions, because we rely on simulation. This is how we construct Figure 6.3.3, which contains cycle costs for several microbenchmarks designed to
size_t SizeClass(size_t size) {
    size_t class_index;
    if (size <= 1024)
        class_index = (size + 7) >> 3;
    else
        class_index = (size + 15487) >> 7;
    return size_class_table[class_index];
}

size_t class = SizeClass(requested size);
size_t alloc_size = size_table[class];

Figure 6.3.4: Size class lookup function.

stress different fast path aspects. With simulation, we can simply remove these
instructions from simulated execution and subtract the resulting cycle count from
a baseline. These are estimates, and not strictly additive, since out-of-order cores
explicitly overlap work. When all removed together (the Combined bars in Fig-
ure 6.3.3), they make up for half the cycles of the fast path.

SIZE CLASS CALCULATION  The first part of an allocation request rounds the re-
quested memory size to the nearest size class supported by the allocator. The num-
ber and spacing of size classes is carefully tuned to balance fragmentation and al-
locator latency, so typically the mapping from size to size class does not have an
easy closed form. For example, because small size classes are more commonly ob-
served, the spacing between small size classes is closer, and it grows the larger the
size class is. In TCMalloc, this mapping is implemented by first computing a size
class index from the requested size and then indexing into two arrays which are pre-
computed at initialization time for the size class and rounded size that it represents
(Figure 6.3.4). The class index only requires an add and a shift, but the two array
lookups can be comparatively costly, even if they hit in the L1 cache because they
are on the critical path of execution. The number of class indices (the size of the
first array) is set by the threshold for a small allocation and by memory alignment
requirements. This number was fixed at slightly above 2100 in 2007 when TC-
Malloc was open-sourced and has not changed. The second array is much smaller,
Currently at 88 (the number of size classes), and has seen two small increases since 2007.

Despite having 88 size classes available, we find that applications often use a relatively small subset. Figure 6.3.5 shows that, for the benchmarks we surveyed, all but one use less than 5 size classes on 90% of malloc calls. In fact, masstree almost exclusively uses a single size class.\(^3\) xalanckmk has a much broader distribution, but even so, it uses two size classes over half of the time. This observation motivates techniques to memorize the most common size classes.

While usually free is perfectly complementary to malloc and we rarely mention it, there is a marked difference in size class computation. free does not take a size parameter, only the pointer to be deallocated, so it must perform extra work to determine the size class to return it to. In TCMalloc, this is implemented by a hash lookup from the address being freed to the size class. This hash tends to cache poorly, especially in the TLB, leading to expensive losses. C++11 ameliorates this problem because the compiler can choose to call operator delete() with an extra parameter equal to the size of the object being freed, as long as the object’s size can be determined at compile time. With -fsized-deallocation, the compiler prefers calling that variant when it can. In our results, we assume sized delete when applicable.

\(^3\)For allocations below 256KB only, which are handled by the fast path.
**pop:**

```plaintext
load temp, MEM[head] ; Get the head.
load next_head, MEM[temp] ; Get head->next.
store MEM[head], next_head ; head = head->next.
return temp
```

**push:**

```plaintext
load temp, MEM[head] ; Get the head.
store MEM[head], new_head ; Set new_head as head.
store MEM[new_head], temp ; new_head->next = temp.
```

**Figure 6.3.6: Critical memory accesses on a free list push/pop.**

**Push/Push a Free List Head** Once a size-class is identified, all that is left is to pop (or push) the head of its free list. Pushing to or popping from a free list generates a dependent chain of three memory accesses, as shown in Figure 6.3.6. In these cases, the most critical operations are the two loads on the pop path, because long-latency load misses can stall execution and commit of younger instructions. Since calls to the allocator are interspersed among application code, the free lists are prone to eviction, making these loads likely to miss. Figure 6.3.3 demonstrates this clearly with the antagonist microbenchmark, which emulates such cache-trashing behavior, and sees a significant increase in Pop time. In contrast, stores misses are less likely to stall the execution or commit of younger instructions, making the deallocation path less performance-critical.

TCMalloc uses a trick to save memory taken up by the free lists: it stores the next pointer at the address of the block of memory it is about to return, instead of allocating a separate field in a struct for it. That is, *head is the value of the next pointer, rather than a more familiar list node with fields node->head and node->next. In addition to reducing allocator memory overhead, dereferencing head to get the next pointer has the side effect of prefetching the returned memory block itself, which can likely help the caller.

