UNIFIED MODELS TO STUDY MULTICORE SCALING LIMITS AND ACCELERATOR-BASED SOLUTIONS

by

Emily R. Blem

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2013

Date of final oral examination: 08/09/13

The dissertation is approved by the following members of the Final Oral Committee:
Karthikeyan Sankaralingam, Associate Professor, Computer Sciences
Mark D. Hill, Professor, Computer Sciences
Shan Lu, Assistant Professor, Computer Sciences
Jeffrey T. Linderoth, Professor, Industrial and Systems Engineering
Doug Burger, Ph.D. Microsoft
Michael J. Schulte, Ph.D, AMD

To my grandparents:
Richard & Dorothy Test,
and William & Marguerite Blem.

ACKNOWLEDGMENTS

I would like to thank all of the people who helped me complete this dissertation.

A huge thanks goes to my advisor, Karu Sankaralingam. He not only gave me the opportunities and collaborations to get my research on this path, but also gave me the confidence to pursue it. He taught me how to distill my work into papers and talks that the community could follow and find interesting, and how to approach ideas, and prior work, critically.

I would also like to thank my committee for all of their feedback and support. In particular, I would like to thank Mark Hill for his interest in and feedback on papers and talks throughout my time at Wisconsin. I would like to thank Doug Burger for his involvement in the dark silicon modelling work and all of his encouragement. I would like to thank Michael Schulte for teaching me as a young graduate student and helping me find my research interests. I would like to thank Jeff Linderoth for letting me work on a course project that became a precursor to some of the ideas in this dissertation. Shan Lu's interest in the work and many questions helped push me to delve further into the ideas in this work.

I would also like to thank the other members of the architecture faculty, David Wood and Guri Sohi, for their support and feedback throughout my career. Mary Vernon's training in analytic models has been indispensable.

I have enjoyed working with all of the members of the Vertical Research Group. In particular, I would like to thank Jai Menon for all of the technical expertise he has shared. I would like to thank Venkatraman Govindaraju, Raghuraman Balasubramanian, and Tony Nowatzki for all of the paper drafts they read and feedback they provided. I thank Marc de Kruijf for his guidance as a senior member of the group. I would also like to thank Newsha Adralani and Vijay Thiruvengadam for their help with benchmark development, and Matt Sinclair for helping me learn about GPU experimentation.

I would also like to thank my co-authors at other institutions, Hadi Esmaeilzadeh and Renée St. Amant, for the many long discussions and fruitful collaborations.

I am grateful for the support I received from the Cisco Systems Distinguished Graduate Fellowship and the ECE Claude and Dora Richardson Distinguished Graduate Fellowship. This research was also supported in part by NSF grants. I would also like to thank the Wisconson Architecture Affiliates for their insights and interest in my work. In particular,

I would like to thank Greg Wright and Chris Vick at Qualcomm for providing me the opportunity for a wonderful internship.

My friendships with fellow graduate students have been indispensible. In particular, Natalie Enright Jerger, who first provided advice when I visited as a prospective graduate student and I am lucky to have as both a mentor and a friend to this day. Sean Pieper, Chuck Tsen, and Suman Mamidi all helped me find my way early in my career. For constant friendship and support as well as good food, I would like to thank Nathan Rosenblum. For all of his advice, acting as a sounding board for my research ideas, and friendship, I would like to thank Joe Meehean. From college, I would like to thank Kristina Pao for being a constant friend, travel companion, and study partner and Seth Jacobson for encouraging me to take my first Computer Architecture course.

Antonia Massa-MacLeod, Sarah Cunningham Bannen, and Julie Simons - the three of you provided so much support, good times, and relaxation, and I can't thank you enough.

Many thanks to my parents, Chuck and Kathy Blem, for always supporting me in my decisions while providing quiet guidance when those decisions needed a little refinement. Your love, support, and ready offers of trips to the Oregon Coast have all helped me get to this point. Thanks, too, to my brother, Seth Nielsen, for letting me tag along on adventures growing up and showing me how to use the Commodore 64. My entire extended family needs to be thanked for all of their interest (feigned or not) in my work, and willingness to just assume that I knew what I was doing.

Finally, Matt Fredrikson, thank you for keeping me going, day in and day out, and always pushing me to go a little further. I am so lucky to have met you, and look forward to starting our post-graduate school lives together.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

UNIFIED MODELS TO STUDY MULTICORE SCALING LIMITS AND ACCELERATOR-BASED SOLUTIONS

Emily R. Blem

Under the supervision of Associate Professor Karthikeyan Sankaralingam
At the University of Wisconsin-Madison

Trends in microarchitecture, reliability, and devices are precipitating accelerated exploration of new architectural approaches. These rapid changes necessitate tools to rapidly evaluate new architectural ideas on specific benchmarks. We observe that previous work in models has focused on either models for *architectural flexibility* or *accuracy*. In this dissertation, we develop three modeling approaches that explore the trade-off between architectural flexibility and accuracy for CPUs and GPUs. First, we develop an upper-bound model to use in design space exploration to find the impact of power, area, and performance trade-offs as power and area change at different rates to aid in resource planning for multicores. Second, we refine previously developed custom core models to accurately predict performance on real hardware to understand performance bottlenecks and facilitate expansion of those models to incorporate microarchitectural changes. Finally, we develop an approach to predict a new architecture's performance using data collected on another architecture without developing code for the second architecture would be even more useful for cases where the second architecture is not yet available or the programmer overhead to convert code is high. Through these three approaches, we find new possibilities for modeling in the architectural flexibility and accuracy trade-off space.

Karthikeyan Sankaralingam

ABSTRACT

Trends in microarchitecture, reliability, and devices are precipitating accelerated exploration of new architectural approaches. These rapid changes necessitate tools to rapidly evaluate new architectural ideas on specific benchmarks. We observe that previous work in models has focused on either models for *architectural flexibility* or *accuracy*. In this dissertation, we develop three modeling approaches that explore the trade-off between architectural flexibility and accuracy for CPUs and GPUs. First, we develop an upper-bound model to use in design space exploration to find the impact of power, area, and performance trade-offs as power and area change at different rates to aid in resource planning for multicores. Second, we refine previously developed custom core models to accurately predict performance on real hardware to understand performance bottlenecks and facilitate expansion of those models to incorporate microarchitectural changes. Finally, we develop an approach to predict a new architecture's performance using data collected on another architecture without developing code for the second architecture would be even more useful for cases where the second architecture is not yet available or the programmer overhead to convert code is high. Through these three approaches, we find new possibilities for modeling in the architectural flexibility and accuracy trade-off space.

1 INTRODUCTION

Trends in microarchitecture, reliability, and devices are forcing computer architecture toward a cross-roads. This cross-roads necessitates accelerated exploration of new architectural approaches. As a result, tools to rapidly evaluate new architectural ideas on specific benchmarks are needed. These tools could serve a variety of roles to aid in the rapid development of new architectures.

This dissertation considers mechanistic models that fill different use-cases for the rapid exploration of new architectures. These mechanistic models consider core performance as complex systems that can be broken into individual components and their interactions. This flexible approach is applied, through three different mechanistic models, to address several use-cases throughout this dissertation.

The use-cases for these models are wide-ranging. The ability to see the impact of power, area, and performance trade-offs as power and area change at different rates would aid in resource planning for multicores. Accurate processor models validated on real hardware would allow architects to understand performance bottlenecks and, if straight-forward, extend those models to incorporate microarchitectural changes. Predicting a new architecture's performance using data collected on another architecture without developing code for the second architecture would be even more useful for cases where the second architecture is not yet available or the programmer overhead to convert code is high. In this dissertation, we focus on modeling tools that are tailored to addressing each of the use-cases above.

In this introduction, we discuss the dissertation's goals, give an overview of the work that is included in the dissertation, highlight the contributions of the work, and give an overview of the document's organization.

1.1 Goals

This dissertation's goal is to understand the performance implications of new architectures using analytic models of the architectures with key benchmark characteristics for rapid design evaluation. It includes three approaches, each geared toward a different use-case, that each contribute separately toward this goal.

This dissertation develops a set of models that each extend previous work to explore new domains. We focus on central processing unit (CPU) and graphics processing unit (GPU)

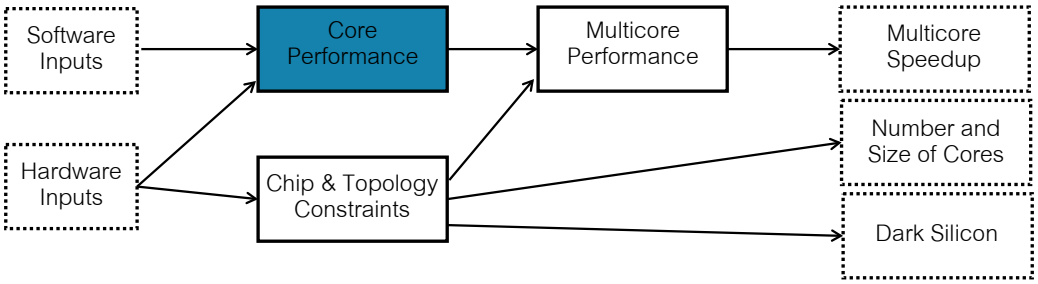


Figure 1.1: Multicore and core model interaction. Key dissertation contributions are to the shaded core model. Inputs and outputs are in dotted boxes.

architectures as they are an interesting and challenging use-case, but are also developed enough that tools are available for us to validate our results. This is not a limitation as the approaches we develop could be used for other emerging architectures. In addition to the use-cases discussed for each model, the dissertation includes a running example in each chapter to compare CPU and GPU performance projected to future technology nodes for individual benchmarks. This running example demonstrates how each model increases the realism of the projections.

1.2 Overview

In this section, we give an overview of the multicore model approach used in this dissertation, highlight that the dissertation's focus is on the core model component, and discuss the three different core modeling approaches used in this dissertation.

As part of a running example throughout this dissertation, we consider a modeling framework to predict the fraction of a chip's area that must go unused due to power constraints (*dark silicon*). As shown in Figure 1.1, three key modeling pieces and two sets of inputs are combined to produce three outputs: the projected multicore speedup, the number and types of cores, and the fraction of the chip that is dark silicon. The software and hardware inputs may vary depending on the core model from a very small number (e.g., five workload characteristics) to a much larger set. The three model components are the core performance model, chip and topology constraint model, and multicore performance model. The chip and topology constraint model and the multicore model are primarily modeled with simple Amdahl's Law based equations [3, 49] that are discussed briefly in Chapter 2; they are not

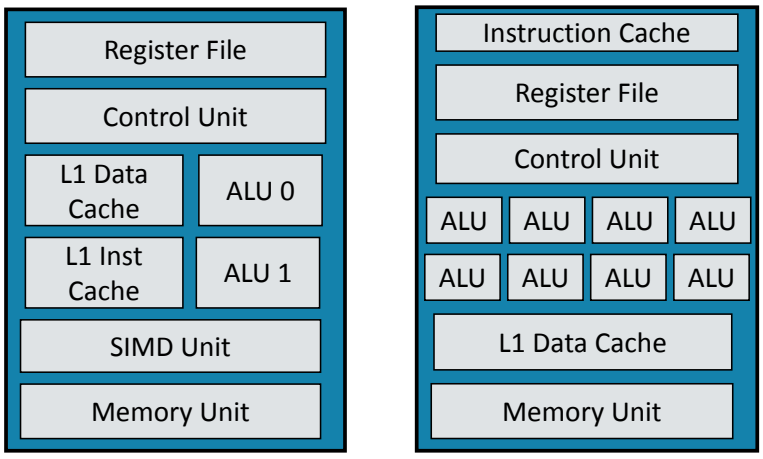


Figure 1.2: S-Core and M-Core Summary.

Definition 1.1 (Cores)**.** A *core* consists of, at least, one or more execution units, a register file, a control unit, and a memory unit. We consider two types of cores: S-Cores and M-Cores.

Definition 1.2 (S-Cores)**.** An *S-Core* is a single-threaded CPU-like core, with a general purpose architecture.

Definition 1.3 (M-Cores)**.** A *M-Core* is a many-threaded GPU-like core, with hundreds or thousands of threads.

---

a focus of this work.

We next introduce several terms used throughout this dissertation to describe the model domain and how each model is evaluated. We consider *analytic mechanistic models*, which model complex systems (computer chips, in this dissertation) by considering the components of the system that impact performance and how those components interact; these effects are modeled using a set of equations that use system characteristics as inputs. They are constructed using a detailed knowledge of the systems, and may be refined through empirical data. In this work, we focus on CPU-like and GPU-like chips. CPU-like chips are multicore chips with general purpose cores designed for high single-threaded performance. GPU-like chips are general purpose GPU chips designed for high many-threaded performance. We define the cores that we consider from each chip in Definitions 1.1- 1.3 and Figure 1.2.

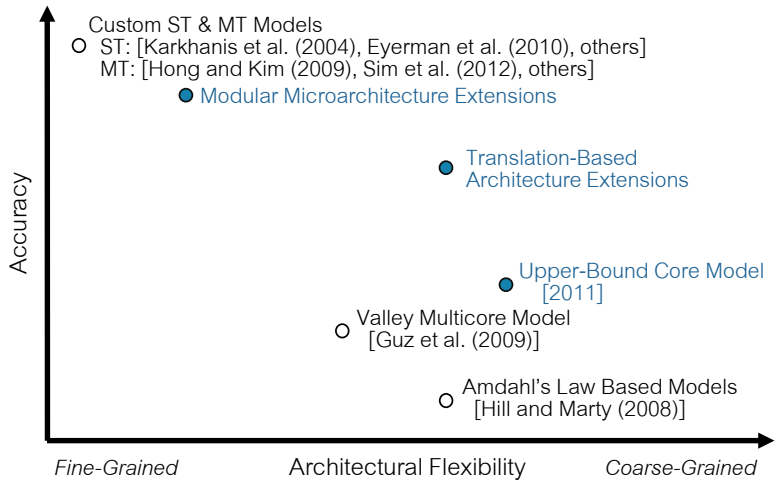


Figure 1.3: Expected Core Model Trade-offs. Contributions are shown in blue. Quantified trade-offs are given in Chapter 6.

Definition 1.4 (Architectural Flexibility)**.** *Architectural flexibility* refers to the range of multicore architectural features that are captured with a single model. Architectural flexibility can be either coarse-grained or fine-grained. *Coarse-grained* flexibility includes flexibility at the highest level, including S-Core versus M-Core styles and the chip topology (symmetric, asymmetric, and other heterogeneous styles). *Fine-grained* flexibility includes more micro-architectural design features such as the cache sizes, issue-widths and styles, execution units, and pipeline depth.

Definition 1.5 (Accuracy)**.** *Accuracy* refers to how close the model's predicted performance is to the performance directly measured on hardware. We consider performance in terms of execution time in ns and execution time speedup over a baseline (generally, a single-threaded Nehalem or Sandybridge core). Our accuracy descriptions include both quantitative and qualitative-trend comparisons.

A single M-Core may have multiple processing units that execute in parallel (usually referred to as streaming processors); an S-Core may also have a SIMD unit that executes on multiple data elements at once but also will have scalar execution units. S-Cores and M-Cores may appear on the same chip, and not all S-Cores or M-Cores on a chip need be identical. The chip design space is designed in more detail in Chapter 2.

Although mechanistic models tend to include only first order effects, the level of detail and versatility can vary. A key theme in our description of the models in this dissertation is the trade-off between architectural flexibility and model accuracy. The most beneficial

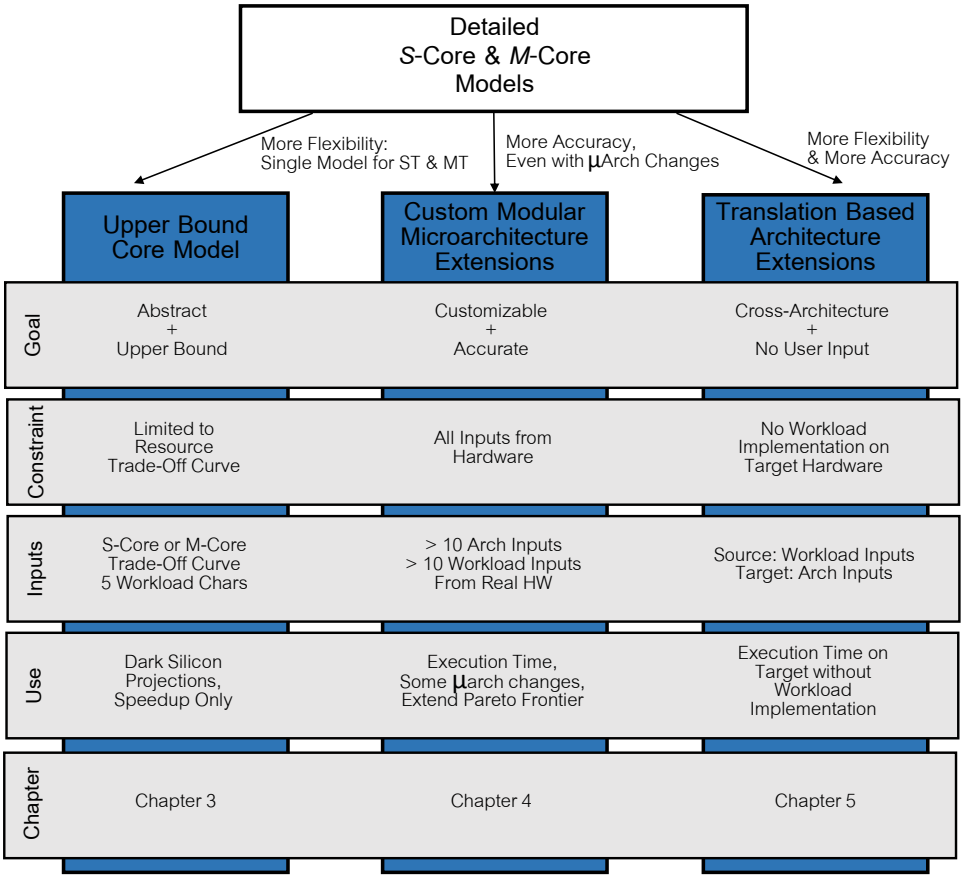


Figure 1.4: Core Models Overview. Contributions are shown in shaded boxes.

mechanistic model for a given use-case is dependent on the desired trade-off level. The three models in this work are placed in context with previous work in terms of these trade-offs in Figure 1.3. In the figure, the x-axis depicts the relative architectural flexibility and the y-axis describes the relative accuracy, which are defined below the figure. We revisit the figure in Chapter 6, where we replace the estimates in Figure 1.3 with measured data from this work.

1.2.1 Core Models

This dissertation is built off of the rich related work in mechanistic models of both *S*-Cores and *M*-Cores [57, 34, 50, 89]. As shown in Figure 1.4, we have completed three key model extensions built off of the detailed *S*-Core and *M*-Core models in prior work: an Upper-

Bound Core Model, Customizable Modular Microarchitecture Extensions, and Translation-Based Architecture Extensions. In the figure, we briefly describe the goals, constraints, inputs, and use-cases for these models. Below, we describe these contrasts in more detail.

The three core models in this dissertation each have a different set of assumptions and intended uses, and as a result pick a different architectural flexibility and model accuracy trade-off point. The first core model in this thesis, the upper-bound model in Chapter 3, focuses on models with high architectural flexibility but low accuracy. The second model in this thesis, the customizable modular microarchitecture extensions in Chapter 4, focuses on models with high accuracy but low architectural flexibility. These two models provide a backdrop for the flexible and accurate models in Chapter 5 which leverage insights from the previous models. The three approaches highlight trade-offs between model prediction accuracy and the architectural flexibility, and are discussed in more detail below.

Upper-bound Core Model: The upper bound model is the most abstract model that we consider; it abstracts away components of the mechanistic model until converging on a single model that can be used for both S-Cores and M-Cores. The goal is to be able to move from a simple resource trade-off curve (e.g., power/performance Pareto frontier) to a per benchmark speedup prediction using only a few real program characteristics. We assume that we have the resource trade-off function, some knowledge of the architecture (S-Core or M-Core style and the processor frequency at trade-off function end-points), and five workload-specific inputs. These workload-specific inputs can be the same for both S-Cores and M-Cores. We can use this model to predict optimistic results about the speedups using only minimal information about the cores; this makes the model a perfect fit for dark silicon projections where we do not have information about the specific processors. The abstraction brings with it a severe limitation: the microarchitecture is described with a single performance score, so changes to the microarchitecture cannot be accurately modeled. The coarse-grain architectural flexibility limits more fine-grained architectural flexibility, making this approach most useful for upper-bound scalability studies.

Modular Microarchitecture Extensions: The custom modular microarchitectural extensions allow accurate performance predictions even with small changes to the microarchitecture. The goal is to predict a specific workload's performance on real hardware using run-time statistics. We use only real hardware, with performance counter and binary

instrumentation (or emulation, for GPUs). These models allow us to accurately predict performance for real hardware, can be used to model the impact of small changes to the architecture, and in the case of dark silicon projections, can be used to predict the future technology scaled performance of multicores built from cores that lie beyond the Pareto frontier. The usage of the S-Core and M-Core modular custom models is equivalent to that of an architectural simulator (e.g., SimpleScalar [17]): they can be used out of the box, the effort to make architectural changes ranges from minimal for parameterized features to more extensive for new features, and the knowledge required to implement changes ranges accordingly. The approach therefore increases the architectural flexibility of mechanistic models. The models are detailed enough for higher accuracy than the upper-bound model, and intermediate values in the model provide insights into architectural bottlenecks.

Translation-based Architecture Extensions: Diverse architectures (e.g., CPUs and GPUs) require different analytic models and different inputs for accurate performance modeling. The translation based architectural extensions return to the upper-bound models' attempt at a single model for both S-Cores and M-Cores, but, recognizing the significant differences in S-Core and M-Core architectures, instead only attempts to use a single set of inputs for both models. Inputs gathered for a CPU are translated to use with a GPU model, and vice versa. The goal of this approach is to use performance characteristics measured on one architecture to predict performance on another. The potential users for this approach includes both architects and programmers. New architectures require significant hardware or simulator implementation work and heavy programmer involvement to realize performance benefits, yet the rapid evolution of hardware presents time-sensitive implementation challenges for both groups. Architects would benefit from early analysis of new designs without full implementation as well as a way to observe the impact of future technology trends on current designs. Programmers would benefit from a tool to understand which code to port and to which architecture. This approach provides all of the flexibility of the custom modular microarchitectural extensions, with additional flexibility in the overall thread organization. There is an accuracy cost in the translation process, but it is smaller than the cost in the upper-bound model.

In this section, we summarized our three models which are described in more detail in Chapters 3, 4, and 5. In the next section, we describe the contributions of each of these models and how they extend our ability to pick between different modeling goals.

1.3 Contributions

This dissertation develops a series of mechanistic performance models to predict chip performance with varying levels of architectural flexibility and model accuracy. The goal is to produce a high fidelity, cross-architecture performance projection mechanism for current and emerging cores and benchmarks that does not require programmer involvement. The contributions of this work are succinctly described by Figure 1.3, which shows how the work in this dissertation fills several points in the trade-off space between architectural flexibility and model accuracy which previous models have failed to explore.

Specifically, the dissertation makes the following contributions:

- Distillation of mechanistic models to unified upper-bound models that cover a diverse architecture space
- General modular approach to architecture-specific mechanistic models
- New mechanistic model modules to accurately predict performance on real hardware
- Identification of architecture-independent qualities of important performance characteristics
- Cross-architecture performance prediction using architecture-independent measurements and no user intervention
- Application of the three different models to dark silicon projections to improve understanding of future technology challenges

The goal of this work is to expand the tool-set available to architects for rapid exploration of new ideas. Early computer architects used analytic mechanistic models to quantify a wide range of architectural design problems at a time when full system simulation was difficult. Today, with the increasing complexity of both hardware and software, we expect a renaissance in the utility of analytic mechanistic modeling as a tool for computer architects. The models in this dissertation are a step toward adding analytic mechanistic modeling tools back into the architect's performance evaluation tool-set.

1.4 Summary of Findings

The evaluation in this thesis has two components: model accuracy for each set of models, and the empirical results from related case studies.

- Upper-Bound General Model:
  - Over a subset of the PARSEC benchmark suite, the average speedup predicted using the upper-bound general model is within 10% of the measured speedup for two S-Core chips in two configurations each.
  - Over a subset of the benchmarks included with GPGPUSim, the average error in the speedup predicted using the upper-bound general model is less than 42% with between one and 120 M-Cores as compared to simulated performance.
  - Using the upper-bound general model with the PARSEC benchmarks, we predict that even at the 8nm technology node, the average speedup will only be 3.7× to 7.9× that of a 45nm Nehalem quad-core chip. Using the modular microarchitecture extensions with the CAB benchmarks[1], between 52% and 81% of the chip is lost to dark silicon, only 19 to 42 cores are used, and geometric mean-speedup is between 5.2× and 20.4× at 8nm.
- Modular Microarchitecture Extensions:
  - Over the CAB benchmark suite , the modular microarchitecture extensions improve accuracy by up to 1.7× for S-Cores and by up to 1.6× for M-Cores.
  - Even with improved accuracy, per-benchmark errors can range from -53% to 148% from S-Cores and from -46% to 653% for M-Cores.
  - On average, errors for S-Cores and M-cores are less than 50% for benchmarks that have measurable speedups.
  - Using the modular microarchitecture extensions with the CAB benchmarks, between 20% and 81% of the chip is lost to dark silicon, more than 60 cores are used, and geometric mean-speedup is between 2.5× and 20×. The increased diversity in projections is a result of the more realistic per benchmark performance variability.

[1] A set of compute-intensive, highly data-parallel benchmarks developed in this dissertation work and discussed in Chapter 2.

- Translation-Based Architecture Extensions:
  - S-Core to M-Core translation is sensitive to the number of warps, but with the correct number of warps, errors are similar to those for the custom model.
  - M-Core to S-Core translation model is as accurate as the custom model when speedups exist.
  - Errors introduced by models such as the translation-based architecture extensions do not significantly influence projections at future technology nodes.

1.5 Organization

Chapter 2 presents the basic multicore model, the design space under consideration, and the methodology used throughout the dissertation. The three main contributions of the dissertation are organized into Chapters 3- 5, where each chapter includes the related prior work, a description of the modeling contribution, validation of the approach, and application to a running example, projections for dark silicon:

Upper-bound General Model Chapter 3 describes the high-level upper-bound model for CPUs and GPUs which sacrifices accuracy for generality.

Modular Microarchitecture Extensions Chapter 4 describes custom CPU and GPU models. It includes detailed descriptions of the prior work for both and the contributions of this thesis to the models. It concludes with analysis of both models on a range of real hardware with real benchmarks. In addition to the running projections for dark silicon example, it includes an application to an instruction set architecture (ISA) study.

Translation-Based Architecture Extensions Chapter 5 describes a novel translation approach to modify measurements from one architecture to use as inputs for performance models for another architecture. It discusses translation from CPUs to GPUs and the resultant accuracy, translation from GPUs to CPUs and the resultant accuracy, and concludes with a discussion of key principles for extending the approach to other accelerators.

Chapter 6 presents conclusions, and highlights potential applications to other architectures through an example.

2 FRAMEWORK AND METHODOLOGY

This chapter describes the framework on which the core models are built and evaluated: the design space under consideration, the multicore model, the benchmarks, measurements, and hardware used for evaluation, and the framework for a running example, the application of the models to dark silicon projections. The decisions in this chapter reflect the desire to cover a wide hardware design space, perform thorough validations on real hardware, and maintain relevance in the evaluation workloads. The chapter largely describes well-known, state-of-the-art techniques that are applied throughout the remaining chapters.

The chapter's content is largely summarized in two tables: Table 2.1 covers the multicore assumptions discussed in the first half of the chapter, and Table 2.2 summarizes the infrastructural decisions discussed in the second half of the chapter.

Table 2.1 outlines the design space and explains the roles of the cores during serial and parallel portions of applications for CPU- and GPU-like chips. Single-thread (S) cores are uni-processor style cores with large caches and many-thread (M) cores are throughput cores with smaller caches (e.g., an SM from a GPU). These were defined in Chapter 1 and are discussed in more detail in the next subsection.

Table 2.1: Multicore Topology Descriptions

Topology	Serial Mode	Parallel Mode
Symmetric	1 Lightweight S-Core	N Lightweight S-Cores
Asymmetric	1 Heavyweight S-Core	1 Heavyweight S & N Lightweight S Cores
Dynamic	1 Heavyweight S-Core	N Lightweight S-Cores
Fused	1 Heavyweight S-Core	N Lightweight S-Cores

(a) CPU: Single-Threaded (S) Cores

Topology	Serial Mode	Parallel Mode
Symmetric	1 M-Core Thread	N M-Cores
Asymmetric	1 Heavyweight S-Core	1 Heavyweight S & N M-Cores
Dynamic	1 Heavyweight S-Core	N M-Cores
Fused	1 Heavyweight S-Core	N M-Cores

(b) GPU: Multi-Threaded (M) and S Cores

Table 2.2: Framework Summary

Chapter/Model	Topologies		Benchmarks	Measurements	Validation
3. Upper-Bound	Sym/Asym/ Dyn/Fused	S	PARSEC	prior work[1]	Xeon/i7 (HW)
		M	GPGPUsim	GPGPUsim	Pre-Fermi (Sim)
4. Modular μ-arch	Sym	S	CAB/Rodinia	HW/Pin	A8/Atom/Xeon/i7(HW)
		M	CAB/Rodinia	HW/Ocelot	Pre-Fermi/Kepler (HW)
5. Translation-Based	Sym	S	CAB/Rodinia	HW/Ocelot	Atom/i7 (HW)
		M	CAB/Rodinia	HW/Pin	Pre-Fermi/Kepler (HW)

Table 2.2 gives an overview of the evaluation methodology used for each model: the topologies evaluated, benchmarks used, measurements taken, and cores validated against. Although we have attempted to use the same approach across all models, there are some inconsistencies due to the time frame when the work was done and the differing design goals at the time.

The rest of this chapter is structured as follows. We start with our design space goals and the multicore model framework that defines many of our design decisions. In the infrastructure chapter, we discuss our benchmark, hardware, and other implementation decisions. Finally, we describe how performance projections from the next three chapters are used to generate dark silicon projections as a running application example, and then conclude. This chapter can be used as reference.

2.1 Design Space Goals

The goal of the design space is to cover the widest array of current and future topologies as possible to maintain relevance at future technology nodes. The design space is broken into two pieces, as is this section: (1) the chip level (e.g., how cores are combined on a single piece of silicon), and (2) the core level (e.g., what kinds of cores are used). Consideration of "uncore" components such as caches, off-chip memory bandwidth, on-chip interconnection network, and memory characteristics are not an explicit goal of this dissertation and as such are not discussed below. However, they can contribute to performance bottlenecks, and are included in the core models as appropriate.

2.1.1 Chip Level

In this section, we describe the chip topologies that we consider and the types of chips that we believe is included in this design space.

CPU versus GPU: The two mainstream classes of multicore organizations under consideration, multicore CPUs and many-thread GPUs, represent two extreme points in the threads-per-core spectrum. The CPU multicore organization represents general-purpose heavy-weight multicore designs oriented toward high single-thread performance as found in a range of devices from smart phones to desktops. The GPU multicore organization represents GP-GPU lightweight cores with heavy multi-threading support and poor single thread performance.

Topology: In addition to the design choice between CPU or GPU chips, there are also different chip configurations, or *topologies*, possible. For each multicore organization, the dissertation categorizes multicore designs into four topologies: symmetric, asymmetric, dynamic, and composed (also called fused). These topologies may have cores of different sizes: *lightweight* cores are smaller, simpler cores while *heavyweight* cores are larger and more performance-oriented. The topologies are discussed below.

- **Symmetric multicores** consist of multiple identical copies of a core. All resources, including the power and area budget, are shared equally across all cores.
- **Asymmetric multicores** consist of one heavyweight monolithic core and multiple identical copies of a lightweight core. The high-performing heavyweight core improves performance for serial portions of the code and the lightweight cores plus the heavyweight core improve performance for parallel portions of code.
- **Dynamic multicores** consist of the same core configuration as asymmetric multicores. To reduce power requirements, during parallel code portions, the heavyweight core is shut down and, conversely, during the serial portion, the lightweight cores are turned off and the code runs only on the heavyweight core [19, 92].

- **Composed/Fused multicores** consist of a collection of lightweight cores that can logically fuse together to compose a higher-performance heavyweight core to improve performance of the serial portion of code [60, 54].

Heterogeneous configurations like AMD APUs and Intel Sandybridge[2] combine the CPU and GPU designs on a single chip. The asymmetric and dynamic GPU topologies resemble those two designs, and the composed topology models configurations similar to AMD Bulldozer. The methodology implicitly models heterogeneous cores of different types (mix of issue widths, frequencies, etc.) integrated on one chip by allowing any combination of cores on a chip.

2.1.2 Core Level

As summarized above, *S*-Cores (single-threaded) are uni-processor style cores with large caches and *M*-Cores (many-threaded) are throughput-oriented cores with typically smaller caches. We consider a range of both *S*-Cores and *M*-Cores to cover a wide-design space to mimic the many architectural solutions currently being considered. The range of cores of interest are briefly discussed below.

S-Cores

In this section, we will describe our design space goals in selecting *S*-Cores. Our specific *S*-Core choices are discussed in the infrastructure section that follows. Before discussing *S*-Core goals, we briefly define some general terms that are used throughout the dissertation:

- **Mobile *S*-Core:** CPU-style cores that are designed specifically for power-efficiency, and tend to be lower performance as a result
- **Desktop *S*-Core:** CPU-style cores that are designed primarily for high-performance, and tend to require more power as a result

Note that both classes of cores may be energy-efficient, since energy is a function of both power and performance, and that the trade-offs may be application dependent.

The range of *S*-cores could potentially range from power-optimized lower performance mobile *S*-Cores to performance-optimized desktop *S*-Cores. The mobile *S*-Cores tend to

[2]When we later consider Sandybridge cores, we are only considering the CPU multicore portion.

be in-order dual-issue cores with relatively small structures to keep die area and power low (e.g., a Cortex-A8). Desktop S-Cores tend to be out-of-order, superscalar cores with larger structures to increase performance (e.g., a Nehalem or Sandybridge). These cores may have different cache sizes, issue widths, processor models, ISAs, execution units, memory bandwidths, and memory sizes, among other differences. In Chapters 4 and 5, we assume that S-cores have SIMD units. In all chapters, for validation purposes, we pick only a few representative cores.

M-Cores

Similarly to the S-Core section, we will describe our design space goals in selecting M-Cores below. Our specific M-Core choices are discussed in the infrastructure section that follows. Before discussing M-Core goals, we briefly define some general terms that are used throughout the dissertation:

- **SP:** the smallest unit of a GPU, an SP performs a single operation on data from a single thread.
- **SFU:** also a per thread unit, an SFU can perform more complex operations (like computing transcendentals).
- **SM:** a streaming multi-processor, consisting of multiple SPs and SFUs, warp scheduler(s), a register file, and potentially a cache; an SM is an example of a M-Core.
- **Warp:** a set of 32 threads that execute in lockstep in an SM.
- **Pre-Fermi:** the earliest generations of Nvidia's general purpose GPUs, with the fewest performance optimizations. Each SM has only eight SPs, two SFUs, and one warp scheduler. There is 16KB of shared memory per SM.
- **Fermi:** Nvidia's second generation of general purpose GPUs, featuring warp-width enhanced caching and warp-coalescing to reduce memory bottlenecks. Each SM has 32 SPs, four SFUs, and two warp schedulers. There is 64 KB of memory per SM, plus a shared 768 KB L2 cache.

- **Kepler:** Nvidia's most recent generation of general purpose GPUs, with 192 SPs per SM, 32 SFUs, and four warp schedulers. There is 64 KB of memory per SM, and 1536 KB of L2 cache.

GPUs, in particular, have a set of terminology that is still evolving. In this dissertation, we tend to use Nvidia-style terminology for clarity as it is currently the most commonly used, although we recognize that other terminology may be clearer [48].

The range of M-cores, similarly, could range from simple pre-Fermi cores with few SPs per SM all the way to Kepler architectures with thousands of cores. The range of pre-Fermi to Kepler architectures includes changes in SIMD and SFU widths, on-chip caching, the number of instructions issued per cycle, and memory access coalescing. These differences are discussed in more detail in Chapter 4. In Chapter 3, we consider only pre-Fermi M-Cores; Chapters 4 and 5 consider both pre-Fermi and Kepler M-Cores. The cores used for validation are discussed in Section 2.3.3, and the cores used for the running dark silicon example are discussed in Section 2.4.

2.2 Multicore Model Framework

This dissertation includes performance projections for multicore chips. In this section, we describe the multicore model: the chip topologies and constraints considered, the Amdahl's Law based multicore performance model, and the multicore speedup calculation. Below, we discuss multicore models with area and power constraints; we use this approach both to compute multicore speedup and as part of the dark silicon projections discussed later in this chapter and used as a running example throughout this dissertation. This work is discussed further elsewhere [28].

2.2.1 Multicore Topology and Chip Constraints

In this section, we describe the chip topologies under consideration and how they interact with chip constraints. Note that in Chapter 3, we consider all four topologies: symmetric, asymmetric, dynamic, and fused. Since the focus of Chapters 4 and 5 is on the core models themselves, there we only consider symmetric cores. This is not a limitation of the approaches in those chapters; the other topologies could also be modeled there. However, since we validate on real hardware, validation on non-symmetric cores would be a challenge.

The number of cores, N, that fit in the chip's resource budget is dependent on the types of cores used, the uncore components included (e.g., a second level of cache), and the chip topology. The type of cores used ($q_{d,th}$) affects the resources required based on the known performance/resource constraint trade-off ($R(q_{d,th})$). Equations for chip constraints have been constructed for area constrained chips [49] and for area and power constrained chips [28].

Asymmetric, dynamic, and fused multicore topologies look similar at the performance level, but are significantly different at the resource constraint level. Asymmetric cores have one large core and multiple smaller cores, all of which always consume power. The dynamic core addresses the needs of power-constrained chips: the large core is disabled during parallel code segments and the smaller cores are disabled during serial code segments. The fused core addresses the needs of power- and area-constrained chips: the smaller cores are fused together to form one larger core during serial execution. Although the fused topology primarily affects the resource budget, it may have performance impacts: fused cores may not perform as well as similarly sized serial cores in asymmetric or dynamic topologies.

2.2.2 Multicore Performance

Amdahl's Law [3] is extensively used to generate upper-bound multicore speedup projections. Hill and Marty's extensions study a range of multicore topologies [49]. They model the trade-off between core resources (r) and core performance as $perf(r) = \sqrt{r}$ (Pollack's Rule [84]). Performance of various multicore topologies is then a function of r, $perf(r)$, and the fraction of code that is parallelizable. This dissertation uses a similar approach, but uses a core model to find the performance of each core.

The core model component finds serial (single-threaded) performance (P_1) and parallel performance (P_∞) of an either CPU or GPU multicore. Two performance bottlenecks limit P_1 and P_∞: computation capability and bandwidth limits. In the upper-bound model, these quantities are found separately while in the custom S-Core and M-Core models they are found as part of the single-core performance calculation. Equations showing the approach for the upper-bound model are listed below; they are trivially modified by removing the minimum calculation for the other models.

The maximum computation performance is the sum of the performances for each core on the chip, the $P_C(q_{d,th}, T)$ values. By definition, for completely serial code, this is just

Table 2.3: Topology-specific P_1 and P_∞ Calculations

Topology	P_1	P_∞
Symmetric	$\min(P_C(q_{L,S},1), P_B(q_{L,S}))$	$\min(N \times P_C(q_{L,S},1), P_B(q_{L,S}))$
Asymmetric	$\min(P_C(q_{H,S},1), P_B(q_{H,S}))$	$\min(P_C(q_{H,S},1) + N \times P_C(q_{L,S},1), P_B(q_{L,S}))$
Dynamic	$\min(P_C(q_{H,S},1), P_B(q_{H,S}))$	$\min(N \times P_C(q_{L,S},1), P_B(q_{L,S}))$
Fused	$\min(P_C(q_{H,S},1), P_B(q_{H,S}))$	$\min(N \times P_C(q_{L,S},1), P_B(q_{L,S}))$

(a) CPU

Topology	P_1	P_∞
Symmetric	$\min(P_C(q_{L,M},1), P_B(q_{L,M}))$	$\min(N \times P_C(q_{L,M},T_M), P_B(q_{L,M}))$
Asymmetric	$\min(P_C(q_{H,S},1), P_B(q_{H,S}))$	$\min(P_C(q_{H,S},1) + N \times P_C(q_{L,M},T_M), P_B(q_{L,M}))$
Dynamic	$\min(P_C(q_{H,S},1), P_B(q_{H,S}))$	$\min(N \times P_C(q_{L,M},T_M), P_B(q_{L,M}))$
Fused	$\min(P_C(q_{H,S},1), P_B(q_{H,S}))$	$\min(N \times P_C(q_{L,M},T_M), P_B(q_{L,M}))$

(b) GPU

the performance of a single core, $P_C(q_{d,th},T)$. For parallel code, $P_C(q_{d,th},T)$ is generally multiplied by the number of active cores.

During serial and parallel phases, core resources are allocated based on the topology as described above and in Table 2.1; serial performance (P_1) and parallel performance (P_∞) are thus computed based on the particular topology and organization's execution paradigm. For CPU organizations, topology specific P_1 and P_∞ calculations are in Table 2.3a.

For GPU organizations, the symmetric organization is similar to a simple GPU. Heterogeneous organizations have one CPU-like core for serial work and GPU-like cores for parallel work as show in Table 2.3b.

2.2.3 Multicore Speedup

Per Amdahl's law [3], system speedup is $\frac{1}{(1-f)+\frac{f}{S}}$ where f represents the portion that can be parallelized, and S represents the speedup achievable on the parallelized portion. Following previous work [49], we expand this approach to use performance results from the previous section.

To find the overall speedup, the model finds P_1 and P_∞ and a baseline core and topology's (e.g., a quadcore Nehalem's) serial and parallel performance as computed using the multicore

performance equations above (P_{B1} and $P_{B\infty}$, respectively). The serial portion of code is thus sped up by $S_1 = P_1/P_{B1}$ and the parallel portion of the code is sped up by $S_\infty = P_\infty/P_{B\infty}$. The overall speedup is then:

$$Speedup = 1/\left(\frac{1-f}{S_1} + \frac{f}{S_\infty}\right) \tag{2.1}$$

2.3 Infrastructure

While the previous section described the modeling framework within which the core models in this dissertation are used, in this section we focus on the infrastructure which we use to implement and evaluate the models. We first describe our benchmark selection process; benchmark selection is complicated by our requirement that we have identical workloads for both S-Cores and M-Cores to evaluate the translation model. Finally, we describe our measurement strategy on real hardware, how the models are implemented, and our validation process.

2.3.1 Benchmarks

Four benchmark suites appear throughout this dissertation: the PARSEC benchmark suite [10] and a set of GPU benchmarks distributed with GPGPUsim [7] in Chapter 3, and, in the remaining chapters, the Rodinia benchmark suite [20] and our Cross-Accelerator Benchmarks (CAB) a set of custom benchmarks with high data-parallelism. These benchmarks are primarily computation- and memory-bound; we leave I/O-bound workloads for future work. The remainder of this subsection describes the rational for each suite and, in particular, the criteria for and writing process for the custom benchmark suite.

Overview In Chapter 3, we are concerned with primarily with multicore performance and require a benchmark with realistic multicore performance. We thus use measurements from the PARSEC benchmark suite as it is a multi-threaded CPU benchmark suite with detailed published performance studies [10, 9]. However, since PARSEC does not include CUDA implementations of benchmarks, the M-Core validation in that chapter uses a set of GPU benchmarks distributed with GPGPUsim [7].

In Chapters 4 and 5, we want to use benchmarks where both S-Core and M-Core implementations are available. The best solution would include both hand-optimized SSE/AVX code and hand-optimized CUDA code for each platform for fair comparisons between the two types of accelerators. Suites with hand-optimized SSE/AVX code and hand-optimized CUDA code have not been publicly released, although some work has been done in the area by Intel [68, 87]. Instead, we were forced to write our own benchmarks[3].

CAB Benchmark Suite We chose the set of custom benchmarks–the Cross-Accelerator Benchmarks (CAB)– to construct a comprehensive set of benchmarks that included significant parallelism potential, used simple single-function kernels for analysis ease, and spanned a range of applications. After considering benchmark suites from multiple sources [68, 83, 87, 91, 10, 7], we decided on nine of the 14 benchmarks from Lee et al. [68]. The five excluded benchmarks are a constraint solver (rigid body physics), GJK (collision detection), radix-sort, tree search, and bilateral filter (image processing). The constraint solver and GJK are both large benchmarks that require gather/scatter operations, which are not supported in the SSE and AVX implementations to which we have access [4]. The optimized radix-sort from the CUDA SDK had subtle errors that caused Ocelot to fail. From our exploration, the performance of the CPU implementation of tree search did not improve with SSE instructions; we believe most performance improvements would be from multi-threading. Bilateral is a non-linear filter, which includes transcendental operations, which are outside the scope of this work. These were less common in other representative benchmarks than our other more complex benchmarks, Monte Carlo, LBM, and Ray Casting. Other suites all either include more complex kernels, which are less helpful for stressing high throughput machines, or focus on streaming media applications which are more amenable to other accelerators. The benchmarks in this suite are described in more detail in Appendix A.

Rodinia Benchmark Suite As a set of "challenge" benchmarks, we include results for GPU performance using a subset of the Rodinia benchmark suite. Accuracy on these benchmarks is expected to be lower than on the hand-written benchmarks as they tend to include more complex kernels. Further, SIMD performance is not expected to be accurate for these

[3]Benchmark code development performed with Vijay Thiruvengadam and Newsha Ardalani

[4]Gather instructions are supported with AVX2, and scatter instructions are supported on Intel's new MIC co-processor.

benchmarks as it is computed using auto-vectorized code compiled with ICC and with all loops marked with `pragma simd`. These results provide a more real-world application of the approach. Note that no model refinements were performed based on results from this suite - all tuning was performed using the simpler set of benchmarks described above.

DSL-based Alternatives Note that we could have used OpenCL to generate CUDA and SSE code as an alternative to writing the above benchmarks, but we found this approach to have several limitations. Both Parboil [91] and Rodinia [20] benchmark suites include OpenCL implementations. Studying the Parboil and Rodinia benchmark OpenCL code, we found that of the 11 Parboil benchmarks, six use optional OpenCL features and would require algorithmic changes to be compatible with Intels OpenCL distribution[5]. We also analyzed `SPMV` and `SGEMM`, for which we have both hand optimized SSE code and OpenCL code. The hand-optimized SSE code had higher speedups than the OpenCL code in both benchmarks. We also could have used OpenCL to implement our GPGPU applications, as it provides most of the same hardware abstractions as CUDA. However, as shown by [35], though there is no fundamental reason why OpenCL doesn't perform as well as CUDA, OpenCL can lose performance due to compiler and runtime differences.