**Sampling** For monitoring and debugging purposes, TCMalloc can also sample allocation requests every N bytes. A sampled allocation dumps and stores a stack trace in addition to performing the allocation itself. Sampling is invaluable in a pro-
<table>
<thead>
<tr>
<th>Valid</th>
<th>Size range (index range)</th>
<th>Size class</th>
<th>Size</th>
<th>Head</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 - 1</td>
<td>1</td>
<td>8</td>
<td>ox8080</td>
<td>ox8080</td>
</tr>
<tr>
<td>1</td>
<td>63 - 64</td>
<td>25</td>
<td>512</td>
<td>ox9090</td>
<td>ox9290</td>
</tr>
<tr>
<td>1</td>
<td>5 - 6</td>
<td>4</td>
<td>48</td>
<td>oxo</td>
<td>oxo</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Figure 6.3.7**: A malloc cache with example values. The cache is searched by first an associative lookup over requested size and later by size class. It stores the corresponding size class, and the first two free-list elements for that size class.

...duction setting for analyzing memory usage and debugging memory leaks without having to stop, let alone recompile, live jobs, but it adds a measurable overhead to each malloc request, since a counter must be decremented and checked against the threshold each time.

**Remaining instructions** The three main steps described above account for \( \approx 50\% \) of fast path cycles. The remainder are split roughly evenly between: function call overhead (pushing / popping registers), addressing calculations (for example, of a free list in a thread cache) and updates to metadata fields (such as free list lengths and total size). While it is tempting to consider hard-coding the latter two in hardware, this would result in a rather narrow and inflexible accelerator, and severely limit its applicability to either other allocators, or even future revisions of the same allocator.

### 6.4 Mallacc: a malloc accelerator

Based on the characterization in the previous sections, we propose Mallacc, a fast-path malloc accelerator to augment a general-purpose processor. Broadly, Mallacc consists of a tiny dedicated malloc cache and a sampling performance counter. Its design requirements are extremely stringent. Since each fast-path call is on average only a few tens of cycles long, proposed structures must be inside cores, or access
def mcszlookup(ReqSize):
    IsHit = Cache.FindRangeContaining(ReqSize)
    if IsHit:
        SizeClass = Cache[ReqSize].SizeClass
        AllocSize = Cache[ReqSize].AllocSize
        ZF = 1
    else:
        ZF = 0
    return SizeClass, AllocSize

def mcszupdate(ReqSize, AllocSize, SizeClass):
    IsHit = Cache.FindSizeClass(SizeClass)
    if IsHit:
        SizeRange = Cache[SizeClass]
        if not SizeRange.Contains(ReqSize):
            SizeRange.LowerBound = ReqSize
        else:
            if Cache.Full():
                Cache.Evict()
            SizeRange = (ReqSize, AllocSize)
            Cache.InsertRange(SizeRange, SizeClass)

Figure 6.3.8: Pseudocode for size class instructions.

Latency will erase any gains, which implies very tight area constraints. In addition, we would like to hard-code as few allocator-dependent details as possible (ideally none), so that many current and future allocators can benefit from acceleration. Our proposed design demonstrates that it is possible to meet these constraints, and the rest of this section describes it in detail. Our descriptions assume the x86 architecture, but the general principles and mechanisms are not x86-specific.

6.4.1 The malloc cache

In Section 6.3.3, we identified size class computation and popping the head of a free list as the biggest contributors to fast-path cycles, especially with cache-heavy workloads invoking the allocator. We can optimize both with a tiny, two-part cache that we call the malloc cache. Conceptually, it learns the mappings from requested allocation sizes to both the size class they correspond to and the first two elements of the free list for that size class. In the case of a malloc cache hit,
putation can almost immediately return to the caller. By only storing the most frequently-accessed size classes, the cache can be kept extremely small (several entries). Lookups, updates and prefetches in the cache are software-managed, so it is also not tied to a particular allocator implementation. A four-entry cache, populated with some example values, is shown in Figure 6.3.7. We will go over its main components.

**Size Class Mappings**  By definition a single size class represents a range of allocation sizes that get rounded up and given the same amount of memory. The malloc cache learns and stores the mappings from a requested size range to the size class representing it.