2.3.2 Measurements and Implementation

We briefly outline our measurement methodology below; additional details, as necessary, are in the relevant chapters. We conclude the subsection with a discussion of the debugging process used throughout this dissertation.

Input Measurement In Chapter 3, model workload inputs for the PARSEC benchmark suite are from published papers [10, 9]; specific inputs derived from those papers are listed in Appendix B. In Chapters 4 and 5, model inputs come from performance counters and either binary instrumentation (S-Cores, using Pin [71]) or emulation (M-Cores, using Ocelot [25], since binary instrumentation is not available for them). Both binary instrumentation and emulation are significantly faster than simulation and are also more platform-independent.

[5] One uses special image processing instructions and the remaining five use atomic or synchronization instructions.

The CAB benchmarks and Rodinia are both instrumented with start and stop macros to collect data only from the key computation kernels.

Implementation Models from all three chapters are implemented in Python. Using workload inputs from either csv files (Chapter 3) or directly from hardware counter, binary emulation, and emulation outputs (Chapters 4 and 5) and hardware inputs from csv files, the models are written in a modular fashion and require no user-intervention; the entire process in Chapters 4 and 5 is push button.

Refinement and Debugging In Chapters 3 and 4, we build our work off of previously published models, and in 5, we find a new way to compute the inputs to those models to allow us to use inputs collected on a completely different architecture. In all three situations, we find that some model refinement is necessary. For these mechanistic approaches, model refinement is an iterative process that includes finding the source of error, finding a new way to model that component once it has been identified, and computing the new result to evaluate the solution.

Identifying the source of error can require some creativity: since we are not using a simulator, we are limited to the information that we can collect using performance counters and Pin. Techniques that may be useful for architects adapting a model to their own system include:

- **Identify trends:** group benchmarks by common characteristics (e.g., branch misprediction rates, number of memory accesses, unusual instructions) and check to see if the set has a higher error rate than disjoint sets
- **Work backwards:** assuming that all but one input is correct, use the known performance to find what that input would need to be for higher accuracy. If the predicted input is consistently different than the measured input, either the measurement is incorrect or it is being used incorrectly.
- **Measurement studies, optimization guides, and white papers:** Validating model assumptions against published measurement studies, optimization guides, and architectural white papers can illuminate incorrect assumptions in the model.

2.3.3 Validation

In this dissertation, we evaluate on real hardware when possible to mitigate any errors or simplifications from simulator models; we further describe our approach in this section. In particular, we evaluate against state of the art hardware as possible; as a result, we use slightly different architectures in Chapter 3 than in Chapters 4 and 5. For S-Cores, all model validation is on real hardware. For M-Cores, validation is on real hardware in Chapters 4 and 5; in Chapter 3, M-Core validation is via simulation.

Error Reporting For each benchmark and core considered, we report the speedup of that combination over the reference performance. The reference performance is the scalar implementation of the benchmark run on a baseline core. For "measured" speedups, both measurements are taken on hardware[6] . For modeled speedups, the reference speedup is still measured on hardware; this eliminates any error normalization that could occur if the reference speedup was also modeled. For individual benchmarks, we report the error, quantified as (`predicted speedup` − `measured speedup`)/`measured speedup`. Finally, we report average absolute error across benchmark suites; average absolute error is found by taking the absolute value of each error estimate and then finding the mean of the individual measurements. In this work, average absolute error is preferable to other approaches, such as root-mean-square-error (RMSE), as it does not penalize outliers more than smaller errors but still prevents positive and negative errors from canceling each other out.

Simulator Effects Although running on real hardware introduces challenges including repeatability and system overheads, we believe it is preferable to simulation as simulators can also have errors. We expect that many un-modeled effects in our models are also un-modeled in simulators; we thus expect that when our modeled performance is not accurate, the modeled performance is closer to simulated performance than it is to measured performance.

Upper-Bound Model In Chapter 3, the cores used for validation reflect state-of-the-art components available at the time that the work was completed. For S-Core validation, we use two Intel cores, a Xeon E5520 and a Core i7 860. For M-Core validation, we use GPGPUSim instead of an actual GPU. Since scaling effects are a key component of the

[6]Except for M-Cores using the upper-bound model, where measurements are taken using GPGPUsim

validation in this chapter, we use the simulator so that we can configure it to model a pre-Fermi style architecture with between 1 and 120 SM cores.

Custom and Translation Models In Chapters 4 and 5, we use a larger set of cores to show both that the model is valid and that it is scalable. For S-Cores, we use an ARM Cortex-A8, an Intel Atom, an Intel Xeon, and a Sandybridge core. For M-Cores, we use both pre-Fermi and Kepler architectures with a range of off-chip bandwidths and numbers of SMs. The complete details of cores used are in Table 2.4a for S-Cores and Table 2.4b for M-Cores.

2.4 Application to Dark Silicon Projections

As an example use-case for all three models, we apply each of them to a multicore model framework with area and power constraints and technology scaling information to find the speedup, number of cores, and type of cores that could be used for future multicores, as suggested by Figure 5.1. These results include dark silicon projections, the fraction of a multicore chip's area that must go unused due to power constraints. This running example addresses the question of how processor performance will scale in future technology generations given area and power constraints. We will apply techniques from previous work [28] in combination with our core models. In that work, technology scaling projections, single-core design scaling, multicore design choices, actual application behavior, and microarchitectural features are all combined to predict the speedup, number of cores, and percentage of dark silicon for multicores at future technology nodes. In this section, we review the parts of that approach necessary to use our core models to make these dark silicon projections.

In this dissertation, the focus is on techniques to predict single-core performance; applications within this framework are a key use-case and are a running example throughout.

2.4.1 Overview

To provide context for these examples, this section gives an overview of the general approach and specific implementation details where appropriate. Below, we summarize each of the steps previously developed to project performance and fraction of dark silicon at future technology generations:

Table 2.4: Hardware used for validation. Quadro FX580 is modeled with GPGPUSIM in Chapter 3.

	Cortex-A8 OMAP3340	Atom N450	Xeon E2550	Nehalem i7-965	Sandybridge i7-2600
Execution Model	In-Order	In-Order	Out-of-Order	Out-of-Order	Out-of-Order
Issue Width	2	2	4(6)	4(6)	4(6)
Freq (Ghz)	0.6	1.66	2.27	3.2	3.40
L1 Cache/core	16KB	32KB	64KB	64KB	64KB
L2 Cache/core	256KB	512KB	256KB	256KB	256KB
L3 Cache/chip	—	—	8MB	8MB	8MB
BW (GB/s)	—	5.3	26	21	26
SIMD Width	4	4	4	4	8
TDP (1 core, 45nm)	0.8W	2.8W	16.8W	30W	38W
Area (1 core, 45nm)	$7.5mm^2$	$8.1mm^2$	$18.4mm^2$	$28.7mm^2$	$29.1mm^2$
SPECmark	1.19	6	26	36.5	47.1
Validation Chapter	4/5	4/5	3/4/5	3/4/5	4/5
Dark Silicon Chapter	4/5	3/4/5	3/4/5	3	4/5

(a) S-Cores

	Quadro FX 580	GeForce 8400GS	GeForce GT330	GeForce GTX480	GeForce GTX660 Ti
Arch	pre-Fermi	pre-Fermi	pre-Fermi	Fermi	Kepler
Compute Capability	1.1	1.2	1.2	2.0	3.0
SP/SM	8	8	8	32	192
# of SMs	4	1	12		7
Freq	1.12	1.24	1.34	0.71	0.98
Mem BW	25.6	4.8	25.3		144.2
TDP (1 SM, $45nm$)	—	—	2.4W	16.7W	35.7W
Area (1 SM, $45nm$)	—	—	$8.1mm^2$	$16.1mm^2$	$54.0mm^2$
Validation Chapter	3*/4/5	4/5	4/5	—	4/5
Dark Silicon Chapter	—	—	3/4/5	4/5	4/5

(b) M-cores

- **Device scaling** model summarizes area, power, and frequency changes at future technology nodes as multiplicative factors. The factors are for 45 nm to as small as 8 nm based on ITRS projections [55] and more conservative projections from Borkar [15].

- **Core scaling** model consists of power/performance (Watts per SPECmark) and area/per-

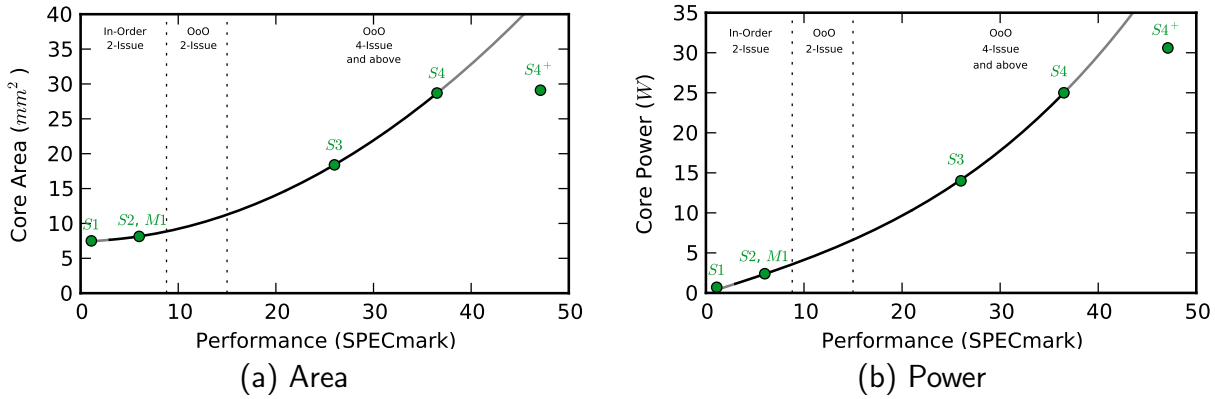


Figure 2.1: Area and Power Trade-offs.

formance (mm^2 per SPECmark) single core Pareto frontiers derived from a large set of diverse microprocessors with performance spanning from that of Intel Atom cores to that of Intel Nehalem cores. Note that all cores considered use 45 nm transistors, only L1 caches are included, and power is measured as reported thermal design power (TDP).

- **Multicore scaling model** is used to compute the area, power, and performance of any application for "any" chip topology for CPU-like and GPU-like multicore performance; this model was described earlier Section 2.2 of this chapter.

- **Device $\times$ core model** combines Pareto frontiers and the device model to find the Pareto frontiers at future technology nodes; any performance improvements for future cores will come only at the area and power costs defined by these curves.

- **Device $\times$ core $\times$ multicore model** with an exhaustive state-space search finds maximum multicore speedups for future technology nodes while enforcing area, power, and benchmark constraints.

2.4.2 Core Characteristics

In this section, we describe how we pick cores to include in dark silicon projections. In Chapter 3, we find core characteristics from the `Device` $\times$ `core` model. The required transition from SPECmark scores on a Pareto frontier to actual performance characteristics is a key

contribution of the upper-bound model in Chapter 3. However, for Chapters 4 and 5, the emphasis is on the actual core models and interpolating necessary hardware characteristic inputs for points on the curve is impractical. We therefore must pick several representative S-Core and M-Core points to use in the state space search for performance and dark silicon. We discuss how we pick those representative cores below.

Representative points: To pick the representative points, we first divide the Pareto frontiers from our previous work [28] into regions: In-order 2-issue, Out-of-Order 2-issue, and Out-of-Order 4-issue. These regions are based on the optimal energy-performance trade-offs across macro-architectures from previous work [5]. To divide the curve into these regions, we scale the performance from that work such that their cross-over points (e.g., from an Out-of-Order 2-Issue core to a 4-Issue core) occur at points that fit with our core SPECmark scores. After scaling the region transition points using the same factor, we find the regions shown in Figure 2.1.

Extended frontiers: Since Esmaeilzadeh et al. [28], the use of even simpler S-Cores in multicore chips has increased, and microarchitectural advances have pushed performance higher with small power and area impacts. To reflect these changes, we add two new points: the next microarchitectural generation of an Intel Desktop S-Core (a Sandybridge i7-2600 CPU) and an even simpler in-order, 2-issue mobile S-Core (an ARM Cortex-A8). Neither of these cores are available at the $45nm$ technology node, so we use scaling parameters derived from SPEC numbers to find their area and power at $45nm$. Data from [13] is used to find the A8 point ($S1$). We observe that the Atom ($S2$) has 5.5x higher performance than the A8 with 3.6x higher power. We add the Sandybridge core ($S4^+$) using reported SPECmark scores and TDP, and area from die photos.

After adding the core points, we extend the frontiers to include those points. We extend the trade-off curves slightly to the lower left to include the Cortex-A8 by adding power/area/performance trade-offs using the slope at the Atom point. We also extend the curve to the upper-right to include the performance level of the Sandybridge. The resulting curves, with extrapolated frontiers in gray, are shown in Figure 2.1. Since the curves previously passed through the Nehalem core ($S4$) and Sandybridge includes microarchitectural improvements, the Sandybridge is to the right of the Pareto frontiers. This motivates our search for a way to move beyond Pareto frontiers to predict core performance.

S-Cores: We pick five real cores to use as S-Cores. The points chosen loosely correspond to an ARM A8 ($S1$), an Intel Atom ($S2$), an Intel Xeon using the Nehalem microarchitecture ($S3$), an Intel i7 using the Nehalem microarchitecture ($S4$), and an Intel i7 using the Sandybridge microarchitecture ($S4^+$) based on known area/power/performance trade-offs. Although some of these cores fall on the Pareto frontiers, it is important to note that they may not be Pareto optimal; they fall on the curves only because the curves were constructed from data points that include these cores. Power, area, SPECmark scores, and microarchitectural characteristics are listed in Table 2.4a.

M-Cores: For the M-Cores, we cannot usefully map M-Core performance to a SPECmark score, and so instead focus on spanning a similar space to that spanned by the S-Cores. We find the area, power, and performance using reported values and die photos for the three common Nvidia M-Core architectures, as detailed below.

We first consider the pre-Fermi M-Core architecture. From Atom and pre-Fermi die photos inspections, we estimate that a pre-Fermi SM (8 SP cores), its caches, and thread register file can fit in the same area as one Atom processor [28]. We assume a similar correspondence with power, although this may be an overestimate.

For the $M2$ and $M3$ cores, we estimate area from die photos and power from Nvidia reported TDP [86, 88]. We use scaling projections from previous work [28] where necessary normalize to $45nm$. The three major architecture revisions currently available, pre-Fermi, Fermi, and Kepler, span a large architectural design space: the Fermi has $4\times$ more SP cores per SM than the pre-Fermi, and the Kepler has $24\times$ more SPs per SM than the pre-Fermi. Kepler SMs also has more advanced issue units and other features that are expected to improve performance over pre-Fermi SMs (note that the A8's scaled SPECmark score is about $33\times$ that of the Nehalem). The area and power impact is smaller: a Kepler SM uses only $3.1\times$ more area and $7.1\times$ more power than a pre-Fermi SM (compared to $3.8\times$ more area and $25\times$ more power for the Nehalem than an A8). By picking these three SMs as our M-Core design points, we have picked representative M-Core points that cover a design space similar to that of the S-Cores. The cores' area, power, SPECmark and microarchitectural characteristics are listed in Table 2.4b.

2.4.3 Projection Implications

We briefly consider the implications of the different design space assumptions in each chapter. As a result of differing core and benchmark design spaces, we do not expect to find the same results in Chapters 4 and 5 as in Chapter 3 and previously published work. As a reference, we compute dark silicon using the new assumptions and the upper-bound model. We expect those results to be more optimistic than the results obtained using the custom S-Core and M-Core models.

Additional implementation details and results can be found in the original publication [28]. The multicore model is explained in more abstract terms in a later work [11]. Expanded sensitivity studies and discussions of implications are included in several later pieces [31, 29, 30].

2.5 Summary

In this chapter, we provided an overview of the framework on which the core models are built and evaluated: the design space under consideration, the multicore model, the benchmarks, measurements, and hardware used for evaluation, and the framework for a running example, the application of the models to dark silicon projections. This content was summarized in two tables that were featured at the beginning of the chapter: Table 2.1 covered the multicore assumptions discussed in the first half of the chapter, and Table 2.2 summarized the infrastructural decisions discussed in the second half of the chapter.

In the next three chapters, we will describe each of the three models summarized in Chapter 1. Throughout those descriptions, the interested reader should refer back to this chapter for clarifications on terminology and methodology.

3 AN UPPER-BOUND GENERAL MODEL

This chapter describes the general upper-bound core model that we use for abstract performance projections. This is the most abstract model that we consider in this dissertation. The upper-bound model abstracts away components of the mechanistic model until converging on a single model that can be used for both S- and M- cores. This work is also described in another work [11].

Goals and Constraints: Our goal is to move from a simple resource trade-off curve to a per benchmark speedup prediction using only a few real program characteristics.

Inputs: The model assumes only a few inputs: the resource trade-off function, some knowledge of the architecture (S-Cores or M-Cores, processor frequency at the end-points of the resource trade-off function), and only five workload-specific inputs. Those workload-specific inputs can be the same for both S- and M-cores.

Use-case: We can use this model to predict optimistic results about the speedups using only minimal information about the cores; this makes it a very useful model for abstract performance projections where we have minimal information about specific processors.

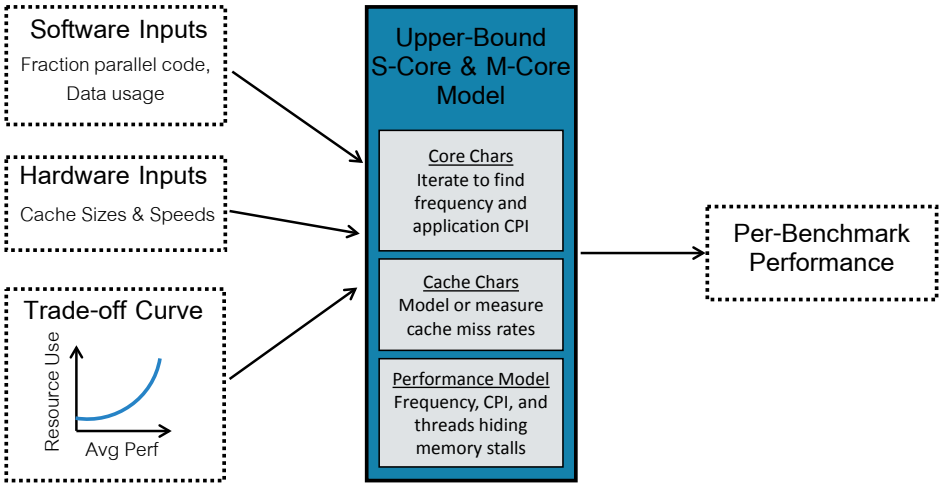


Figure 3.1: Upper-bound model overview.

The abstraction does bring with it a severe limitation: the microarchitecture is described with a single performance score, so specific changes to the microarchitecture cannot be accurately modeled. The coarse-grain architectural flexibility limits more fine-grained architectural flexibility, making this approach most useful for upper-bound scalability studies.

The upper-bound model approach was specifically developed to use in conjunction with resource trade-off curves. Examples of trade-off curves include the power-performance Pareto frontiers [28] and the use of Pollack's Rule [84] by Hill and Marty [49]; Pollack's Rule models the trade-off between core resources (r) and core performance as $perf(r) = \sqrt{r}$. Performance of symmetric, asymmetric, dynamic, and fused multicore topologies is then a function of r, $perf(r)$, and the fraction of code that is parallelizable. The upper bound core model in this dissertation tightens their upper-bound on multicore performance using real core measurements in place of the $perf(r)$ function and considers the impact of additional architectural and application characteristics.

The remainder of this chapter describes the first order core model using architectural and application input data in the context of the multicore model from Chapter 2; Figure 3.1 shows the specific implementation used in this chapter of the general multicore model from Figure 1.1 in Chapter 1. The goal is to improve multicore performance projection accuracy while maintaining simplicity and generality.

3.1 Model Description

In Chapter 2, we described multicore performance in terms of serial performance (P_1) and parallel performance (P_∞). Then, we broke each of those components into limits due to core performance (P_C) and chip memory bandwidth (P_B). This chapter is focused on finding P_C, but for completeness and to show an application of the descriptions in Chapter 2, we summarize the use of the other multicore components here. In Chapters 4 and 5, we will only focus on core models (P_C).

Below and in Figure 3.1, we summarize the model components' intuition, simple inputs, and useful outputs.

Core Performance: This component finds the expected performance of a single core. The impact of core microarchitecture, S-Core or M-Core organization, and memory access latency are all incorporated here. The inputs include organization, CPI, frequency, cache hierarchy, and cache miss rates. Because our cores are initially described only as a performance score,

we develop an approach to find CPI and frequency using that score relative to the scores of the cores that are on the trade-off function's end points.

Memory Bandwidth Performance: This component finds the maximum performance given memory bandwidth constraints by finding the number of memory transactions per instruction. The inputs are the trade-off function design point, application memory use, and maximum memory bandwidth.

Chip and Topology Constraints: This component finds the number of each type of core to include on the chip, given the multicore topology, organization, trade-off function, and resource constraints. Four multicore topologies are described in detail in Table 2.1. Each core is described as a combination of the trade-off function design point and thread style.

Multicore Performance: This component finds the expected serial and parallel performance for an application on a multicore chip with a particular topology. Inputs include outputs from the previous three models: core performance, number of cores of each type, and the memory bandwidth performance. We assume the first order bottlenecks for chip performance are core performance and memory bandwidth, and compute the total serial and parallel performance using these constraints.

Multicore Speedup: This component accumulates the results with the fractions of parallel and serial code into a single speedup projection using an Amdahl's Law based approach.

3.1.1 Design Space

The range of low to high performing cores is represented using simple performance/resource constraint trade-offs (the *trade-off function*). These model inputs can be derived from a diverse set of input measures for performance (e.g., SPECmark, CoreMark) and resource constraints (e.g., area, power). In our results, we assume two trade-off functions: Pareto-optimal frontiers with area-performance trade-offs and power-performance trade-offs. They are then used to predict performance in a diverse set of applications with only a few additional application-specific parameters. The trade-off function may be concrete points from measured data, a function based on curve fitting to known designs, or an abstract function of expected trade-offs. Examples of the input trade-offs using curve fitting are Pareto frontiers like those presented in Figure 2.1, previous work [4, 28], or even more abstract trade-off functions like Pollack's Rule [84].

Table 3.1: Model Input Parameters. Recall that L refers to lightweight cores, H refers to heavyweight cores, S- refors to S-Cores, and M refers to M-Cores.

Software Input	Description
CPI_{exe}	Cycles per inst (zero-latency cache) from comp model
f_{∞}	Fraction of code that can be parallelized
r_{ls}	Fraction of instructions that are loads or stores
m_{L1}	L1 cache miss rate (from cache model or measured)
m_{L2}	L2 cache miss rate (from cache model or measured)
Core Level Input	**Description**
$q_{d,th}$	Core performance (e.g., SPECmark, d: L/H, th: S/M)
$R(q_{d,th})$	Core resource cost (area, power) from trade-off func
ν	Core frequency (MHz) from computation model
T	Number of threads per core (CPU or GPU style)
t_{L1}	L1 cache access time (cycles)
t_{L2}	L2 cache access time (cycles)
t_{mem}	Memory access time (cycles)
Chip Level Input	**Description**
BW_{max}	Maximum memory bandwidth (GB/s)
b	Bytes per memory access (B)

3.1.2 Model Details

This section describes in detail the model's five components. Building on previous work [42] for an idealized symmetric multicore with a perfectly parallelized workload (Equations 1-4 below) and heterogeneous multicore extensions [49] to Amdahl's Law using abstract inputs, we use real core and multicore parameters and add additional hardware details to find a tighter upper-bound on multicore performance. The simple inputs listed in Table 3.1 are easily measured, derived from known results, or even based on intuition.

Each component is described by first outlining how it is modeled and then discussing any derived model inputs. This approach highlights our novel incorporation of real application behavior and realistic microarchitectural features.

Core Performance

Single core performance ($P_C(q_{d,th}, T)$) is calculated in terms of instructions per second in Equation 3.1 by scaling the core utilization (η) by the ratio of the processor frequency to CPI_{exe}:

$$P_C(q_{d,th}, T) = \eta \times \nu / CPI_{exe} \tag{3.1}$$

The CPI_{exe} parameter does not include stalls due to cache accesses, which are considered separately in the core utilization (η). The core utilization is the fraction of time that threads running on the core can keep it busy. The maximum number of threads per core is a key component of the core style: CPU-like cores are single-threaded (S) and GPU-like cores are many threaded (e.g., 1024 threads per 8 cores). Topologies may have a mix of heavyweight ($d = H$) and lightweight ($d = L$) cores. Core utilization is modeled as a function of the average time (cycles) spent waiting for each load or store (t), fraction of instructions that are loads or stores (r_{ls}), and the CPI_{exe}:

$$\eta = \min\left(1, \frac{T}{1 + t \times r_{ls}/CPI_{exe}}\right) \tag{3.2}$$

We assume two levels of cache and an off-chip memory that contains all application data. The average time spent waiting for loads and stores is a function of the time to access the caches (t_{L1} and t_{L2}), time to visit memory (t_{mem}), and the predicted cache miss rate (m_{L1} and m_{L2}):

$$t = (1 - m_{L1})t_{L1} + m_{L1}(1 - m_{L2})t_{L2} + m_{L1}m_{L2}t_{mem} \tag{3.3}$$

Although a cache miss model could be inserted here (e.g., [56, 28]), we assume the miss rates are known here. Changes in miss rates due to increasing number of threads per cache should be reflected in the inputs. We assume the number of cycles per cache access is constant as the frequency changes, while memory latency (in cycles) increases linearly with frequency increases. Note that Fermi style GPUs could be modeled using this cache latency model and cache miss rate inputs.

Derived Inputs

To incorporate known single-threaded performance/resource trade-offs into the model, we convert single-threaded performance into an estimated core frequency and per-application CPI_{exe}. This novel approach uses generic single-threaded performance results (e.g. SPECmark scores) to derive parameters for specific multi-threaded applications. This works because, assuming comparable memory systems, the key performance bottlenecks are the processor frequency (ν) and microarchitecture (summarized by CPI_{exe}). The approach finds the

```
for each point k on Trade-off Function do
    ν_k=(ν_max − ν_min)/nPoints × k + ν_min;
    CPI_exe,k=l;
    speedup_goal=(SPEC_k/SPEC_max);
    compute perf for point max, P_max;
    while speedup_goal and speedup_achieved not within threshold do
        compute perf for point k, P_k;
        speedup_achieved=(P_k/P_max);
        if speedup_achieved < speedup_goal then
            try a smaller CPI_exe,k;
        else
            try a larger CPI_exe,k;
        end
    end
end
```

Algorithm 3.1: Pseduocode to find CPI_{exe} and ν.

processor frequency and microarchitecture impact based only on the known single-threaded performance and the frequency of the highest and lowest performing processor, so it can be applied to abstract design points where only the performance/resource trade-offs are known. The approach is described below, and summarized in Listing 1.

ν: To find each core's frequency, we assume frequency scales linearly with performance, from the frequency of the lowest performing point to the that of the highest performing point.

CPI_{exe}: Each application's CPI_{exe} is dependent on its instruction mix and use of hardware optimizations (e.g., functional units and out-of-order processing). Since the measured CPI_{exe} for each benchmark is not available, the core model is used to generate per benchmark CPI_{exe} estimates for each design point. With all other model inputs kept constant, the approach iteratively searches for the CPI_{exe} at each processor design point. The initial assumption is that the highest performing core has a CPI_{exe} of ℓ. The smallest core should have a CPI_{exe} such that the ratio of its performance to the highest performing core's performance is the same as the ratio of their measured scores. Since the performance increase between any two points should be the same using either the measured performance or model, the same ratio relationship is used to estimate a per benchmark CPI_{exe} for each processor design point. This flexible approach uses the measured performance to generate reasonable performance model inputs at each design point.

Memory Bandwidth Performance

The maximum performance possible given off-chip bandwidth constraints ($P_B(q_{d,th})$, in instructions per second) is the ratio of the maximum bandwidth to the average number of bytes of data required per operation:

$$P_B(q_{d,th}) = \frac{BW_{max}}{b \times r_{ls} \times m_{L1} \times m_{L2}} \tag{3.4}$$

This is from Guz et al.'s model; previous bandwidth saturation work includes a learning model [93].

Limitations

The model's accuracy is limited by our optimistic assumptions, thus making our speedup projections over-predictions.

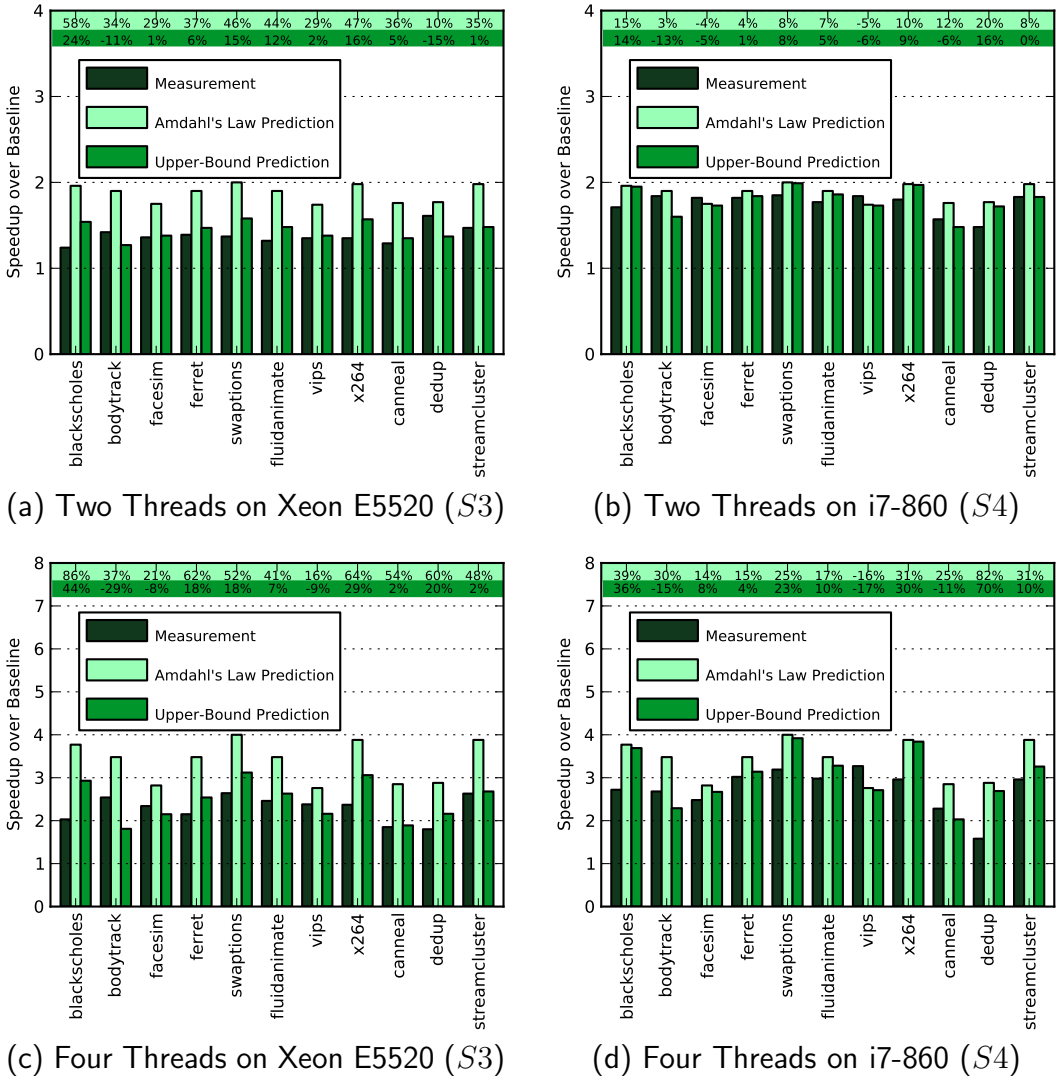
Memory Latency: A key assumption in the model is that memory accesses from a processor are in order and blocking. We observe that for high performance cores where CPI_{exe} approaches 0, performance is limited by memory latency due to the in order and blocking assumption:

$$\begin{aligned}\lim_{CPI_{exe}\to 0} P_C(q_{d,th}, T) &= \lim_{CPI_{exe}\to 0} \frac{\nu}{CPI_{exe}} \min\left(1, \frac{T}{1 + t\frac{r_{ls}}{CPI_{exe}}}\right) \\ &= \frac{T\nu}{t \times r_{ls}}\end{aligned}$$

This limitation is expected to have negligible impact when CPI_{exe} is greater than 1, but its impact increases as CPI_{exe} decreases, implying more superscalar functionality in the core.

Microarchitecture: We assume memory accesses from different cores do not cause stalls. Further, we assume that the interconnect has zero latency, shared memory protocols have no performance impact, threads never migrate, and thread swap time is negligible. The assumptions cause the model to over-predict performance, making projected speedups optimistic.

Application Behavior: The model optimistically assumes the workload is homogeneous, parallel code sections are infinitely parallelizable, and no thread synchronization, operating



(a) Two Threads on Xeon E5520 ($S3$)

(b) Two Threads on i7-860 ($S4$)

(c) Four Threads on Xeon E5520 ($S3$)

(d) Four Threads on i7-860 ($S4$)

Figure 3.2: S-Core Validation: Speedup over one i7-860 thread

system serialization, or swapping overheads occur.

3.2 Model Validation

To validate the model, we compare speedup projections from the model to measured and simulated speedups for CPU and GPU organizations. To validate the model's tighter bound, we also find the speedup projections using Amdahl's Law.

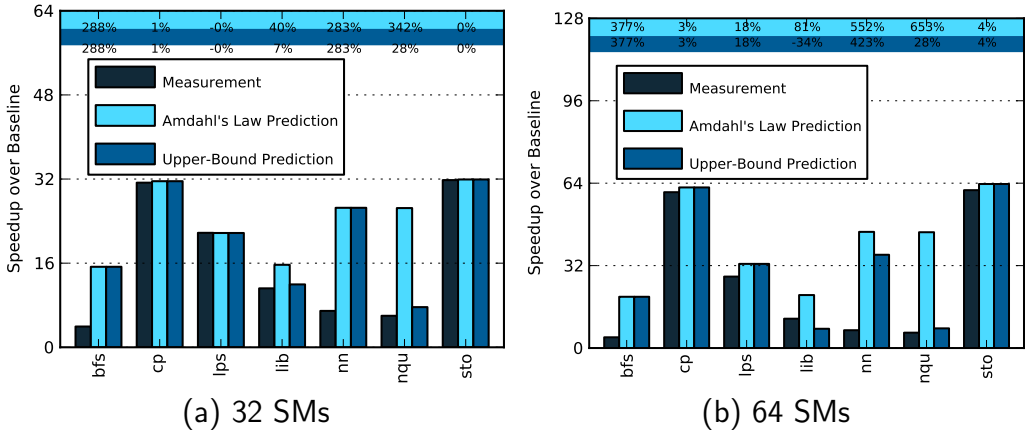


Figure 3.3: M-Core Validation: speedup over one SM

For S-cores, we compare the predicted speedup to measured speedups on 11 of the 13 PARSEC benchmarks using either 2 or 4 threads on two four-core chips, a Xeon E5520 and a Core i7 860, as shown in Figure 3.2. We assume $t_{L1} = 3$ cycles, $t_{L2} = 20$ cycles, and $t_{mem} = 200$ cycles at 3.2Ghz.

The model captures the impact of the differing microarchitectures and memory systems, is still optimistic, and provides a tighter bound than the Amdahl's Law projections. The upper-bound core model's errors range from -29% to 70% with an average absolute error of 14%, while the Amdahl's Law approach has errors that range from -16% to 86% with an average absolute error of 31%. We use the average CPI_{exe} approach as described above using average SPECmark performance to guide our predicted CPI_{exe} vales. We thus expect that (1) per-benchmark performance predictions may not be optimistic, if the benchmark has computation or branch prediction behavior not represented by the average performance characteristics, and (2) the average speedup over the set of PARSEC benchmarks would be optimistic and more accurate than individual per-benchmark results. Indeed, we find that the average speedup predicted is within 10% of the measured average speedup.

We validate M-core projections comparing speedups generated using the model to those simulated using GPGPUSim (version 3, PTX mode) [7]. We use the number of threads and blocks generated by the CUDA code to deal with occupancy issues beyond the warp level, and estimate f from the speedup between 1 and 32 SMs (assuming perfect memory). f therefore includes the effect of blocking and other serializing behavior. For the M-Core

model, we use $t_{L1} = 1$ and let t_{mem} be the average memory latency over the entire kernel as reported by GPGPUSim for a 32 SM GPU (varies from 492-608 cycles).

Figure 3.3 shows results for 7 of GPGPUSim's 12 CUDA benchmarks. The three specific benchmarks demonstrate three cases: the highly parallel StoreGPU is well predicted by both Amdahl's Law and the upper-bound core model, N-Queens Solver's limited number of threads is significantly better predicted using the upper-bound core model, and Neural Network's memory bandwidth saturation is poorly predicted by both models; these results are shown in more detail in Appendix C.

The geometric mean (C.1d) shows that the model maintains a tighter upper bound on speedup projections than Amdahl's Law projections. However, per-benchmark errors are still large: they range from 0% to 423%. The average absolute error is 106% for the upper-bound core model versus 189% for the Amdahl's Law approach, showing the improved predictions.

Note that upper-bound model approach is philosophically different than Hong and Kim's approach of using model inputs based on the CUDA program implementation [50], which will be discussed in more detail in Chapter 4.

3.3 Application to Dark Silicon Projections

Throughout this dissertation, we use the models described in each chapter to produce dark silicon projections and associated speedups for future technology nodes. The approach for these projections was described in Chapter 2. Additional implementation details and results using the upper-bound core model are presented elsewhere [30].

As an example, Figure 3.4 shows results found when applying this approach to a set of performance trade-off curves for power and area and searching for the configuration with the best speedup at a series of technology nodes. Figure 3.4 summarizes all of the speedup projections in a single scatter plot for conservative and ITRS scaling models. For every benchmark at each technology node, we plot the speedup of eight possible multicore configurations (CPU-like, GPU-like) × (symmetric, asymmetric, dynamic, composed). The upper curve indicates *performance* Moore's Law or doubling performance for every technology generation.

- With optimal multicore configurations for each individual application, at 8 nm, only 3.7× (conservative scaling), or 7.9× (ITRS scaling) geometric mean speedup is possible.

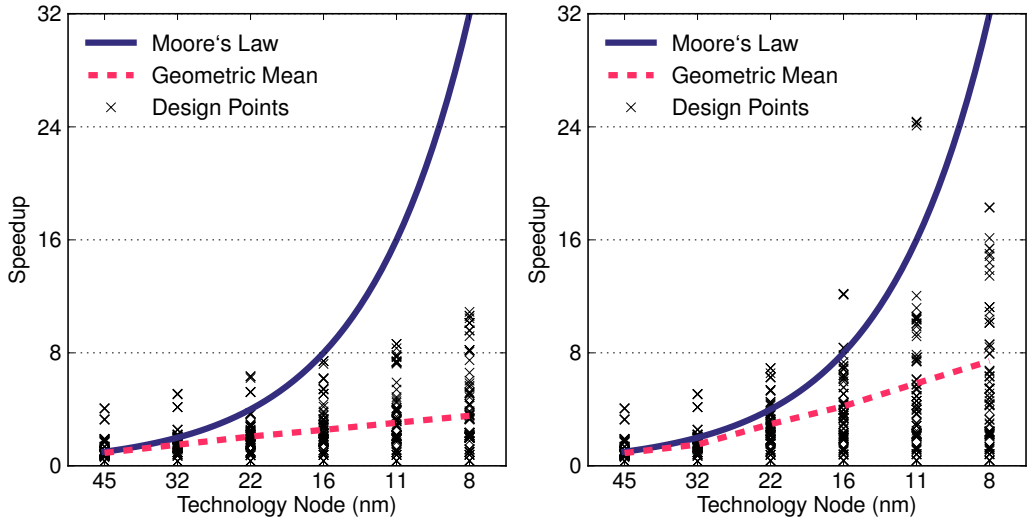
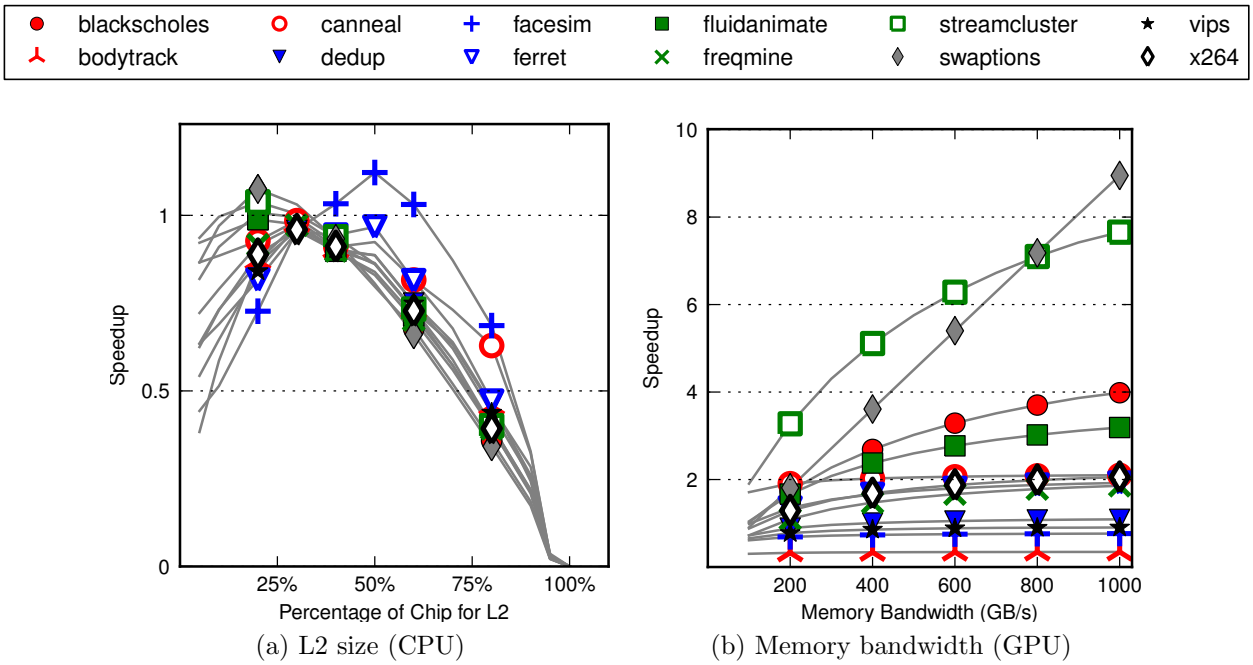


Figure 3.4: Speedup across process technology nodes across all organizations and topologies with PARSEC benchmarks.

- Highly parallel workloads with a degree of parallelism higher than 99% will continue to benefit from multicore scaling.
- At 8 nm, the geometric mean speedup for heterogeneous dynamic and composed topologies is only 10% higher than the geometric mean speedup for symmetric topologies.
- Improvements in transistor process technology are directly reflected as multicore speedup; however, to bridge the dark silicon speedup gap even a disruptive breakthrough that matches our aggressive scaling model is not enough.

Our analysis above examined "typical" configurations and showed poor scalability for the multicore approach. A natural question is, *can simple configuration changes (percentage cache area, memory bandwidth, etc.) provide significant benefits?* We elaborate on three representative studies of simple changes (L2 cache size, memory bandwidth, and SMT) below. Further, to understand whether parallelism or the power budget is the primary source of the dark silicon speedup gap, we vary each of these factors in two experiments at 8 nm. The upper-bound model is flexible enough to perform these types of studies.



(a) L2 size (CPU) (b) Memory bandwidth (GPU)

Figure 3.5: Impact of L2 size and memory bandwidth on speedup at 45 nm.

L2 cache area. Figure 3.5(a) shows the optimal speedup at 45 nm as the amount of a symmetric CPU's chip area devoted to L2 cache varies from 0% to 100%. In this study we ignore any increase in L2 cache power or increase in L2 cache access latency. Across the PARSEC benchmarks, the optimal percentage of chip devoted to cache varies from 20% to 50% depending on benchmark memory access characteristics. Compared to a 30% cache area, using optimal cache area only improves performance by at most 20% across all benchmarks.

Memory bandwidth. Figure 3.5(b) illustrates the sensitivity of PARSEC performance to the available memory bandwidth for symmetric GPU multicores at 45 nm. As the memory bandwidth increases, the speedup improves as the bandwidth can keep more threads fed with data; however, the increases are limited by power and/or parallelism and in 10 out of 12 benchmarks speedups do not increase by more than 2× compared to the baseline, 200 GB/s.

SMT. To simplify the discussion, we did not consider SMT support for the processors (cores) in the CPU multicore organization. However, SMT support can improve the power

efficiency of the cores for parallel workloads to some extent. We studied 2-way, 4-way, and 8-way SMT with no area or energy penalty, and observed that speedup improves with 2-way SMT by 1.5× in the best case and decreases as much as 0.6× in the worst case due to increased cache contention; the range for 8-way SMT is 0.3-2.5×.

3.4 Related Work

Hill and Marty [49] extend Amdahl's Law to model multicore speedup with symmetric, asymmetric, and dynamic topologies and conclude dynamic multicores are superior [49]. Several extensions to Hill and Marty [49] model have been developed for modeling 'uncore' components (e.g. interconnection network and last level cache), [70], computing core configuration optimal for energy [66, 22], and leakage power [102]. All these model uses area as the primary constraint and model single-core area/performance trade-off using Pollack's rule (Performance $\propto \sqrt{Area}$ [84]) without considering technology trends.

Azizi et al. [4] derive the single-core energy/performance trade-off as Pareto frontiers using architecture-level statistical models combined with circuit-level energy-performance trade-off functions. For modeling single-core power/performance and area/performance trade-offs, our core model derives two separate Pareto frontiers from empirical data. Further, we project these trade-off functions to the future technology nodes using our device model. Esmaeilzadeh et al. [32] perform a power/energy Pareto efficiency analysis at 45 nm using total chip power measurements in the context of a retrospective workload and microarchitecture analysis. In contrast to the total chip power measurements for specific workloads, we use the power and area budget allocated to a single-core to derive the Pareto frontiers and combine those with our device and chip-level models to study the future of multicore design and the implications of technology scaling.