When a requested size comes in, it is associatively checked with the upper and lower bounds of the ranges that currently make up all cache entries. If the request size is inside a range, the access is declared a hit, and the cache returns the size class and its corresponding rounded size. More interestingly, on a miss, execution goes to a fallback path, where software is left to compute the size class as it ordinarily would. Software is then responsible to update the cache with the new (requested size, allocated size, size class) entry. The cache either inserts a new size class entry with the new range, or it expands the range for an already existing size class. If the cache is full for an insertion, an old entry is evicted based on an LRU policy.

The cache is managed by two new instructions: `mcszlookup` and `mcszupdate` (malloc cache size lookup/update). `mcszlookup` takes the requested allocation size in an input register and returns the size class and allocation size in two output registers if the requested size is found in the cache. The zero flag (ZF) is set if found and cleared if not. `mcszupdate` takes the original requested size, the computed size class, and the allocation size in three input registers and either inserts a new entry into the cache or updates an existing one. No registers are modified. Pseudocode for the instruction mnemonics is shown in Figure 6.3.8. Figure 6.4.1 is an assembly snippet demonstrating how they integrate with the rest of allocator code.

Our actual implementation has one additional optimization – instead of keying
Start:
; raz = size class (dest)
; rbx = allocated size (dest)
; rcx = requested size (source)
mcszlookup rax, rbx, rcx ; Sets ZF
je ComputeSizeClass ; if ZF = 1, jump.
Resume:
; Continue with the rest of malloc.
ComputeSizeClass:
; The usual software calculation for the size class (raz)
; and allocated size (rbx). Then update the cache.
mcszupdate rcx, rbx, rax
jmp Resume

Figure 6.4.1: Integration of size class instructions in an allocator.

the array on the actual requested size range, we use the range of size class indices, as
defined in Figure 6.3.4, and add dedicated hardware to compute the index from the
requested size. Because the space of indices is significantly smaller than the space
of requested sizes, the cache can learn mappings faster, with fewer cold misses.
The hard-coded hardware adds an additional cycle of latency to the cache, which
we do model. It is the only TCMalloc-specific optimization in Mallacc, and can
be disabled with a configuration register. In this mode, the malloc cache will build
ranges of known requested sizes, although with slightly higher miss rates.

Free list caching An allocation call requires popping an element off a free
list. As mentioned in Section 6.3.3, this is the most performance-critical part of
a fast-path call, because it caches poorly and accesses memory prone to eviction
by the application’s own cache accesses. The malloc cache tackles this bottleneck
by storing copies of the head and the next head of the free list associated with a
size class alongside the size class mappings. Figure 6.3.7 illustrates this with an
example.

Storing the first two list items in the malloc cache allows a Mallacc-enabled al-
locator to immediately return a value to the application after a hit. It also allows
the next instruction in a linked list pop, the one that sets the head of the linked list
```python
def mchdpop(SizeClass):
    Found = Cache.FindSizeClass(SizeClass)
    if Found:
        Head = Cache.GetHead(SizeClass)
        Next = Cache.GetNext(SizeClass)
        if Head and Next:
            Cache.SetHead(SizeClass, Next)
            Cache.InvalidateNext(SizeClass)
        ZF = 1
        return Head, Next
    ZF = 0
    return NULL, NULL

def mchdpush(SizeClass, NewHead):
    FoundEntry = Cache.FindSizeClass(SizeClass)
    if FoundEntry:
        CurrHead = Cache.GetHead(SizeClass)
        Cache.SetNext(SizeClass, CurrHead)
        Cache.SetHead(SizeClass, NewHead)

def mcnxtprefetch(SizeClass, NewNext):
    CurrHead = Cache.GetHead(SizeClass)
    CurrNext = Cache.GetNext(SizeClass)
    if CurrHead and not CurrNext:
        Cache.SetNext(NewNext)
    elif not CurrHead:
        Cache.SetHead(NewNext)
```

Figure 6.4.2: Pseudocode for linked list instructions.

to the current next element, to commit immediately without waiting for an often-required L2 or L3 miss in order to first fetch that next element. We find that second effect especially important, because the long-latency miss often blocks other otherwise-ready instructions from Committing.