Chakraborty [19] considers device-scaling and estimates a simultaneous activity factor for technology nodes down to 32 nm. Hempstead et al. [47] introduce a variant of Amdahl's Law to estimate the amount of specialization required to maintain 1.5× performance growth per year, assuming completely parallelizable code. Chung et al. [23] study unconventional cores including custom logic, FPGAs, or GPUs in heterogeneous single-chip design. They rely on Pollack's rule for the area/performance and power/performance trade-offs. Using ITRS projections, they report on the potential for unconventional cores considering parallel kernels. Hardavellas et al. [46] forecast the limits of multicore scaling and the emergence of

dark silicon in servers with workloads that have an inherent abundance of parallelism. Using ITRS projections, Venkatesh et al. [98] estimate technology-imposed utilization limits and motivate energy-efficient and application-specific core designs.

Previous work largely abstracts away processor organization and application details. This study provides a comprehensive model that considers the implications of process technology scaling, decouples power/area constraints, uses real measurements to model single-core design trade-offs, and exhaustively considers multicore organizations, microarchitectural features, and real applications and their behavior.

3.5 Summary and Open Questions

Our first order multicore model projects a tighter upper bound on performance than previous Amdahl's Law based approaches. This extended model incorporates abstracted single threaded core designs, resource constraints, target application parameters, desired architectural features, and additional first order effects—exposing more bottlenecks than previous versions of the model—while remaining simple and flexible enough to be adapted for many applications.

The simple performance/resource constraint trade-offs that are inputs to the model can be derived and used in a diverse set of applications. Examples of the input trade-off functions may be Pareto frontiers like those presented in Azizi et al. [5] and Esmaeilzadeh et al. [28], known designs that an architect is deciding between, or even more abstract trade-off functions like Pollack's Rule [84] used in Hill and Marty [49]. The proposed speedup calculation technique can be applied to even more detailed parallelism models, such as Eyerman and Eeckhout's critical sections model [33]. This complete model can complement simulation based studies and facilitate rapid design space exploration.

The upper-bound model's flexibility is also the source of its most significant limitations. The microarchitecture is described with a single performance score, so most changes to the microarchiteture cannot be accurately modeled. The coarse-grain architectural flexibility limits more fine-grained architectural flexibility. Further, per-benchmark results have limited accuracy since they are based on so few inputs. Performance limiters like instruction level parallelism, long-latency instructions, and resource contention for execution units are not modeled, leading to optimistic projections. This leads to our next model, modular microarchitecture extensions to predict performance for real cores.

4 MODULAR MICROARCHITECTURE EXTENSIONS

This chapter describes modular microarchitecture extensions to previously published *S*- and *M*- models.

Goals and Constraints: The immediate goal of this work is to use previous performance models to accurately predict performance on real hardware; a more far-reaching goal is to make it easier for architects to modify custom models to accurately explore the impact of proposed microarchitecture changes; and the ultimate goal is to provide a basis for the translation-based architecture extensions in Chapter 5.

In the previous chapter, we described upper-bound performance models that used a known performance on a representative workload (e.g., SPECmark) along with a few simple inputs to predict performance for multi-core topologies. Because of that model's approach, per benchmark speedup predictions had errors up to 70% for *S*-Cores and higher for *M*-Cores, but trends over a set of benchmarks tended to be more accurate. In this chapter, we consider the case where we can run the workload of interest on hardware, but are interested in predicting performance on that workload using an analytic model[1].

Inputs: Our assumptions in this chapter include that we have real hardware on which we can take measurements using performance counters and either binary instrumentation or an emulator, any changes to the hardware do not lead to changes in memory behavior (or we can accurately predict those changes), we have binaries compiled for the architecture under study, and that we are not using a simulator for any measurements. Relying on hardware and either binary instrumentation or emulation limits the types of measurements that we can make, but also increases tool portability and makes data collection very fast.

Use-case: The use-cases for this model include predicting performance when changes are made to the microarchitecture (e.g., issue width or instruction latency); the predicted performance can be used either to investigate the performance impact of a specific change or for design space exploration. Applying the example to our running dark silicon projections example, the detailed mechanistic models in this chapter can be used to add projections for

[1]See the Use-case paragraph below for the utility of such a model.

new cores that are outside of the Pareto-frontier we used previously. In another application, these models are an essential component of our cross-architecture performance prediction in Chapter 5.

The chapter starts with model descriptions for both S- and M- cores developed by others. We then consider the models as a modular framework upon which we build modular extensions to capture additional features. We adapt the models to specific cores using subsets of those modular extensions, and validate the resulting models against measurements on real hardware. Finally, we use the models to generate multicore projections and explore the dark silicon projections from the previous chapter in more detail. The chapter concludes with related work and a discussion of open questions.

4.1 Background: Generic Models

We leverage two first order models that were developed independently: Eyerman et al.'s out of order S-Core model [57, 34] and Hong and Kim's M-Core model [50, 89]. Additional mechanistic models for both S-Cores and M-Cores may have slightly different assumptions, but the approaches are all similar. We discuss these other approaches in more detail in this chapter's related work, Section 4.7.

At the highest level, both models measure the amount of time required for computation, memory accesses, and control, and the amount of time from each of those categories that can be overlapped:

$$T = T_{computation} + T_{memory} + T_{control} - T_{overlap} \tag{4.1}$$

In each section below, we describe the computation for each of the components in Equation 4.1 for the S- and M- models as described in previous work. In Sections 4.2 and 4.3, we will describe extensions to these models and modifications required to use only real execution on real processors along with, if necessary, binary instrumentation (S-Cores) or emulation (M-Cores, since binary instrumentation is unavailable)[2].

[2]Previous S-Core models and some M-Core models assumed access to a simulator.

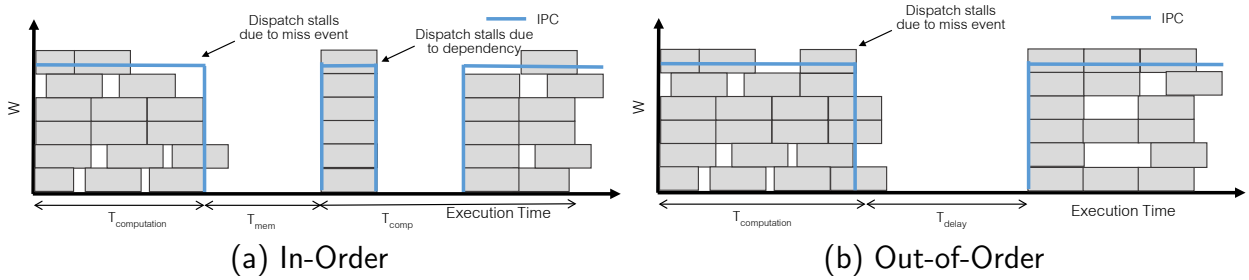


Figure 4.1: Basic Execution Models for S-Cores.

4.1.1 S-Core Performance Models

Mechanistic S-models include the $T_{computation}$, T_{memory}, and $T_{control}$ terms from equation 4.1:

$$T = T_{computation} + T_{memory} + T_{control} \tag{4.2}$$

For S-Core Models, the $T_{computation}$ term is a function of the number of instructions and how quickly they can be executed, assuming no pipeline stalls. The remaining two terms, T_{memory} and $T_{control}$, quantify the expected pipeline stalls due to events such as memory accesses and branch mispredictions. Note that there is no $T_{overlap}$ term; we assume a single-threaded S-Core and thus there are no threads to hide computation or memory latencies. The interval approach is shown in Figure 4.1 for both In-Order and Out-of-Order cores and notation used in this chapter is summarized in Table 4.1.

The memory cost is modeled as the number of cache miss events (of various types), the latency of each miss event, and the effect of the miss event on the issue rate. The cache miss events considered depend on the hardware; previous work includes L1 cache misses, L2 cache misses, ITLB misses, and DTLB misses [34] . The complete set of inputs used in the S-Core model are listed in Table 4.1.

The computation cost is modeled as the number of instructions, the amount of instruction level parallelism, and the average instruction latency. Instruction level parallelism and the average instruction latency together determine the IPC (independent of stalls, similar to IPC_{exe} in Chapter 3). Instruction level parallelism has slightly different derivations between In-Order cores and Out-of-Order cores, as detailed below.

In-Order: Basic in-order single threaded processor models have been proposed in multiple contexts [16, 38, 14, 27, 65]. Instruction stalls can be caused by dependencies, long latency instructions, and cache misses even at nearby first-level caches. To find the average time required for each instruction type, we find, for each instruction type, the average time required to maintain in order completion and time required for for branch mispredictions, in order pipeline, and dependency information [27, 38]:

$$T_{computation} = \frac{N_I}{D} + P_d \times N_d + P_L \times N_L \tag{4.3}$$

In the above equation, N_I is the number of instructions, D is the issue width, P_d is the average penalty for dependent instructions, N_d is the number of instructions that have dependent instructions following them, P_L is the average penalty for long latency instructions, and N_L is the number of long latency instructions.

Out-of-Order: For out-of-order cores, $T_{computation} = N/IPC$. Here, we find IPC by noting that $IPC = ILP/l_{avg}$, where IPC and ILP are measured without considering processor stalls due to memory, branch mispredictions, or other non-ideal behavior.

In previous work, instruction level parallelism is measured by simulating benchmarks with extra instrumentation that measures the length of each dependency chain (that fits in the instruction window) every cycle [57]. The average ILP is, then, the average dependency length divided by the processor's issue width. Rather than use a simulator, we measure ILP using a Pin based tool, MICA, which measures ILP by maintaining an instruction window and checking how many instructions are available to be issued each cycle. The two approaches are equivalent.

Next, we describe an M-Core model at a similar level of detail.

4.1.2 M-Core Performance Models

This section presents an overview of previous GPU models; refinements to the model are in the next two sections. We consider Hong and Kim's model and its extensions [50, 89].

At the top level, the Hong and Kim model combines time spent on execution (including

Table 4.1: Inputs to S-Model and Modules Where Used.

Software Input	Description	Module(s)
f_∞	Fraction of code that can be parallelized	—
N_I	Total number of instructions	Issued Insts.
N_M	Total number of μ-ops	Issued Insts.
n_i	Number of instructions of type i	Avg. Inst. Lat.
n_μ	Number of μ-ops of type μ	Avg. Inst. Lat.
ILP	Instruction level parallelism, given window size	ILP
μLP	μ-op level parallelism, given window size	ILP
N_{mL1}	Number of L1 cache misses	L2 Accesses
N_{mL2}	Number of L2 cache misses	L3 Accesses
N_{mL3}	Number of L3 cache misses	Mem Accesses
N_{mITLB}	Number of ITLB cache misses	DTLB Misses
N_{mDTLB}	Number of DTLB cache misses	ITLB Misses
N_{mL3_o}	Number of L3 cache misses that overlap	Mem Accesses
Core Level Input	**Description**	**Module(s)**
D	Issue width	ILP, T_{mem}
W	Instruction window size (= D, for in-order cores)	ILP
l_i	Latency of instruction of type i (cycles)	Avg. Inst. Lat.
p_i	Ports on which instruction may issue (0-5)	Resource Cont.
$M(i, \mu)$	Mapping from instructions to μ-ops	Issued Insts.
ν	Core frequency (MHz)	T_{total}
L_{dr}	Window drain time	$T_{control}$
L_{fe}	Front end pipeline length	$T_{control}$
t_{L1}	L1 cache access time (cycles)	Avg. Inst. Lat.
t_{L2}	L2 cache access time (cycles)	L2 Accesses
t_{L3}	L3 cache access time (cycles)	L3 Accesses
t_{mem}	Memory access time (cycles)	Mem. Accesses
t_{ITLB}	Latency of an ITLB miss (cycles)	ITLB Misses
t_{DTLB}	Latency of a DTLB miss (cycles)	DTLB Misses
Chip Level Input	**Description**	**Module(s)**
BW_{max}	Maximum memory bandwidth (GB/s)	—
b	Bytes per memory access (B)	—

control) and time spent on memory accesses, then subtracts overlap:

$$T = T_{computation} + T_{memory} - T_{overlap} \tag{4.4}$$

The exeuction pipeline that leads to these overlaps is shown in Figure 4.2.

The key idea of the M-Core models is to figure out how much work (computation and

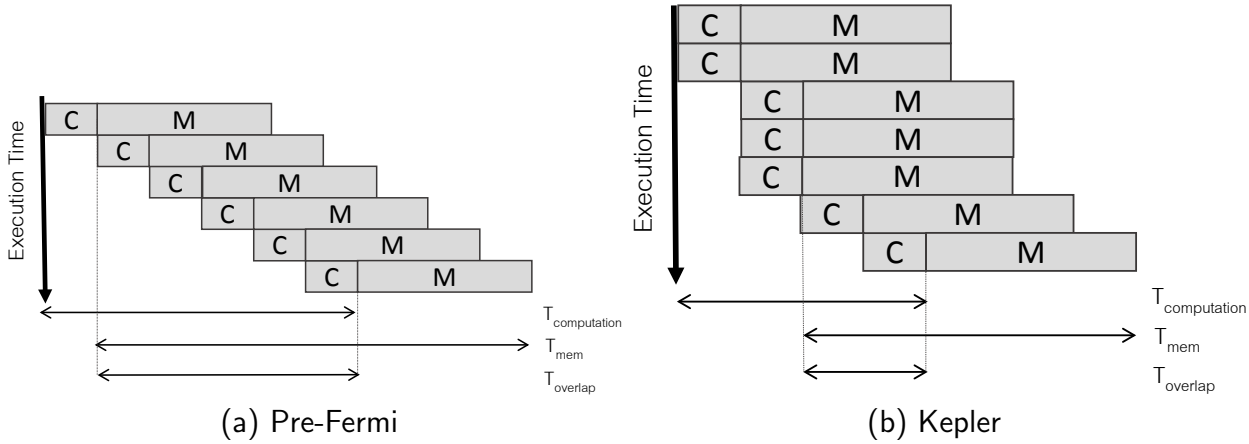


Figure 4.2: Basic Execution Models for M-Cores

memory) occurs per warp, how many warps are co-located on each SM, whether there are any chip-wide bottlenecks (e.g., memory bandwidth exceeded), and finally how much of the work overlaps. This approach is detailed in [50, 89].

4.2 Customizable Modular Mechanistic Models

In this section, we describe the new modules that we added to accurately predict performance on real hardware. We split the modules into S-Core and M-Core specific modules. Although some modules were initially developed for a particular architecture (e.g., the resource constraint module was initially developed to improve accuracy in Sandybridge performance projections), the modules are generic and can be applied to multiple architectures (e.g., the resource constraint module is also used for Nehalem architectures).

This section describes new modules without focusing on where they are used; in the next section, we describe core-specific models built by picking the correct custom modules to add to the model and include the validation of those models.

4.2.1 S-Core

When each architectural component's effect is considered independently as in Figure 4.3, it highlights the modular nature of the mechanistic model. Each effect is its own independent module, and new effects can be layered into the model by adding new modules. Below,

Table 4.2: M-Core Inputs and Modules Where Used.

Software Input	Description	Module(s)
W_{SM}	Number of warps per SM	T_{total}
N_I	Total number of instructions per warp	$T_{parallel}$,$T_{overlap}$
N_M	Total number of micro-ops per warp	$T_{parallel}$,$T_{overlap}$
N_{SFU}	Total number of micro-ops that execute in SFU per warp	SFU
N_{sync}	Total number of synchronization instructions per warp	Synchronization
N_{CF_div}	Extra instructions due to control flow divergence per warp	Branches
N_{con}	Memory bank conflict overheads per warp	Banks
n_i	Number of instructions of type i per warp	Avg. Lat.
n_μ	Number of μ-ops of type μ per warp	Avg. Lat.
$ILP(W)$	Instruction level parallelism per warp	IPC
N_{mem}	Number of Memory Accesses per warp	T_{mem},$T_{overlap}$
N_{off_chip}	Number of off-chip memory accesses per warp	T_{mem},$T_{overlap}$
avg_trans_warp	Average transactions per access	T_{mem},$T_{overlap}$
Core Level Input	**Description**	
D	Issue width	$T_{parallel}$
W	Instruction window size	$T_{parallel}$
l_i	Latency of instruction of type i (cycles)	Avg. Lat.
$M(i,\mu)$	Mapping from instructions to μ-ops	Issued Insts.
ν	SM frequency (MHz)	T_{total}
SM	Number of SMs	T_{total}
SP	Number of SPs per SM	$T_{parallel}$
t_{mem}	Memory access time (cycles)	T_{mem}
Δ	Transaction delta	T_{mem}
Chip Level Input	**Description**	**Module(s)**
BW_{max}	Maximum memory bandwidth (GB/s)	T_{mem}
b	Bytes per memory access (B)	T_{mem}

we explain a set of new modules and how they can be incorporated into the model. In Section 4.3, we will pick specific modules to model specific cores and show the accuracy of the resulting models.

Issued Instructions: A key input to the model is the number of instructions executed, along with their latencies and the amount of instruction level parallelism. Although the model is intrinsically agnostic to the ISA, the x86 ISA's use of micro-ops does introduce an additional layer of complexity. To properly model the processor, we need to consider the instructions as they are actually executed, in the micro-op form.

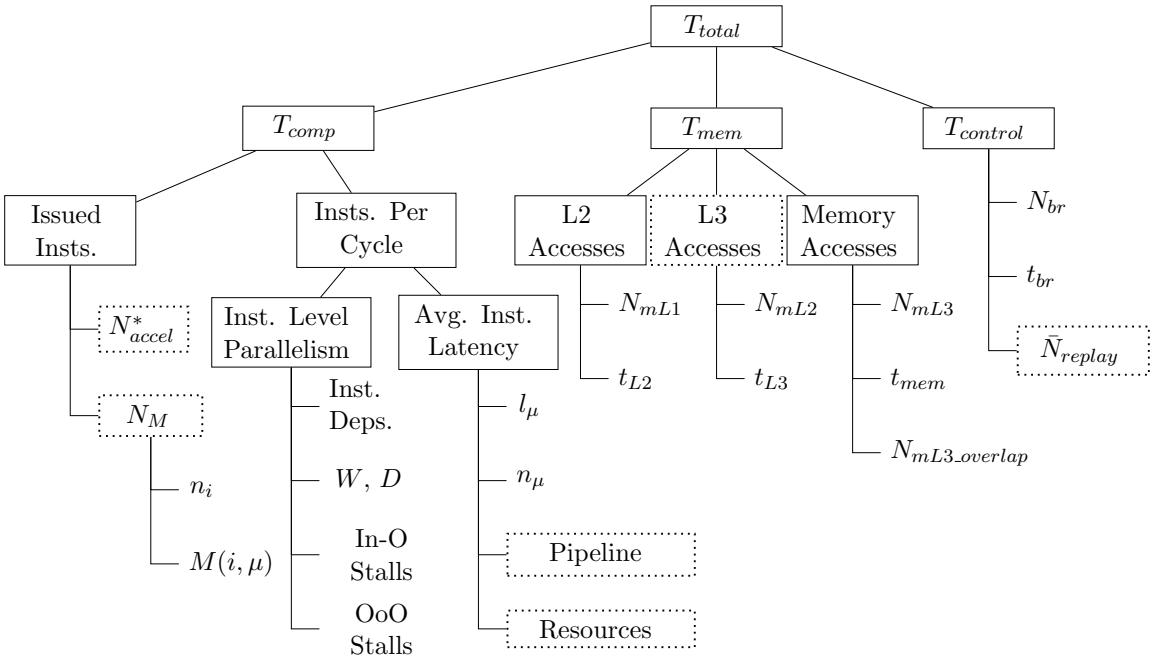


Figure 4.3: Hierarchical modules that effect performance for S-Cores. New modules have dotted borders. The starred N_{accel} module is discussed in Chapter 6.

Using x86 cores thus introduces new complexity: instructions from a trace do not include the micro-ops that are actually executed and each instruction must be parsed to see which instructions include memory operations. This is addressed by examining the operands for each instruction to find any memory operands. Further, we use a dictionary with known micro-op breakdowns (and latencies) for each instruction [36].

Since the model actually operates at the micro-op level, not the macro-op level, the impact on the ILP, which was measured at the macro-op level, must also be addressed. However, we observe that $ILP = \mu LP$: since each instruction is assumed to take one cycle in the ILP calculation and micro-ops are (generally) a dependent chain, the net effect on the ILP is zero:

$$\mu LP = \frac{N_M}{T_{comp,M}} = \frac{N_I * \text{avg\_M\_per\_I}}{T_{comp,I} * \text{avg\_M\_per\_I}} = ILP \tag{4.5}$$

L3 Accesses: Adding a layer of cache is relatively straight-forward. The key decision is whether the module inflates instruction latencies or whether it causes a pipeline stall. In this case, we assume that the addition of an L3 cache makes L2 cache misses just act like

a long instruction (instead of a stall). L3 cache misses lead to memory accesses, which we assume still cause a pipeline stall.

In-Order Issue Constraints: In-order, superscalar pipelines imply that two instructions that issue in the same cycle may not be dependent on each other. In previous work, the penalty due to this dependence requirement only included instructions stalled after the dependent instruction. We update this penalty to include the dependent instruction. For our 2-wide processors, this equation becomes:

$$P_d = N_{dep} \times \frac{D-1}{D} \tag{4.6}$$

Here, N_{dep} is the number of instructions that are dependent on the next instruction, D is the issue width, and $(D-1)/D$ is the probability that two instructions that have a dependence are in the same stage.

To find N_{dep}, we directly use a Pin-based tool, MICA, which includes an ILP measurement component [52]. Since we are only interested in dependency distances of one, we observe that we can use MICA's ILP calculation with the instruction window size set to 2 to get the global probability that an instruction is dependent on the instruction immediately after. The ILP tool assumes an out-of-order processor with single cycle instructions, but by setting the window size to two, we force it to only consider two instructions at a time. Since these two instructions can only issue at the same time if there is no dependence between them, we can use this to find the probability that each instruction has a dependency distance of one:

$$p_{distance=1} = ILP - 1 \tag{4.7}$$

Note that ILP is always greater than or equal to one given the assumptions in the MICA tool. We could get this information more directly, but this allows us to use the same off-the-shelf tools for both the In-Order and Out-of-Order models. Note that this approach will not work for wider issue widths, but we assume that issue widths greater than two will not be used for in-order processors.

Resource Constraints for Simple Pipelines: Each pipeline in a core may only be able to execute certain instructions. For relatively narrow pipelines (e.g., dual-issue), these

constraints are fairly simple to model. Below, I describe the approach for the Atom pipeline; the approach for the Cortex-A8 is similar.

We start by computing the fraction of instructions that can issue from only port 0, only port 1, or either port. Note that memory accesses, integer shift/shuffle/pack instructions, multiplies, divides, and other complex instructions only issue from port 0, but (some) SIMD instructions can issue from either port, as shown in Figure 4.4a.

Then, we distribute the instructions that can issue to either port across the two ports to get the best balance possible between the two pipelines:

$$f_{portX} = f_{regmov} + f_{INTadd} + f_{BOOL}$$
$$f_{port1} = f_{fpadd} + f_{branch} + f_{LEA}$$
$$f_{port0} = 1 - f_{portX} - f_{port1}$$

Then, we use a simple queuing model to find any overheads introduced by the unbalanced pipeline, as described below.

Resource Contention for Wider Pipelines: As shown in Figure 4.4b, the Sandybridge architecture includes six execution ports, but each port has only limited functionality (three execution ports with varying SIMD capabilities, two load ports, one store port). In most cases, this is not an issue: the distribution of ports tends to be sufficient given the ILP and instruction mixes for most programs. However, for some benchmarks (especially SIMD heavy benchmarks), the pipeline can be a constraint.

The resource contention module computes the impact of any contention for the individual pipelines. To do so, it first finds the probability than a micro-op that is ready to issue will be issued to a given port, p_μ. We assume a uniform distribution of instruction types. This is possible because we keep a dictionary with instructions as keys and the number of micro-ops are issued as a result of that instruction to each port (in some cases, the instruction may be able to issue to more than one port; in rare cases for less commonly used instructions, we do not have complete mapping information and make an informed guess). Since some instructions can issue to more than one port, we use an iterative approach to balance the instructions across ports as best as possible. This may lead to an optimistic distribution of instructions across ports. The approach is shown in Listing 2.

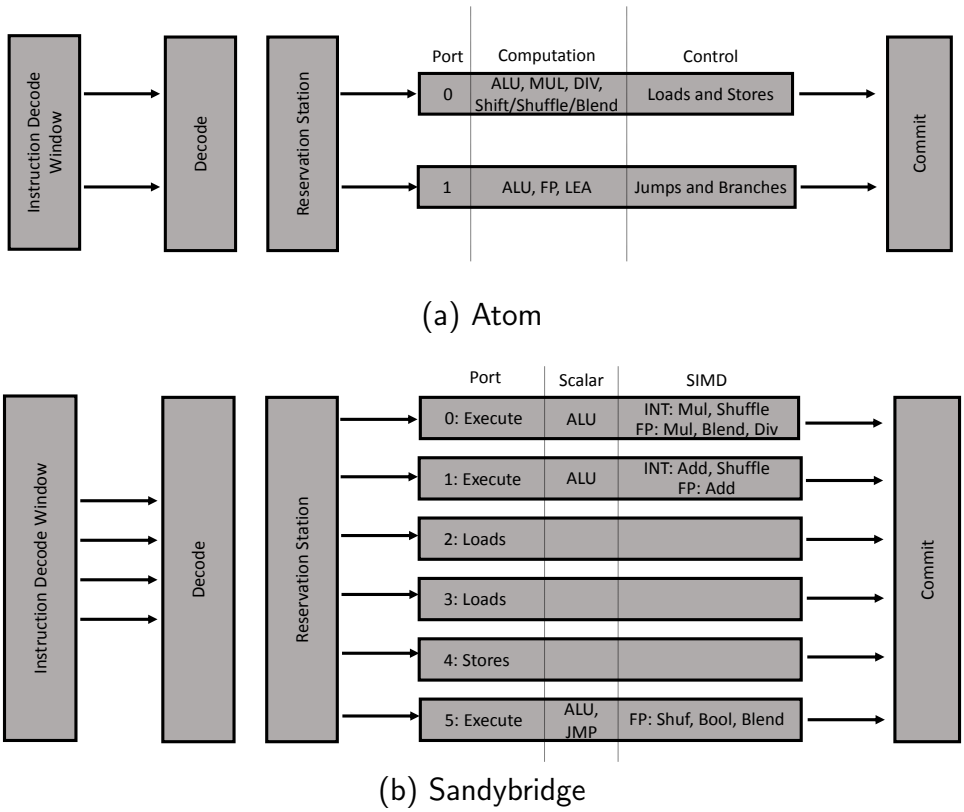


Figure 4.4: Pipeline restrictions that lead to resource contention.

Then, we find the average arrival rate for each execution port i, λ_i, as a function of the IPC_M, D (issue width), and the probability that a ready μ-op will be issued to that port, p_μ:

$$\lambda_i = \frac{min(D, IPC_M)}{D} * p_\mu \tag{4.8}$$

Then, observing that pipelined execution units mean that an instruction can be issued to a port every cycle (service time, ω, is one), we use basic M/M/1 queuing equations for each pipeline i to find the average wait time for each pipeline:

$$W_i = \frac{1}{\omega - \lambda} - \frac{1}{\omega} = \frac{1}{1 - \lambda} - 1 \tag{4.9}$$

```
for all instruction types, i do
    for all μ in M(i, μ) do
        n_μ += n_i;
    end
end
for all μ-op types, μ do
    for all r in R(μ, r) do
        n_r += n_μ;
    end
end
for all μ-op types, μ do
    if length(R(μ, r)) == 1) then
        Add μ-op to that port;
    end
end
while unbalanced do
    for all μ-op types, μ, where length(R(μ, r)) > 1) do
        try a new distribution of μ-ops
    end
end
```

Algorithm 4.1: Approach to find distribution of instructions across ports.

Finally, the new average micro-op latency, l_μ, is increased by the largest wait time across the execution ports, W_{max}:

$$l_\mu = l_\mu + W_{max} \tag{4.10}$$

4.2.2 M-Core

As discussed for S-Cores, we can consider the M-Core model in a modular fashion, as shown in Figure 4.5. This representation highlights the modular nature of the models. Like in the S-Core model, we consider each component as an independent effect. However, unlike the S-Core model, in the M-Core model threading can hide latencies from some events. This is captured in the $T_{overlap}$ module.

Below, we describe additions or modifications to modules in the M-model. GPU architectures are classified by their compute capability; early GPGPUs had compute capability 1.x, Fermi architectures use compute capability 2.x, and Kepler architectures have compute ca-

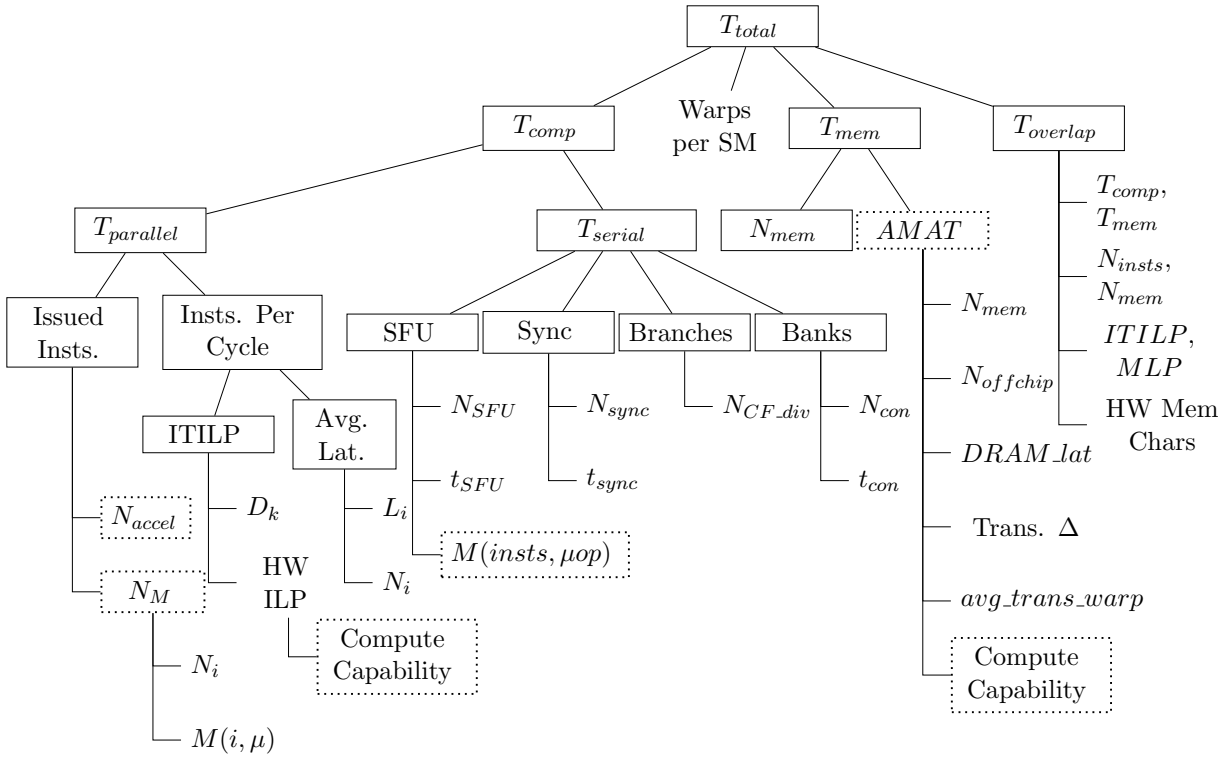


Figure 4.5: Hierarchical modules that effect performance for M-Cores. New modules have dotted borders. Architecture specific modules shaded.

pability 3.x. The compute capability includes general architectural factors including ability to coalesce memory accesses, support for atomic and sync instructions, dynamic parallelism, maximum grid and block sizes, shared/local/constant memory sizes, number of registers per SM and per thread, and number of SPs/SFUs/warp schedulers/instructions issued per scheduler per SM. Where relevant, we discuss the impact of the compute capability on each module.

Previous work [89] added support for caching and limited support for the impact of ILP on wider SMs. Below, we discuss the impact of two compute capability factors, memory coalescing and the ability to issue multiple instructions per clock in a single SM; we also discuss the impact of GPU machine code and SFU instructions. Each factor is accounted for by adding a new module to the model.

Instruction timing: In recognition of the differences in instruction throughput for different instructions, we use the instruction throughput and latency findings from [101] to refine the average instruction latency calculation. In addition, we observe that some `ptx` instructions are mapped to multiple SASS instructions on the actual hardware. Although previous work has used either `ptx` instructions directly [50] or used decuda to get actual SASS instructions [89], we use a dictionary to map `ptx` instructions to the SASS (essentially, micro-op) equivalents.

SFU Instructions: In previous work, the SFU instruction cost is considered only in terms of whether or not SFU work can be overlapped with regular computational work [89]. Although this is an important consideration, the impact of issuing SFU instructions must also be considered. We therefore include SFU instructions in our total instruction count. When SFU instructions are included in the total instruction count, the cost of issuing them is included in the computation of $W_{parallel}$. If their issue cost is not included, they can be completely hidden and effectively execute for free.

Multi-Instruction Issue To add multi-issue, we use ILP to find the probability that multiple instructions issue per cycle, as described here. In previous work, the assumption is that the number of instructions in flight per SM is limited by $ITILP_{max}$, the amount of ILP required to hide all pipeline latency [89],:

$$ITILP_{max} = \frac{l}{warp\_size/SIMD\_width} \tag{4.11}$$

Since Kepler SMs are wide and sophisticated enough to issue multiple instructions per warp, we update this equation:

$$ITILP_{max} = W \times \frac{l}{warp\_size/SIMD\_width} \tag{4.12}$$

Memory Coalescing The key impact of compute capability that we consider is its impact on memory coalescing. We derive empirical functions to find the amount of warp coalescing based on memory access patterns; previous work has estimated warp coalescing through code inspection [50] or through direct measurement [89]. Warp coalescing is a key input as

it can be the difference between 32 individual memory transactions and a single memory transaction for a single warp's memory load or store.

- **Compute Capability less than 1.2:** Warp coalescing is on the half-warp level. All threads in a half warp must access a single aligned 64B, 128B, or 256B segment, and they must issue addresses in sequence in order to be coalesced: the kth thread in a half warp must access the kth word in a segment (although not all threads need to participate in the memory access). If these requirements are not met, each thread in the warp will issue a separate 32B transaction. We measure the warp coalescing rate by modifying a built-in trace analyzer for Ocelot that previously found the coalescing rate for compute capability 1.2 or 1.3.

- **Compute Capability 1.2 or 1.3:** Coalescing occurs for any pattern of accesses that fits into a segment (32B for 8-bit words, 64B for 16-bit words, or 128B for 32- and 64-bit words). The transaction size can be halved if only the lower or upper half of the transaction will be used. We measure the warp coalescing rate using a built-in trace analyzer for Ocelot.

- **Compute Capability 2.0 or higher:** The addition of caching simplifies the problem. Memory accesses by threads of a warp are coalesced into the minimum number of L1-cache-line-sized aligned transactions necessary to satisfy all threads. We measure the warp coalescing rate directly for Kepler GPUs using Nvidia's Visual Profiler.

Figure 4.6 shows the expected number of memory transactions per warp, a measure of warp coalescing, for each compute capability using the approach detailed above for each benchmark in the CAB suite. Note that 32 transactions per warp implies that every data element generates its own memory transaction, while one transaction per warp is suggests perfect coalescing.

4.3 Modeling Specific Cores

Below, we discuss the modules from the previous section required for each core, real world considerations, and the measured and predicted performance for each S-Core and M-Core

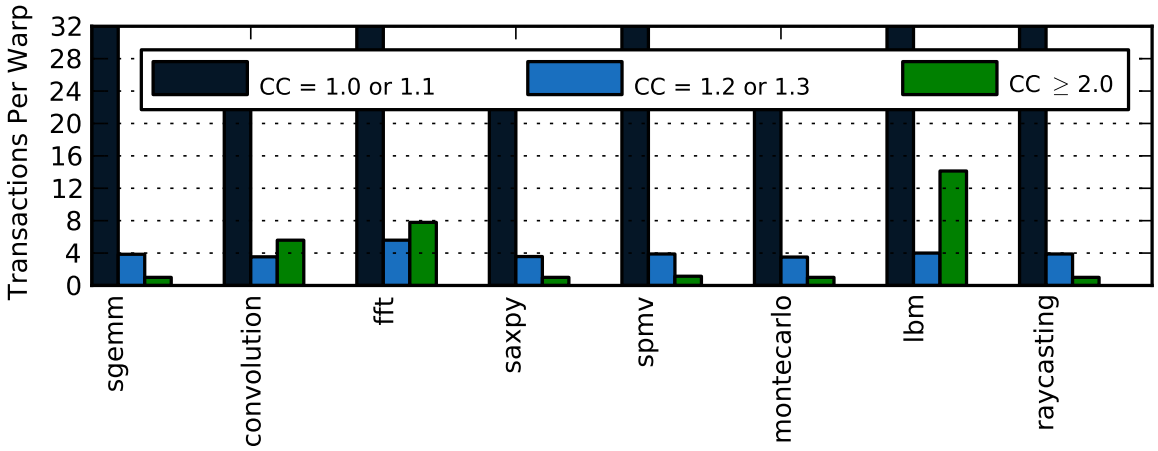


Figure 4.6: Comparison of Coalescing Rates: Average Memory Transactions per Warp Memory Load or Store

under evaluation, with particular attention paid to the goals below. For both S-Cores and M-Cores, we find the model's accuracy on the CAB and Rodinia benchmark suites[3].

As discussed in Chapter 2, the S-Core and M-Core models are validated against a representative set of cores. When comparing the model results against performance results on real cores, we have four key goals:

- **Automation:** Push-button predictions; no per-benchmark refinements.
- **Accuracy:** Individual measurements are accurate; analyzed through per-benchmark analysis for each core.
- **Inter-Architecture Trends:** Within S- or M-Cores, trends correctly predicted; analyzed through the geometric mean, maximum, and minimum speedup for each core at the end of the S-Core and M-Core subsections.
- **Intra-Architecture Trends:** Between S- and M-Cores, trends correctly predicted; analyzed through the per-benchmark multicore speedup at the end of this section.

Before describing our specific models and their accuracy, we briefly discuss the state-of-the-art for S-Core and M-Core models. M-Core models have been previously validated

[3]The subset of Rodinia benchmarks and our selection of the CAB benchmarks was discussed in Chapter 2.

on real hardware with simple kernels and either do not report the error in execution time estimates or only provide predicted relative speedups between different algorithms [50, 89]. We are not aware of previous work that includes M-Core model for Kepler architectures. S-Core models are validated on simulators, where some microarchitecture effects are missing and simulation errors may introduce random noise. Our goals in this chapter are to increase accuracy over state-of-the-art and to include important S-Core and M-Core specific details where they have not been previously modeled.

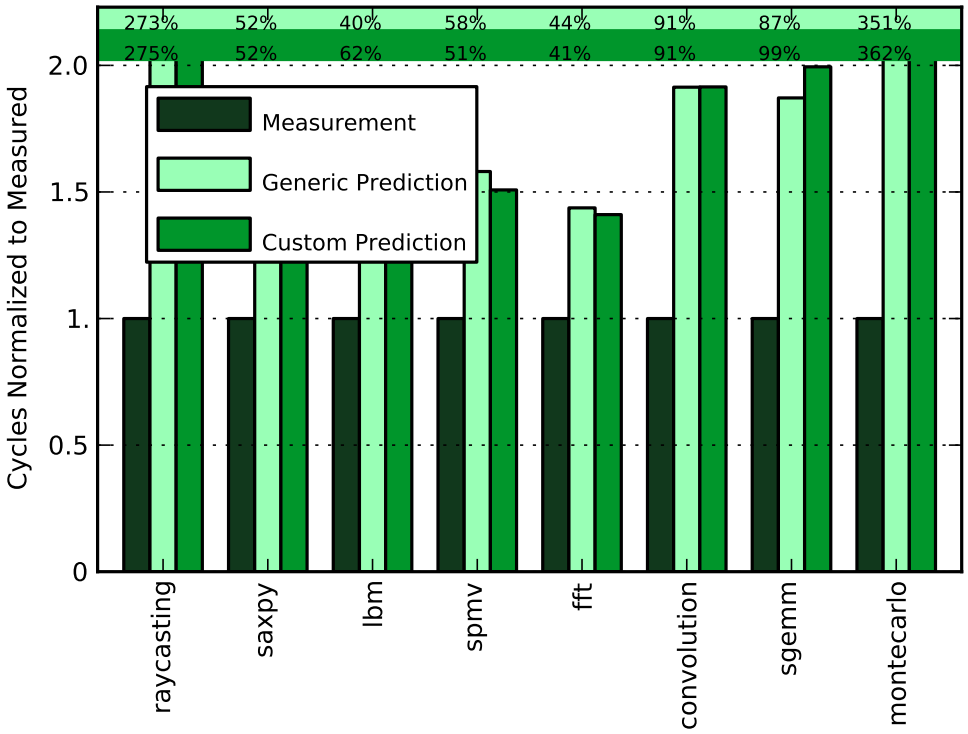
In this chapter, our key motivation is to build a set of S-Core and M-Core models that we can use in Chapter 5 for cross-architecture modeling. We are not trying to accurately model small changes in the microarchitecture - we are more interested in performance changes on the scale of 50% than 5%. As a result, larger errors are tolerable.

We now describe our S-Core and M-Core models and the accuracy which we achieve.

4.3.1 S-Cores

We consider four specific S-Cores: the Cortex-A8, the Atom N450, a Xeon 5220, and a Sandybridge i7-2600, as described in Chapter 2. For each core, we detail the additional modules that we added to accurately model the core and then discuss the accuracy of our performance predictions. Each core's architecture and timing information is from a mix of vendor-supplied processor white papers and other sources [36, 44, 43].

For each S-Core, we include results for the CAB benchmark suite and the Rodinia benchmark suite[4]. For each suite, we include a bar-graph showing measured and predicted SIMD speedup over scalar Sandybridge performance[5]. In the bar-graphs, we show both a generic prediction and a custom prediction. The generic predictions are from a direct implementation of the previous work using performance counter and binary instrumentation tools; these predictions do not include any custom modules. The custom predictions for each core include modules listed in a table below the relevant figure. We conclude this section by looking at intra-S-Core accuracy, comparing accuracy trends across the cores. All results in this section are for a single core.



Modules	In-Order Issue, resource constraints for simple pipelines
Limitations	ARM does not have Pin support, performance counter support limited
Accuracy	CAB: 41% to 362%, average: 129%, Avg. Abs. Error: 129%

Figure 4.7: *S*-Model for Cortex-A8. Error in normalized cycle counts given at top of graphs.

*S*1: Cortex-A8

Among the cores that we examine, the Cortex-A8's architecture is the simplest and requires the fewest custom modules. In fact, other than modifications required to get our data from real hardware instead of from a simulator, the in-order issue module, and the resource constraints for simple pipelines modules, the Cortex-A8 requires no additional modifications.

The Cortex-A8 includes a NEON SIMD unit that is separate from the other execution units; there is a 20-cycle penalty to move data from the general purpose pipeline to the NEON unit (although is hidden with enough instructions). Because our SIMD versions of benchmarks are written with x86 intrinsics, we only used scalar code for the Cortex-A8. If

[4]Additional CAB results are in Appendix D; all Rodinia graphs are in Appendix E.

[5]Due to measurement limitations, for the Cortex-A8 we show only relative error, as discussed below

we had used the NEON unit, it would require an additional module to predict the impact of this 20-cycle penalty.

For the Cortex-A8, we are only able to get full application measurements; we were unable to use performance counters on regions of code. This is an infrastructure problem that has been fixed for newer ARM boards and operating systems. For the purposes of this dissertation, the only limitation of using full application measurements is that we cannot compare against Sandybridge kernel performance as we do for all other cores. A detailed comparison of these architectures is available in prior work [13].

Figure 4.7 shows our model accuracy for the Cortex-A8 on the CAB and Rodinia benchmarks. The accuracy is severely impacted by two factors: (1) the full application profiling includes many compulsory cache misses and (2) the Cortex-A8 has only 256 MB of memory. The operating system and application data are both stored on an SD-card attached to the same board as the core, but access times are significant (greater than 1 ms/access) and not modeled. Adding SD-card accesses would be an additional module in the S-Core custom model, and would improve accuracy significantly for the Cortex-A8.

We next consider a core with a similar issue model, but added complexity from the x86 ISA and additional performance features, the Atom.

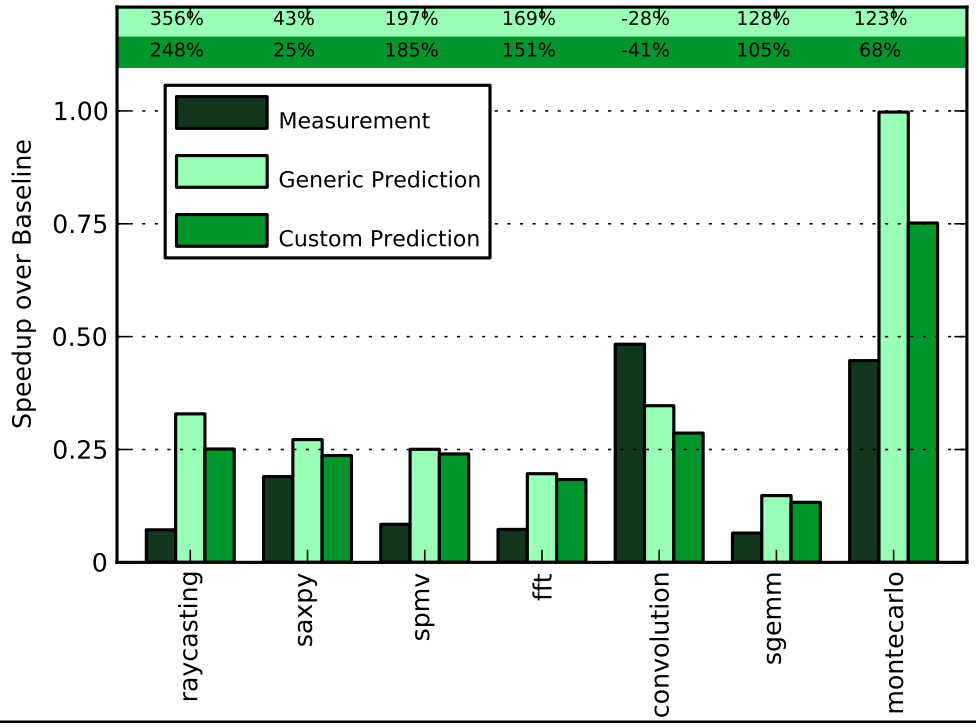
$S2$: Atom

Figure 4.8 shows predicted speedups over the baseline, a Sandybridge core running scalar code. We pick this metric so that we can compare S-Cores and M-Cores against a single baseline. Since the Atom's clock rate less than half that of the Sandybridge, the Atom is an in-order core, the Atom has a narrower issue rate, and the Atom is also running scalar code, these "speedups" are actually slow-downs. The benchmarks that run the fastest are still only half as fast as the Sandybridge.