We introduce two new instructions to enable such operation. Most importantly, mchdpop (Figure 6.4.2) takes in the requested size class as an input (which we have ideally gotten from the previous optimization), and returns the cached copies of the first two list elements on a hit. If either of them is not present (NULL) in the cache entry, the access is declared a miss, the other one is also invalidated, and execution falls back on software to perform the pop (Figure 6.4.3). Its dual
malloc:
    ; rax = size class.
    ; rbx = location of the head of the free list.
    ; rcx = returned: head element.
    ; rdx = returned: next head element.
    ; rdi = temporary.
    mchdpop   rcx, rdx, rax   ; Search malloc cache.
    je    cache_fallback       ; If we missed, go to fallback.
    mov    QWORD PTR [rbx], rdx  ; Otherwise, update head.
    ; ...
    jmp    malloc_ret

    cache_fallback:
    ; Execute the original software.
    mov    rcx, QWORD PTR [rbx]   ; head = *freelist->head.
    mov    rdx, QWORD PTR [rcx]   ; next = *head.
    mov    QWORD PTR [rbx], rdx  ; freelist->head = next.

    malloc_ret:
    mcnxtprefetch  rax, QWORD PTR [rdx]   ; Prefetch the next head.
    ; Clean up stack and return value.

free:
    ; rax = freed pointer.
    ; rcx = size class.
    mchdpush  rcx, rax          ; Update malloc cache head.
    ; The rest of free

Figure 6.4.3: Integration of linked list instructions in an allocator.

operation, mchdpush, is invoked on the deallocation side and updates the cached
Head with the pointer being freed, shifting the previous head to the Next slot.

Note that these instructions are merely performance optimizations meant to
isolate free lists from cache antagonists. The real free list head pointer is always
valid and updated in software on both a hit and a miss, as is any metadata (length,
total size, etc.).

For a pop operation to consistently hit, we need two elements already cached.
To maintain that for differently-balanced allocation patterns (i.e., with different
rates of allocations and deallocations over time), we introduce yet another instruc-
tion, mcnxtprefetch. Conceptually, mcnxtprefetch prefetched a memory loca-
tion into the malloc cache’s Next entry, and is called after a pop hits and moves

127
its Next element in the Head position. In this case, a subsequent pop request can immediately hit as long as the prefetch has had enough time to return from the cache hierarchy. While not necessary for correctness, enabling a prefetch to update the Head field of an empty cache entry as well as the Next field allows for the prefetch instruction to be called on a miss, and leads to higher hit rates. We assume that in Figure 6.4.3. Finally, to ensure that the copies of the list elements stored in the malloc cache are always consistent (Head always points to Next), entries with an outstanding prefetch block and do not service pushes or pops until the prefetch comes back, or gets rolled back on a misprediction.

**Core integration** First, it is important to point out that the malloc cache only stores copies of list elements for fast access – the definitive version of free lists is always in regular memory. Thus, at interrupts or context switches, the whole cache can always be flushed without writebacks or correctness concerns. Similarly, at branch mispredictions, entries from the mispredicted epoch can be discarded, just as they would in any other long-latency unit.

Second, as part of the core, the malloc cache can potentially see instructions out-of-order. In order to not break the invariant that a cached Head’s next pointer always points to the adjacent Next element, our three linked list instructions are ordered with each other. We implement that by an implicit read-write register dependency through an architecturally-invisible register, which out-of-order execution has to observe. As discussed earlier, blocking the cache when a prefetch is outstanding is also required to preserve the linked list invariant.

Finally, the prefetch instruction is slightly unconventional. Like a software prefetch in L1, it is allowed to commit, so that it does not block subsequent code, but a result still has to make its way from the cache hierarchy to the malloc cache. From the core’s point of view, this is treated in a virtually identical manner to a store, which is also allowed to commit with an outstanding memory access, but reserves a slot in a structure (sometimes called a senior store queue), where it waits for an acknowledgment from coherency controllers.
6.4.2 Sampling

The sampling code in TCMalloc (and its equivalents in jemalloc [53] and others) presents an additional opportunity to remove several cycles from the allocation critical path. The operation performed by the sampler — accumulate a value and capture a stack trace at a threshold — is precisely what a performance counter does and what the perf_events subsystem performs when the performance monitoring unit (PMU) raises an interrupt on an event. We propose dedicating a hardware performance counter for sampling allocation sizes, which entirely removes a conditional branch on the fast path. Stack traces are only required when a user explicitly requests them, and this can be handled through the perf_events interface as it typically is currently.

The main difference between our proposal and current performance counters is that it will need to increment by the value of a register, which holds the requested allocation size. As a result, the PMU will need access to the actual register file (or just the ROB), instead of the more typical occupancy statistics. As the design of a performance counter is fairly straightforward, we will focus on the design of the malloc cache for the remainder of this chapter.

6.5 Methodology

To evaluate Mallacc, we ran simulations on two systems — a conventional aggressive out-of-order processor as a baseline, and the same processor equipped with Mallacc, as described in the previous section. We also performed limit studies on our optimizations for an optimistic upper bound of speedup. To do so, we ran simulations in which the instructions comprising the three steps from Section 6.3.3 are simply ignored by performance simulation.

Microbenchmarks To better understand allocator performance and the effect from our optimizations, we first constructed a suite of microbenchmarks to stress certain aspects of the fast path code. They are divided into two categories based on their allocation patterns: strided and Gaussian. Strided benchmarks allocate in
<table>
<thead>
<tr>
<th></th>
<th>cycle error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gauss</td>
<td>5.32</td>
</tr>
<tr>
<td>gauss_free</td>
<td>3.67</td>
</tr>
<tr>
<td>tp</td>
<td>12.3</td>
</tr>
<tr>
<td>tp_small</td>
<td>5.92</td>
</tr>
<tr>
<td>sized_delete</td>
<td>4.21</td>
</tr>
<tr>
<td>Average</td>
<td>6.28</td>
</tr>
</tbody>
</table>

Table 6.5.1: Simulator validation on malloc microbenchmarks.

increments of N bytes, up to some value, while Gaussian benchmarks issue allocation requests by drawing from normal distributions. All strided benchmarks fit perfectly in L1 caches and represent the very best baseline cases. Gaussian benchmarks allocate more and subsequently have larger working sets and more interesting caching behavior.

- **tp**: A throughput-oriented microbenchmark. It performs a series of back-to-back malloc and free calls, which always hit in thread caches. Each iteration strides through request sizes in increments of 16 bytes from 32 to 512 bytes.

- **tp_small**: Same as above, but we only stride up to 128 bytes. This ensures that: (i) each iteration accesses a different free list; and (ii) we only use four size classes. This microbenchmark captures the fastest possible fast path on the allocation side.

- **sized_delete**: A variant of tp_small that uses eight size classes and sized deletes to speed up deallocation.

- **gauss**: A more realistic allocation pattern. gauss chooses randomly between small (16-64 byte) and relatively large (256-512 byte) allocations. 90% of allocations are chosen from the small set to represent typical behaviors for strings and small lists. Within each range, the size is selected from
a Gaussian distribution. However, no objects are ever freed, which renders free lists virtually useless. This is a lower bound on the gains from any free-list centric optimizations.

- **gauss_free**: Same allocation behavior as gauss, but it randomly frees allocated memory with 50% probability.

- **antagonist**: Same allocation behavior as gauss_free, but after every allocation, invokes a simulator callback which evicts the less used half of each set of the L1 and L2 data caches. This mimics the behavior of an application that strides through a large working set, without simulating the millions of instructions required for the stride.

All microbenchmarks explicitly minimize the number of instructions between allocator calls (which is important when each call is only 40 instructions) and are run with sufficient warmup time.

**MACROBENCHMARKS**  We evaluate our optimizations on the four benchmarks from SPEC CPU2006 that use the system allocator and two workloads that are more likely to be found in datacenters. For datacenter-like workloads, we use the open-source search engine xapian and the key-value store masstree [117]. xapian is configured as a leaf node and performs searches on an index of 1.6 million English Wikipedia pages, as well as on a smaller index of the same number of page abstracts. The set of queries focuses on popular Wikipedia pages, obtained from Wikipedia's weekly top 25 article digests. For masstree, we run its wcoll and same performance tests. For SPEC benchmarks, we simulate several SimPoints [159] of 1B instructions each per benchmark, for xapian we skip query parsing and only simulate query execution, and for masstree we run from start until completion.

**ALLOCATOR**  We use TCMalloc at revision 050f2d. To model our proposed instructions, we annotate potential optimization sites in TCMalloc code by inserting
special x86 marker instructions. Later, in simulation we replace these instructions with our proposals. These marker instructions were carefully chosen to a) not already appear in TCMalloc and b) have the same number and type of operands as our proposed instructions. This is done so the compiler does not optimize surrounding code sub-optimally.

**Compiler** All benchmarks and allocators are built with GCC 6.1 at -O3 with -fsize-deallocation.

**Simulator** All experiments are run using the XIOSim simulator [90], configured for an aggressive out-of-order core modeled after an Intel Haswell microarchitecture. Since we are evaluating malloc fast path code, we validated our performance model on microbenchmarks against a Haswell desktop processor and achieved a mean error of 6.3% (Table 6.5.1). We omitted antagonistic because it uses a simulator callback to emulate cache trashing and does not run natively.