The Atom architecture requires four extra modules: the x86 micro-op conversion module, the in-order execution module, the SIMD module, and the resource constraints for simple pipelines module.

We plot results for both the CAB benchmark suite and the Rodinia benchmark suite[6]; these speedups are only for the main computation kernels in the benchmarks. For CAB benchmarks, if there are multiple kernels, they are aggregated into a single speedup number.

[6]See Appendix E.



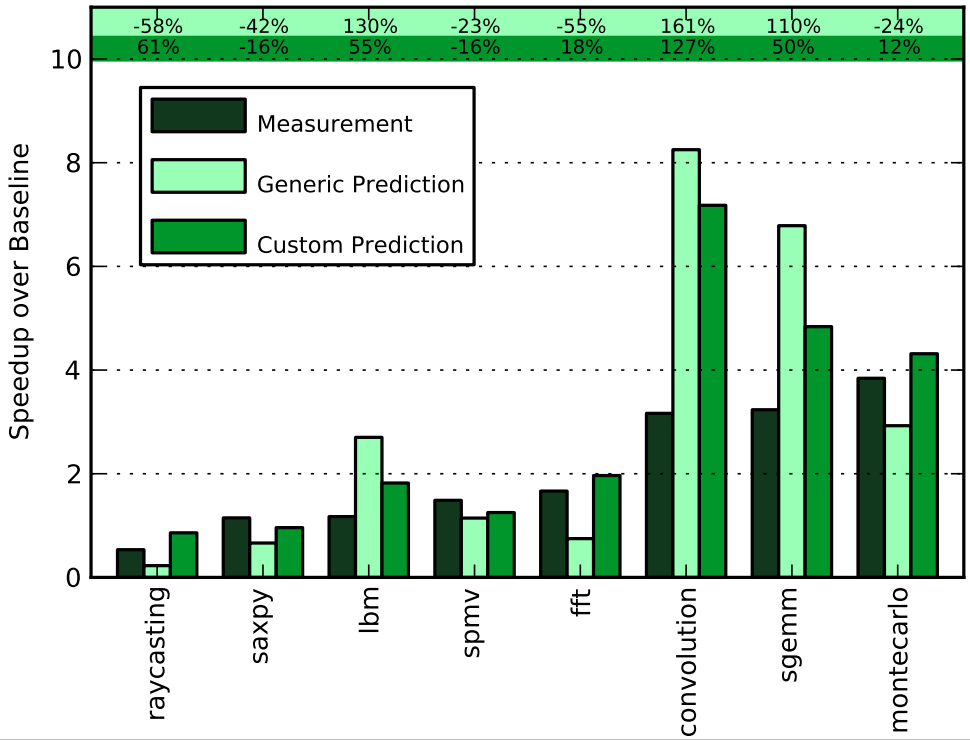
Modules	In-Order Issue, x86 μ-ops, resource constraints for simple pipelines
Limitations	Memory latency dominates, measuring μ-ops
Accuracy	CAB: -41% to 248%, average: 106%, Avg. Abs. Error: 118% Rodinia: -24% to 451%, average: 174%

Figure 4.8: S-Model for Atom. Error in speedups given at top of graphs.

Note that the `lbm` benchmark did not compile correctly for the Atom core, despite passing the correct flags.

We find that the errors are relatively large for both benchmark suites: on average, we predict performance that is 2× better than it actually is for both suites. These errors are, again, due to under-predicting the performance impact of memory and disk accesses. We note that the Atom's wall clock execution time is actually slower than the A8's wall clock time for at least two of the complete benchmarks due to disk accesses.

In the figures, we show both generic and custom predictions. Our custom predictions are 1.3× and 1.4× better than generic predictions from previous work on average for the CAB and Rodinia suites, respectively. Although our errors are still large, they are better than our implementations of state of the art analytic models.



Modules	OoO Issue, x86 μ-ops, resource constraints for complex pipelines, L3 cache
Limitations	Performance optimizations
Accuracy	CAB: -16% to 127%, average: 36%, Avg. Abs. Error: 44%
	Rodinia: -70% to 457%, average: 55%

Figure 4.9: S-Model for Xeon. Error in speedups given at top of graphs.

$S3$: Xeon

In Figure 4.9, we show the Xeon performance speedups over the Sandybridge scalar baseline. Benchmarks are ordered from least to greatest measured speedup within each benchmark suite. For the hand-vectorized CAB benchmarks, we observe speedups that range from less than one (due to minimal vectorization and Xeon's relatively lower expected performance than the Sandybridge) up to four times (the width of the Xeon's SIMD unit).

To model the Xeon core, we use the x86 micro-op module, out-of-order execution module, resource contention module, L3 cache module, and branch replay module.

We next consider the errors in speedup predictions for the CAB and Rodinia benchmark

suites[7]. We observe that the errors in predicted speedups over a scalar Sandybridge implementation range from -16% to 127%, with an average error of 36%. This is more accurate than our previous results for the A8 and Atom models, as the Xeon's larger memory leads to minimal stalls for disk accesses.

The remaining errors are due to memory overlapping and performance optimizations which are not included in our model. Accurate predictions for the fraction of off-chip memory accesses that are over-lapped are difficult and have a large impact on this out-of-order core. The impact is clear in our predictions for the Rodinia benchmark suite, where we have larger data-sets which lead to more off-chip data accesses. The error in this more challenging set of benchmarks ranges from -70% to 457% and is 55% on average. This suggests that additional work on predicting memory access overlaps would improve our accuracy.

We observe that the custom modules have significantly higher accuracy than the generic predictions. With generic predictions, errors across the two suites range from -66% to 484%. The Average Absolute Error is 75% for the CAB benchmark suite and 197% for the Rodinia benchmark suite. This is 1.7× and 2.0× higher, respectively, than the Average Absolute Error using the custom models.

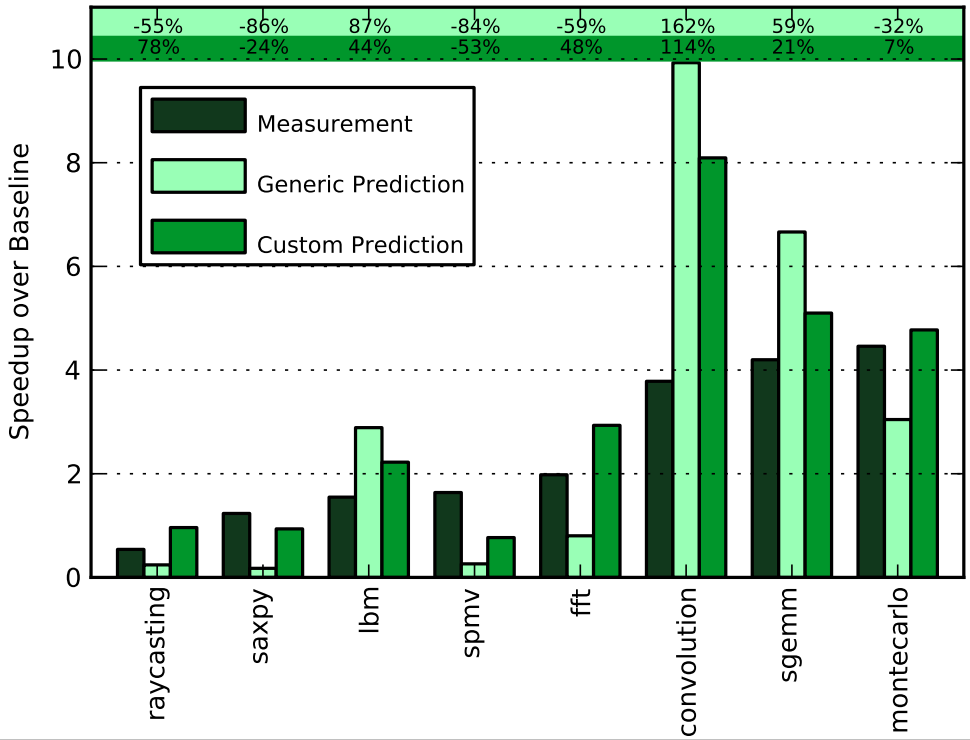
*S*4+: Sandybridge

To model the Sandybridge core, we use the x86 micro-op module, out-of-order execution module, resource contention module, L3 cache module, and branch replay module. Note that these are the same modules as we used for the Xeon core, but that we have different hardware characteristics for each. In particular interest for the model, the Xeon and Sandybridge have different instruction timings, pipeline constraints, and core frequencies.

In constructing the Sandybridge model, several microarchitectural details led to additional differences in the model as compared to previously published work:

- The use of SIMD code does not fundamentally impact the performance model and thus does not lead to a new module; SIMD instructions are added to the latency, resource contention, and micro-op dictionaries, but no other change is required. One notable change is that SIMD loads and stores take, on average, a cycle longer than scalar loads and stores.

[7]Recall that we include only a subset of the Rodinia benchmarks that compiled on all platforms



Modules	OoO Issue, x86 μ-ops, resource constraints for complex pipelines, L3 cache
Limitations	Performance optimizations
Accuracy	CAB: -53% to 114%, average: 29%, Avg. Abs. Error: 49%
	Rodinia: -77% to 641%, average: 83%

Figure 4.10: *S*-Model for Sandybridge. Error in speedups given at top of graphs.

- Sandybridge has an issue width of four instructions, but, after the issue cycle, instructions are decoded into micro-ops and there is a micro-op cache for instructions in loops. As a result, Sandybridge can supply up to six instructions per cycle to the execution units; we therefore set what was previously called the issue width to six.
- Sandybridge has two window sizes that could be picked from when finding the ILP: the actual instruction window and the reservation station micro-op window. Although we are interested in the micro-op ILP, we find that looking at all instructions in the larger instruction window provides a better estimate of ILP
- Although we account for some speculative instructions in the branch prediction calculation, we find that due to additional speculative execution, small errors in the

instruction to micro-op process, and the potential for micro-op fusion, the number of micro-ops predicted from the number of instructions is not always correct. As a result, we use the number of micro-ops counted by the performance counters as an input.

In Figure 4.10, we show S-Model accuracy on CAB benchmarks for the Sandybridge core. Note that speedup trends are similar to those observed for the Xeon. For the Sandybridge, we allowed the compiler to use AVX instructions; as a result, `bfs` is nearly 8× faster than the scalar version.

For the CAB benchmark suite, we find that errors in speedup predictions range from -53% to 114%, with an average of 29%. The range is larger for the Rodinia benchmark suite: errors in speedup predictions range from -77% to 641% with an average error of 83%. The large under-predictions for speedups are due to unexpected architectural performance optimizations that are not included in the model. We believe that over-predictions for speedups (e.g., large under-predictions for execution time) are caused by over-estimating the number of over-lapped memory accesses, as discussed in the Xeon model.

As for Xeon, we note that the generic prediction model has even larger errors: errors in predictions range from -86% to 748%. The Average Absolute Error is 1.6× and 1.7× higher for the generic predictions than for the custom predictions for the CAB and Rodinia benchmark suites, respectively.

Intra-S-Core Accuracy

Finally, we consider accuracy trends across S-Cores. In Figure 4.11, we have plotted the geometric mean speedup across each benchmark suite for the Atom, Xeon, and Sandybridge S-Cores[8]. The whiskers on each bar show the maximum and minimum speedup. We observe that as the core complexity increases, our prediction accuracy decreases. There are additional second-order effects that we have not modeled that become important as core performance increases.

4.3.2 M-Core

In this section, we consider two specific cores: a Pre-Fermi core, the FX Quadro 580, and a Kepler core, the GeForce GTX 660 Ti. Although we do not evaluate the model on a Fermi

[8]Cortex-A8 is excluded as we only have per-benchmark, instead of per-kernel, information for it.

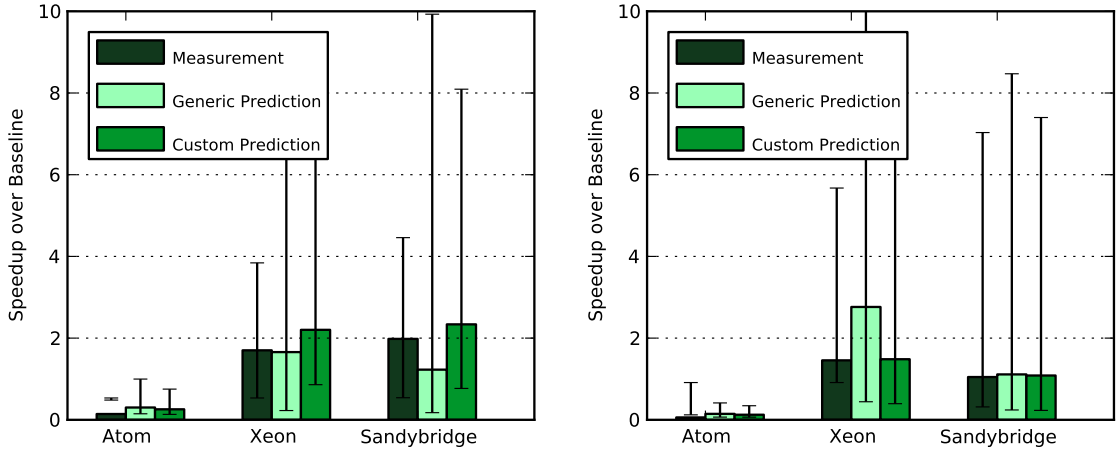


Figure 4.11: S-Model Trends Across Cores.

core, by including both a Pre-Fermi and Kepler architecture, we span all changes included in the Fermi architecture. For each core, we detail the additional modules that we added to accurately model the core and then discuss the accuracy of our performance predictions. Each core's architecture and timing information is from a mix of vendor-supplied processor white papers and other sources [79, 80, 101, 1].

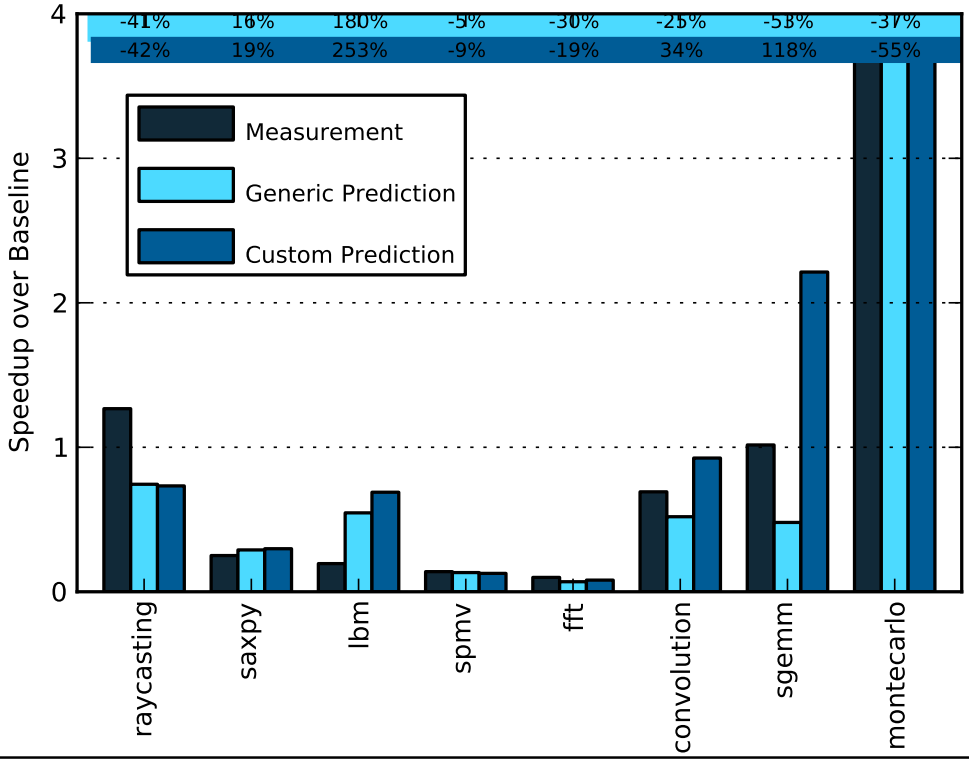
For each M-Core, we include results for the CAB benchmark suite and the Rodinia benchmark suite[9]. For each suite, we include a bar-graph showing measured and predicted per-M speedup over measured scalar Sandybridge performance. We conclude this section by looking at intra-M-Core accuracy, comparing accuracy trends across the cores.

$M1$: Pre-Fermi

We first consider a Pre-Fermi M-Core, the Quadro FX580. For this core, we add an instruction timing module and our custom implementation of the SFU and Sync instruction overhead modules.

In Figure 4.12, we show the per-SM speedup over a Sandybridge S-Core running scalar benchmarks. Note that all benchmarks except for `montecarlo` and `raycasting` are actually slower on the Pre-Fermi SM; this is not surprising since the Pre-Fermi core's frequency is only 1.12 Ghz and the core is designed to be one of many units used in parallel.

[9]Additional results for CAB cores in Appendix D and all results for Rodinia in Appendix E.

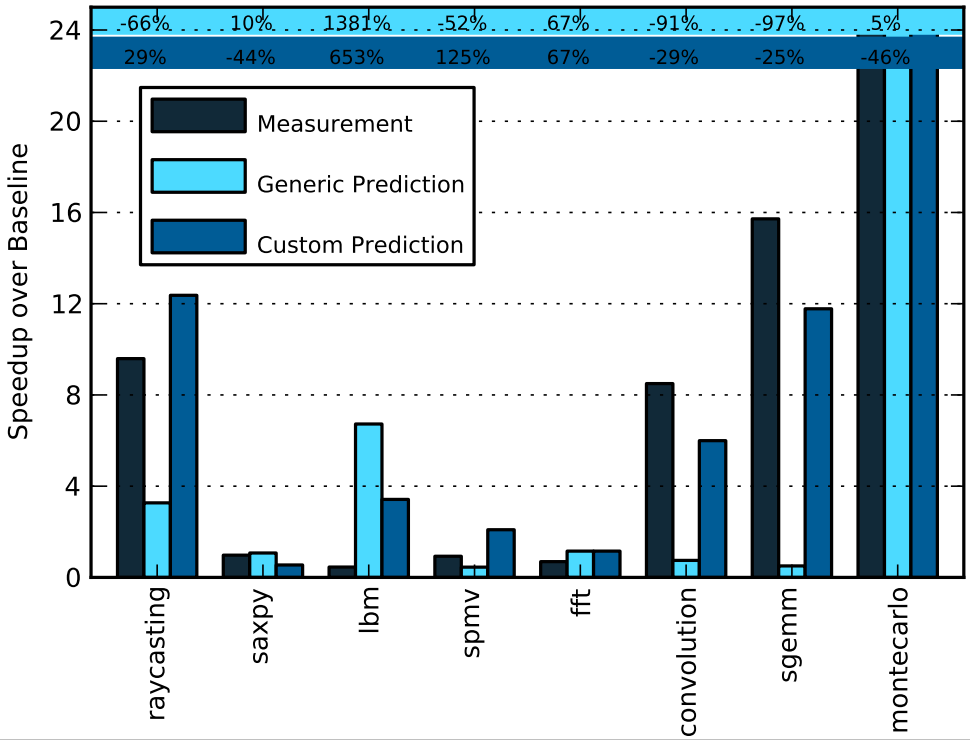


Modules	SFU & Sync Instructions, Instruction Timing
Limitations	No binary instrumentation, performance counter support limited
Accuracy	CAB: -42% to 34% on typical benchmarks, Avg. Abs. Error 68%
	Rodinia: 3% to 52% on typical benchmarks

Figure 4.12: M-Model for Pre-Fermi. Error in speedups given at top of graphs.

On the CAB benchmark suite, the percent error in predicted speedups ranges from -42% to 253%. Large errors are most common for benchmarks with very low speedups (less than 0.25×). Excluding these points, our errors range from -55% to 34% with `sgemm` and `lbm` as outliers with errors of 118% and 253%, respectively. We over-predict speedups for these two benchmarks. We note that these benchmarks have more instructions and memory accesses per warp than any other benchmark that we consider. We believe that our implementation overestimates the level of memory level parallelism for these benchmarks.

For the Rodinia benchmark suite, we over-predict speedups in all cases, by between 3% and 1015%. For benchmarks where the chip performance (4 SMs) would at least match Sandybridge performance, the Rodinia speedups that we predict all have errors of 52% or



Modules	Instruction Timing, SFU & Sync Instructions, Caching, Multiple Issue
Limitations	No binary instrumentation
Accuracy	CAB:-29% to 29% for benchmarks with speedups, Avg. Abs. Error 127% Rodinia: -45% to 74%

Figure 4.13: M-Model for Kepler. Error in speedups given at top of graphs.

less. For the benchmarks that do not experience speedups, we correctly predict that the benchmark is not sped up, even if the error in the prediction is large.

Finally, we note that although our custom model does improve predictions for the majority of benchmarks, the Average Absolute Error is only improved by 6%.

$M3$: Kepler

We next consider a Kepler M-Core, the more powerful M-core. For this benchmark, in addition to the modules used for the Pre-Fermi M-Core, we add modules for multiple instruction issue, caching, and improved warp coalescing.

In Figure 4.13, we show the per-SM speedup over a Sandybridge S-Core running scalar

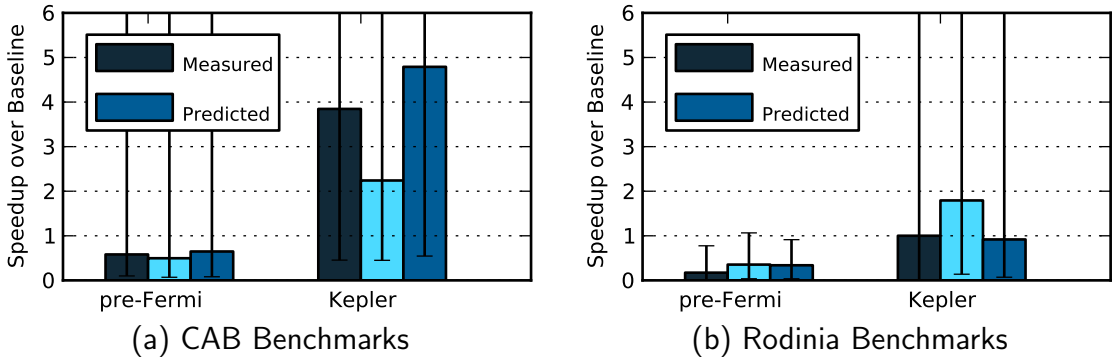


Figure 4.14: M-Model Trends Across Cores.

benchmarks. A single Kepler SM can be much higher performing than a single Pre-Fermi SM since it has 24× more SMs, more SFUs, better memory coalescing, higher memory bandwidth, and can issue more threads per cycle. This is reflected in the higher per-SM speedups, which range from negligible to over 24×.

We next consider the errors in our predicted speedups. The errors for CAB benchmark predicted speedups are between -46% and 653%, but we again note that the largest errors are for benchmarks that do not have significant speedups on the M-core. Excluding those points, the errors in our predicted speedups range from -46% to 29%. We note that the errors computed without our new modules are significantly higher, up to 1381%. The errors for the Rodinia benchmarks follow a similar pattern.

Intra-M-Core Accuracy

Finally, we consider accuracy trends across M-Cores. In Figure 4.14, we have plotted the geometric mean speedup for each M-Core, along with whiskers to show the maximum and minimum speedups. We observe that for the Pre-Fermi core, we correctly predict, on average, a speedup of 0.5×. For the Kepler M-Core, we only predict a speedup of 2.5× on average, compared to the measured 4×. However, the custom modules still do better than the generic model, which predicts a slowdown on average.