### 6.6 Results

#### 6.6.1 Allocator Time Speedup

Figure 6.6.1 shows the reduction of time spent in allocator code for our SPEC and cloud workloads. These results use a 32-entry malloc cache to demonstrate the potential of our accelerator; we will later present a cache size sensitivity study. On the total time spent in the allocator (including both malloc and free), Mallacc is able to achieve an average of 18% speedup, out of 28% projected by the limit study. Most of that is due to improvements on malloc calls, which see an average of nearly 30% speedup (Figure 6.6.2). The amount of speedup achieved is highly correlated with the fraction of time on the fast path shown in prior sections. We call out three particular benchmarks to get a deeper understanding of the causes for improvement, or lack thereof.
XAPIAN  xapian uses a very small set of size classes, and malloc calls almost exclusively take the fast path. As shown in Section 6.3, this is true whether it is searching over an index of small documents (abstracts) or an index of large documents (full articles). This makes xapian a great candidate for fast path acceleration and Figure 6.6.2 confirms that – the malloc cache provides over 40% speedup on malloc calls.

Figure 6.6.3 implies that the causes for this improvement are the latency-reducing portions of Mallacc – size class lookups, sampling, and, to a much smaller degree, linked list caching. It is a distribution of time in malloc calls over the call duration for three cases: the baseline implementation, our limit study, and Mallacc. The baseline case is already very fast – with virtually all calls between 20 and 40 cycles, not unlike our striding microbenchmarks, which implies very small effects from cache antagonism. Our best-case latency optimizations manage to reduce the average call length almost twofold, with median calls now at 13 cycles, and a distribution very close to that of the limit study. The size class cache in particular is very effective because of the small number of size classes used by xapian.
Figure 6.6.2: Improvement in time spent on malloc() calls (both fast and slow paths).

XALANCBMK  As demonstrated by Figure 6.3.2, xalancbmk uses the most number of size classes, requiring 30 size classes for 90% coverage. Nevertheless, it has enough size class locality to also benefit from Mallacc, achieving over 40% speedup on malloc calls. Figure 6.6.4 shows the malloc call duration distribution for this benchmark. The first spike corresponds to the fastest of fast path calls, where the effects are similar to those seen in xapian. The next large spike, between 20 and 70 cycles includes fast path calls that missed in L1 and L2 caches and had to go to L3 (34 cycles latency on Haswell). The malloc cache is particularly beneficial in this region because of its cache isolation properties. Finally, note that Mallacc only improves fast-path behavior without affecting slower calls.

MASSTREE  masstree has the lowest overall malloc speedup of all the workloads we tested. As we pointed out in Section 6.3.2, this is because the masstree performance tests never free any memory, so many malloc calls must request large amounts from the page allocator. The little time spent on the fast path results in an allocator time improvement of just 5%. However, a real deployment of masstree would inevitably free memory and likely have significantly higher
thread-cache use, so we would expect different results.

6.6.2 Sensitivity to malloc cache size

The malloc cache is a part of the core, where silicon real estate is expensive, so we must maximize performance gains with the least number of entries. To understand the effects of malloc cache sizing, we sweep malloc cache sizes from 2 to 32 on our suite of microbenchmarks. The results of this sweep are shown in Figure 6.6.5.

Unsurprisingly, we find that too small of a cache will result in slowdown rather than speedup. At a high enough miss rate, not only is execution going through the fallback paths (the same instructions that we started optimizing away), but also with the additional malloc cache lookups to determine that. However, once the cache is large enough to capture the majority of allocation requests, we quickly achieve speedup. One example are the strided benchmarks, which have no size class locality until we can capture all of their requests, resulting in very sharp jumps. sized_delete, tp, and tp_small use 8, 25, and 4 size classes, respectively, and we see that the speedup inflection points occur precisely at those malloc cache sizes. The Gaussian benchmarks have more size class locality because they are more likely to allocate small size classes, which results in a more gradual increase in speedup until cache size 12, because Gaussian benchmarks allocate from 13 possible size classes.

Once the malloc cache is sufficiently sized, Mallacc can achieve within 10-20% of ideal speedup. The lone exception is tp. For certain points of execution, this microbenchmark allocates and deallocates from the same size class in a very tight loop (≈ 30 cycles for a malloc-free pair). In this case, the malloc cache blocks until each of the malloc prefetches returns with a value, causing the slowdown. The prefetch instruction is based on exactly the opposite assumption – that there is enough time between requests to prefetch for the next one and this slowdown is expected. None of our macro workloads exhibit slowdown due to prefetch blocking.

It is important to remember that these microbenchmarks are designed to stress
Figure 6.6.3: Xapian sees a significant improvement on already-fast calls.

the fast path of malloc, not to exhibit realistic allocation behavior. As we showed in Figure 6.3.5, most benchmarks use a very small number of size classes. We swept malloc cache sizes and only xalan-cbmk is meaningfully affected by a smaller size – it loses 6 percentage points of allocator time improvement between 32 and 16. Because of that, we consider 16 sufficient for most workloads.

6.6.3 Full program speedup

Finally, we present improvements on full benchmark execution time, not only allocator time. This speedup is obviously bounded by the total time each benchmark spends in the allocator. Figure 6.6.6 shows these fractions for our workloads, compared to published data from Google’s datacenters [92]. Most of our workloads spend a much lower fraction of time in allocator code, so we can expect small gains. As mentioned before, the massertree performance tests have very high malloc time because they exclusively allocate memory and never free any, resulting in many slow path calls.

Table 6.6.1 shows full program speedup for workloads where the measured speedup through simulation is statistically significant. For them, the mean program speedup is 0.43%, with a maximum of 0.78% for perlbench.
Figure 6.6.4: Xalan benefits both from latency reduction and cache isolation.

Figure 6.6.5: Effect of cache size on malloc speedup.

Because absolute time in the allocator speedup tends to be small, run-to-run variance on some of the workloads is enough to mask out any improvements we achieved with the malloc accelerator. More precisely, we do not include the workloads for which a single-sided Student’s T-test fails to reject a hypothesis of full-program slowdown with 95+% probability. Note that for all workloads, the speedup in allocator code is always statistically significant: in Figure 6.6.1 the improvement in allocator time is always much higher than typical run-to-run variation (error bars), even so if we pessimistically add simulation bias (6% from Table 6.5.1) in the least favorable direction. Non-allocation code is unchanged be-
between our baseline and optimized experiments, but it dominates execution time, so even small random variations in simulating it can appear to mask all the gains in allocator code. This is why we prefer reporting gains in the allocator alone, where we can be certain in the significance of results.

6.6.4 Area cost of Mallacc

Mallacc consists of the malloc cache and a performance counter. The malloc cache requires 152 bits of storage per entry. Because the malloc cache is fully associative, it must be implemented using content addressable memories (CAMs) for lookup and standard SRAM for storage. We do not lay out the malloc cache to provide precise area estimates, but it is so small that a reasonable upper bound will suffice. Also, we ignore the area of the performance counter, since it is just one 64-bit register per hardware thread.

The malloc cache requires three CAM arrays to implement the index and size class search and LRU functions, while the rest of the data – allocated size and list pointers – can be stored in an SRAM array. The index CAM requires 24 bits per entry to store two 12-bit indices, while the size class CAM requires 8 bits per entry to store size classes, and the LRU CAM stores \( \log_2 n \) bits per entry, where \( n \) is the number of entries. The SRAM array requires 117 bits per entry to store two 48-bit pointers (currently, x86 only uses the lower 48-bits of 64-bit addresses), 20 bits for the allocated size, plus a valid bit. Our analysis has shown 16 entries to be sufficient for the workloads analyzed; this means the CAMs and SRAM are 72 bytes and 234 bytes, respectively.

We used CACTI 6.5+ [102] to estimate the sizes of these four arrays in 28nm. The CAMs collectively occupy 873 \( \mu \text{m}^2 \) and the SRAM occupies 346 \( \mu \text{m}^2 \) for a total of 1219 \( \mu \text{m}^2 \). This is certainly a pessimistic upper bound; Jeloka et al. recently demonstrated a 512 byte configurable CAM array occupying merely 1208 in 28nm \( \mu \text{m}^2 \) [82]. We scale published area numbers of shifters and adders (for the additional index computation) from accelerator models [157] by ITRS technology scaling factors and estimate a total area of 265 \( \mu \text{m}^2 \), bringing our upper
bound to about 1500 μm².

Consider this area in the context of a typical high-performance CPU. An Intel Haswell core measures 26.5 mm² (including private L1 and L2 caches). If integrated into a Haswell CPU, Mallacc is merely 0.006% of the core area. Pollack’s Rule states that historically, the performance increase of a chip is approximately proportional to the square root of the increase in complexity, where complexity refers to area [14]. By this rule, an area increase of 0.006% would only produce 0.003% speedup. In contrast, Mallacc demonstrates average speedup of 0.43%, which is over 140× greater. It is clear that Mallacc far surpasses the “1% performance for 1% area” rule of thumb that has informally guided processor development over the last few decades.
As hardware accelerators become first-class citizens alongside traditional programmable hardware like CPUs and GPUs and the number of specialized blocks on an SoC grows, system-level analysis and modeling will become increasingly important to extracting the most value from these blocks. This dissertation is one of the first in the architecture community to enable system-level accelerator research and demonstrate its impact on important workloads today. First, as the CPUs are a major contributor to overall system power, we presented a validation of a widely power modeling tool for multicore CPUs to enable accurate power modeling for general-purpose processors. Second, we presented an SoC simulator that enables performance and power modeling of complex accelerator-centric systems running end-to-end workloads. Third, we showed how leveraging coherency features of SoC accelerator interfaces can greatly improve accelerator performance and energy efficiency without changing the core datapath. Finally, we demonstrated how
broad acceleration for workloads running in large distributed systems can reduce latency of memory allocation requests by 50% at a negligible area cost.

The work in this dissertation describes some initial steps in the quest to make hardware specialization ubiquitous across all computing platforms, but much work remains. Here we discuss three aspects of accelerator design which still need much additional investigation:

1. **Programmability**: One of the biggest risk of building custom hardware is that by the time it is ready to be deployed, new features or functionality will be required that the accelerator cannot support. While certain classes of accelerators are fully or highly programmable, such as GPUs and DSPs respectively, they still cannot match the efficiency of fixed-function accelerators. Moving forward, we must develop architectural mechanisms to allow fixed-function accelerators to adapt or offload unsupported work to a more programmable element. This will require innovations in specialized programming paradigms, data sharing mechanisms, and more.

2. **Composability**: Currently, many accelerators are designed to be monolithic, in that they are not intended to communicate directly with other accelerators. Enabling lightweight and reconfigurable inter-accelerator communication mechanisms can enable efficient pipelines of accelerated components, virtualization of accelerator resources, and greater flexibility in mapping workloads to an accelerator substrate.

3. **Expanding broad acceleration**: While memory allocation is a promising candidate for broad acceleration, there are many other components to the datacenter tax, with different latency vs throughput requirements, that are also amenable. A single accelerator for one component of the datacenter tax may not provide enough overall performance wins to convince silicon vendors to integrate it, but if it were combined with several others, the potential performance gains could be too much to ignore. Also, the components of the datacenter tax may also vary across cloud providers, so there could be even more similar low-level routines waiting to be identified.
References


[126] Kyle J Nesbit, Ashutosh S Dhodapkar, and James E Smith. Ac/dc: An adapt-

[127] Stephen Neuendorffer and Fernando Martinez-Vallina. Building Zynq ac-
celerators with Vivado high level synthesis. In FPGA, 2013.

[128] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border

[129] Lena E Olson, Mark D Hill, and David A Wood. Crossing guard: Mediating

[130] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Au-
tomatic thread extraction with decoupled software pipelining. In Interna-
tional Symposium on Microarchitecture, 2005.

[131] Anghuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli,
Rangharajan Venkatesan, Brucek Khailany, Joel S. Emer, Stephen W. Keck-
ler, and William J. Dally. SCNN: an accelerator for compressed-sparse con-

[132] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSSx86:
A Full System Simulator for x86 CPUs. In Design Automation Conference
2011 (DAC’11), 2011.

[133] Rajan Patel. Google lens: real-time answers to ques-
tions about the world around you, 2018. URL https://www.blog.google/products/google-lens/
google-lens-real-time-answers-questions-about-world-around-you.

[134] Aashish Phansalkar, Ajay Joshi, and Lizy K John. Analysis of redundan-
cy and application balance in the spec cpu2006 benchmark suite. In Interna-

[135] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural Sup-
port for Address Translation on GPUs. ASPLOS, 2014.

[136] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beck-
mann, Mark D Hill, Steven K Reinhardt, and David A Wood. Heteroge-
neous System Coherence for Integrated CPU-GPU Systems. In MICRO,
2013.