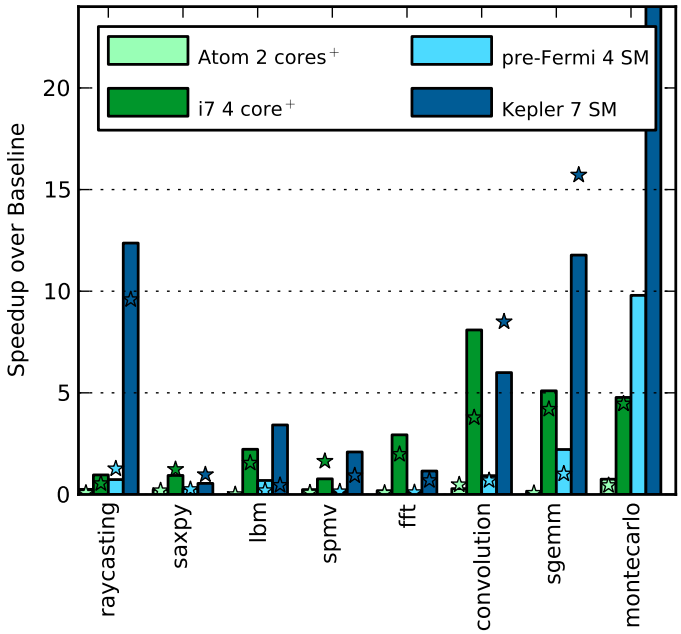


Figure 4.15: S- and M-Core Symmetric Multicore Projections. Stars show measured core performance. For benchmarks marked with a +, measured multicore performance found using multicore model using measured core performance.

4.4 Multicore Applications

To summarize, we consider accuracy of predictions for S-Core and M-Core symmetric multicores. Although a multicore model is not the focus of this dissertation, the work described in Chapter 2 and elsewhere [28] includes a symmetric multicore model which we use here. Additional future work on the multicore models, for S-Cores in particular, include cache contention models, communication overheads, and parallelism effects. This section is included for completeness and to set up the framework for the dark silicon projections in the next section; it is not a key finding.

In Figure 4.15, we show the accuracy for a set of four multicores: (1) a dual-S-Core chip using Atom cores, (2) a quad-S-Core chip using SandyBridge cores, (3) a quad-M-Core chip using pre-Fermi cores, and (4) a hepta-M-Core chip using Kepler cores. For the M-Core chips, we compare against hardware measurements on the Quadro FX580 and GeForce GTX660 Ti, as discussed in Chapter 2. We do not have multi-threaded S-Core benchmarks. Instead, observing that the CAB benchmark kernels are highly parallel, we assume 99%

parallelism in these benchmarks and project results for a symmetric multi-core. Note that the "measured" results for S-Cores are based on single-core measurements and must also make this assumption.

In Figure 4.15, predicted speedups over a single Sandybridge S-Core running a scalar binary are shown as bars, and the measured speedups are shown as stars. We are interested in several metrics here: (1) are trends between S-Cores and M-Cores correctly predicted, (2) is S-Core model and M-Core model accuracy similar, (3) are there intrinsic factors that make S-Core or M-Core models harder to make accurate, and (4) do any of the trends lead to biased results (e.g., do we consistently over-predict for one meta-architecture, but under-predict for another?).

We consider each of the above questions separately:

Q1 Trends between S-Cores and M-Cores are correctly predicted in all but one case, `convolution`. In that case, the Kepler GPU outperforms the quad-core Sandybridge by a small margin and the Kepler model slightly underpredicts performance.

Q2 Both models have similar accuracy patterns; Kepler has more cores and thus per core errors are exacerbated in the multicore model.

Q3 S-Core models' difficulties come from the many optimization structures that have been added to improve single-core performance; M-Core models' difficulties come from predicting how work will be overlapped in their highly-threaded computational model. Although both models have challenges, neither is intrinsically easier than the other.

Q4 Neither model consistently over or under predicts.

4.5 Application to Dark Silicon Projections

In this section, we consider the impact of using these models on dark silicon projections for future technology generations. The framework for this running example was discussed in Chapter 2. In Chapter 3, we showed results for this example using assumptions from [28]. In this chapter, our assumptions have changed discussed in Chapter 2, and summarized here:

- **Cores:** We only allow five cores, the Sandybridge, Nehalem, Xeon, Atom, and A8.
- **Benchmarks:** We use the CAB benchmark suite.

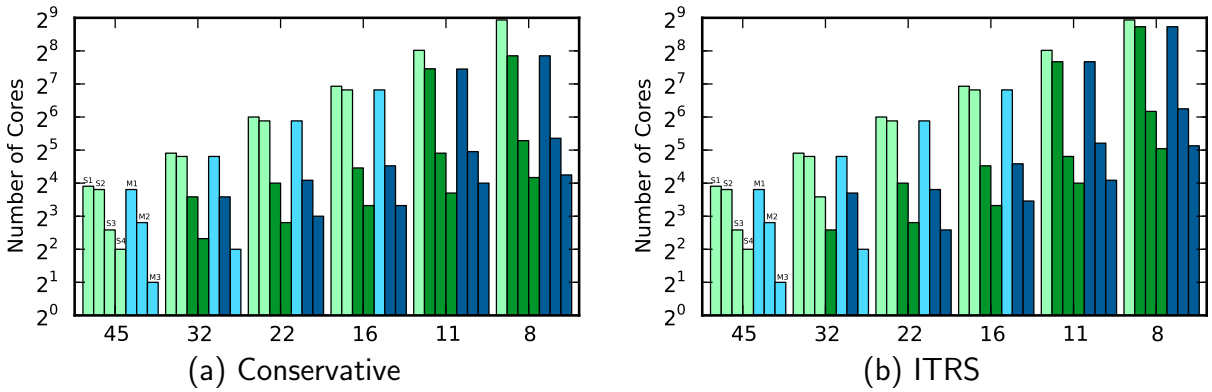


Figure 4.16: Number of each S-Core (blue) and M-Core (green) that fit per generation. Darker bars are power limited.

- **Parallelism:** We assume 99% parallelism for the CAB benchmarks on S-Cores.
- **Topologies:** We only consider the symmetric topology.

While the above assumptions limit the study's design space, they are not intrinsic limitations of the approach. Further, since the goal of this running example is to show ways in which the approaches could be used to extend the dark silicon approach, rather than full exploration of the dark silicon design space, these assumptions do not limit the applicability of the results below.

In this section, we present two sets of results: (1) projections using the upper-bound model, and (2) projections using the custom models. Projections using the upper-bound model are reproduced here with the new set of assumptions from above to provide a baseline on which to compare the new custom model projections.

4.5.1 Projections

Chapters 3, 4 and 5 each include dark silicon projections using the approach detailed above. In Figure 4.16, we show the maximum number of each type of core at each technology node, assuming a symmetric topology. Initially, all chips are area constrained; darker bars indicate that the chip is power, not area, constrained. The small $S1$ core is the only core that is never power constrained; all of the other cores are power constrained by the $18nm$ technology generation. The number of cores at $8nm$ ranges from 16 $S3$ cores to as many as

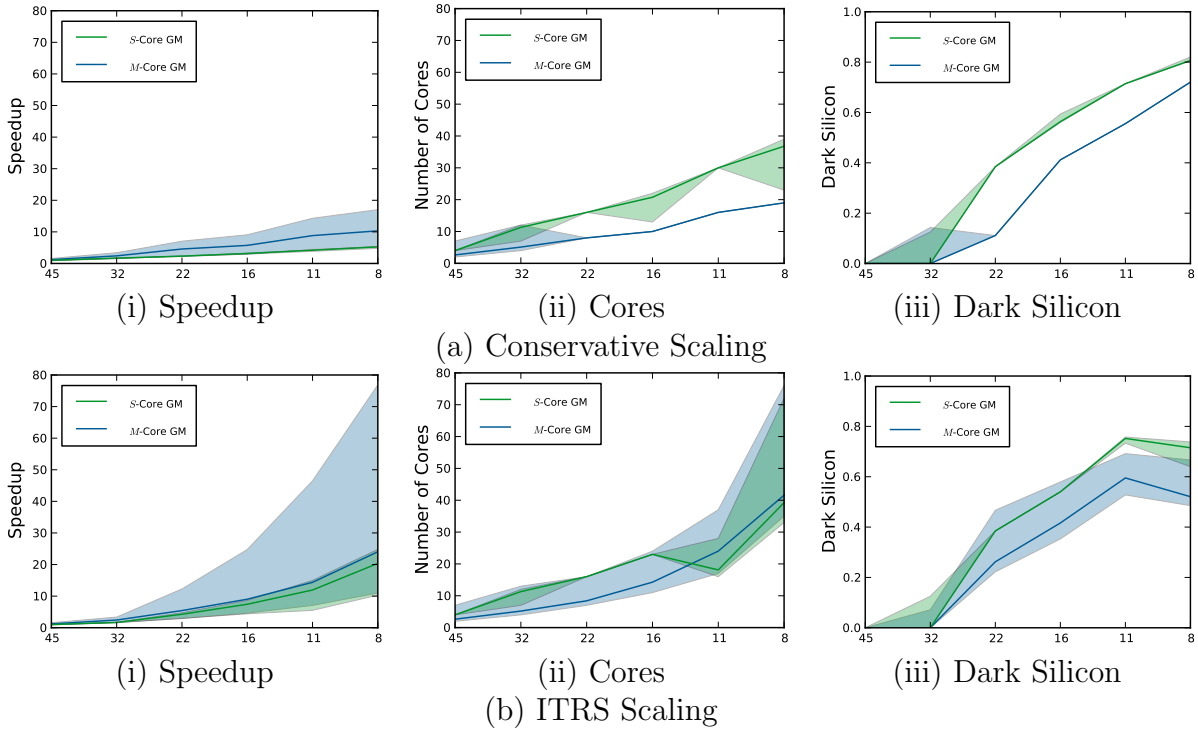


Figure 4.17: Dark Silicon Projections using Upper-Bound Model.

480 $S1$ cores using conservative scaling and from 29 $S3$ cores to as many as 480 $S1$ cores using ITRS scaling.

4.5.2 Upper-Bound

In the interest of fair comparisons, in this section we reproduce results using the upper-bound model from Chapter 3 with the above assumptions. These results are shown in Figure 4.17 with both conservative and ITRS projections. The geometric mean speedup using S-cores over the CAB benchmark suite at 8 nm is 5.2× (conservative) or 20.4× (ITRS), depending on the scaling used. We note that these geometric means are higher than those found in Chapter 3, 3.1× to 6.4×. This difference was expected as (1) the CAB benchmark suite is assumed to have 99% parallelism, and (2) two new cores, the A8 and Sandybridge, extend the frontier. Further, the M-Core speedups are significantly larger: 10.3× to 24.0. Again, the difference is expected since the benchmarks are more highly parallel and we have introduced new architectures. In particular, the new Fermi and Kepler architectures have improved

caches and warp coalescing that lead to fewer bandwidth constrained situations for those cores.

We now look at the results in more detail, first looking at overall impacts on number of cores and the amount of dark silicon, and then looking at specific core choices and their impact.

Dark silicon: The amount of dark silicon is similar to previous work: 81% (cons) to 72% (ITRS) for S-Cores and 72% (cons) to 52% (ITRS) for M-Cores. The number of each type of core chosen is also similar: 37 (cons) to 39 (ITRS) for S-cores and 19 (cons) to 42 (ITRS) for M-cores. The amount of dark silicon is slightly lower for the GPU due to the new more memory efficient architectures ($M3$).

Core choice: Atom and A8 were never chosen. We can do a back of the envelope calculation to understand why they were never chosen. Assuming memory bandwidth is not a constraint, our main consideration is the speed of each processor and how many we can fit. The Sandybridge is 40× faster than an ARM A8 core, but even at 8nm only 15× more ARM A8 cores are projected to fit onto the die. Therefore, the Sandybridge is expected to outperform the ARM A8 cores using our upper-bound model. This trend holds true for the Atom core, as well. Using this back of the envelope calculation, we find that if the A8 were about ten times smaller (at 45nm) and all other constraints stayed the same, an A8 based chip would outperform a Sandybridge based chip. When area or mildly power constrained, tend to pick a Xeon as more fit. $S3/S4/S4+$ all have similar speedups (within 10%) at 8nm, so we always pick one of them. $M2$ or $M3$ is always chosen because of their improved performance and memory coalescing. At 8nm, Kepler is almost always chosen. However, its high power and area footprint mean that at some larger technology nodes, we still pick Fermi over Kepler.

S-Core versus M-Core: The M-core is faster than an S-Core for five of nine benchmarks using ITRS scaling and for eight of nine benchmarks using conservative scaling. This is in contrast to our previous results, and a direct result of improved GPU architectures coupled with a highly data parallel workload.

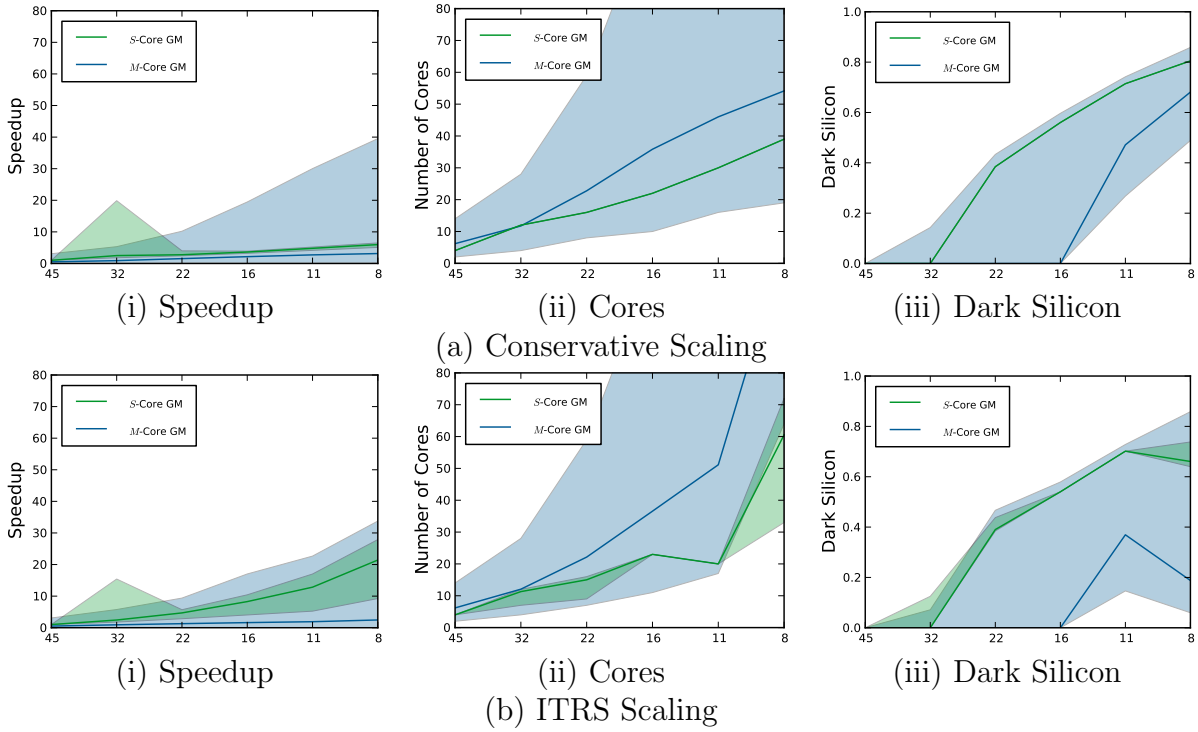


Figure 4.18: Dark Silicon Projections using Custom Models.

4.5.3 Custom

In Figure 4.18, we consider the impact of future technology nodes using our custom models. The custom models find IPC for each core based on detailed input parameters, have more detailed cache miss rate parameters, and do not assume that performance scales directly with frequency increases. As a result, we would expect the speedup projections to be lower than with the upper-bound model.

We notice several trends. The *S*-Core results are similar to the results we saw using the upper-bound model: after 5 technology generations, the geometric mean speedups range from 6× to 20×, depending on the scaling used. The range of per benchmark speedups is also similar. However, for *M*-Cores, the results are significantly different. The geometric means speedup is only 2.5× to 3×, and speedups can be much higher (up to 40×) for highly parallel workloads that are not memory bound. However, the workloads with very small speedups (or slow downs) on *M*-cores prevent the average speedup from being any larger.

Theoretically, using the custom models or the upper-bound models to generate these

results should not impact the predicted geometric mean speedup. As long as the trends between cores are the same (e.g., performance difference between Nehalem and Sandybridge is preserved), using the custom models should not be any different than using the trade-off based model. Custom models may model the cache behavior better than the trade-off function based upper-bound models (although in this example we used known miss rates for the benchmarks in the upper bound model, so it is not applicable).

For the M-Core model, we have a more sophisticated multicore model, and so more factors will come into play and we expect that our results may differ. Namely, we do not have the 99% parallelism assumption that we used for the S-Core multicores, and parallelism will play a larger part. In addition, improved cache modeling will improve the quality of our projections for M-cores. For those benchmarks with high parallelism, we often pick the lighter weight $M1$ cores, and as a result the percentage of dark silicon is lower in these cases.

However, the trade-off based result requires knowing a wide spectrum of SPECmark scores. We could use the custom models to predict the Sandybridge performance without having actually measured anything on the Sandybridge. In fact, we could use the model with varying parameters to find enough points to generate an entirely new Pareto frontier. The strength of this model is not that we can find more accurate future dark silicon projections, but that we can model cores that may lie outside of the Pareto frontier and add them to our projections.

4.6 Application to Instruction Set Architecture Studies

In this section, we consider an additional application of the custom models in this chapter: the use of the custom models to understand processor behavior to aid in an instruction set architecture (ISA) study.

RISC versus CISC wars raged in the 1980s when chip area and processor design complexity were the primary design constraints and desktops and servers exclusively dominated the computing landscape. Today, energy and power are the primary design constraints and the computing landscape is significantly different: growth in tablets and smartphones running ARM (a RISC ISA) is surpassing that of desktops and laptops running x86 (a CISC ISA). Further, the traditionally low-power ARM ISA is entering the high-performance server market, while the traditionally high-performance x86 ISA is entering the mobile low-power device market. Thus, the question of whether ISA plays an intrinsic role in performance

or energy efficiency is becoming important, and we seek to answer this question through a detailed measurement-based study on real hardware running real applications in another work [12, 13].

In that work, we analyze measurements on the ARM Cortex-A8 and Cortex-A9 and Intel Atom and Sandybridge i7 microprocessors over workloads spanning mobile, desktop, and server computing. We use intuition gained from the custom modular models described in this chapter toward a methodical understanding of the role of the ISA (versus other microprocessor features such as the cache size and speed) in modern microprocessors' performance and energy efficiency. We find that ARM and x86 processors are simply engineering design points optimized for different levels of performance, and there is nothing fundamentally more energy efficient in one ISA class or the other.

In the remainder of this section, we briefly discuss the utility of the custom models in this study and conclude with a summary of our results from the study; additional details can be found in our other work [12, 13].

4.6.1 Methodology Overview

In this study, we our overall approach is to understand all performance and power differences and use measured metrics (e.g., execution time, instructions executed, cache miss counts) to quantify the root cause of differences and whether or not ISA differences contribute.

In particular, to find the impact of the ISA on performance, we find the execution time and cycle counts to understand raw performance differences between the processors. We then measure the dynamic instruction counts, instruction mixes, code binary size, and average dynamic instruction length to understand the impact of the ISA on code generation. Finally, to understand performance differences not attributable to ISA, we measure detailed microarchitecture events.

Using a simplified version of the custom models in this chapter, we attribute performance gaps between processors to frequency, ISA, or ISA-independent microarchitecture features. This reasoning is summarized below:

- We observe that the Intel i7 has at least twice the issue width of the other processors (four versus two). We use the Pin-based MICA tool to confirm that our benchmarks all

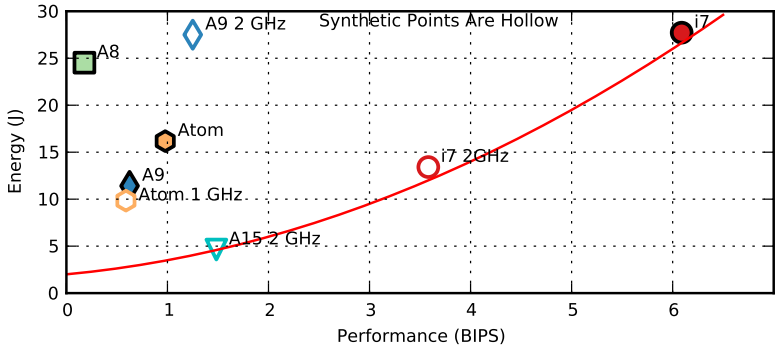


Figure 4.19: Energy-Performance Trade-offs for ISA study.

have limit ILP greater than four; the difference in issue width thus explain performance differences up to 2×.

- We observe large microarchitectural event count differences (e.g., A9 branch misses are more common than i7 branch misses). These differences are not because of the ISA, but rather due to microarchitectural design choices (e.g., A9s BTB has 512 entries versus i7's 16K entries).

- Per benchmark, we can use intuition from the custom *S*-Core models to attribute the larges gaps in i7 to A9 performance (and in Atom to A8 performance) to specific microarchitectural events: namely, differences in branch mis-predictions per thousand instructions, instruction cache misses, and data cache misses.

Using the intuition from these models, we conclude that the microarchitecture has significant impact on performance. The ARM and x86 architectures have similar instruction counts. The highly accurate branch predictor and large caches, in particular, effectively allow x86 architectures to sustain high performance. Inefficiencies due to the x86 ISA, if any, are not observed.

4.6.2 Results Overview

In this section, we summarize the findings of the ISA study. In addition to the performance analysis described above, we additionally measured the average power use for each processor while executing each benchmark. Using that information with the performance data, we constructed energy-performance trade-off curves.

In Figure 4.19, we show the geometric mean energy-performance trade-off using technology node scaled energy. We generate a quadratic energy-performance trade-off curve. Note that given the limited number of points used, a core's location on the frontier does not imply optimality.

We used intuition from the custom models in this chapter to generate additional, synthetic processor points which are shown using hollow points. These synthetic points include a performance targeted ARM core (A15) that was unavailable at the time and frequency scaled A9, Atom, and i7 cores.

For the A15, we use reported CoreMark scores (CoreMarks/MHz) and mW/MHz from Microprocessor reports and ARM documentation. We assume a 2 GHz operating frequency and compute the CoreMark score and energy. We then scale A9 BIPS results by the ratio of the A15 CoreMark score to the A9 CoreMark score to get an A15 BIPS-based performance projection.

For the frequency scaled cores, we project performance by assuming a linear relationship between performance and frequency. We scale energy projections using DVFS-based assumptions.

Using these results, we find that regardless of ISA, balancing power and performance leads to energy-efficient cores. It is the design goals and methodology, not the ISA, that really matters.

4.7 Related Work

Performance evaluation techniques include analytic models, simulation, and statistical simulation. Analytic models have the advantage that they are fast and their design can lend intuition [26, 66, 76]. Analytic models can be separated into those that are empirical, from first principles (mechanistic), or a hybrid of the two. Empirical models, often generated using regression analysis after simulating multiple design points, do not lend intuition about first order effects or accuracy outside of the simulated range. Mechanistic models are based on an understanding of the system and can therefore be expected to be reasonable estimates outside of the validated range. The models that we propose are a hybrid of empirical and mechanistic models; using a hybrid model is similar to gray-boxing techniques and lends flexibility and simplicity to the model [26].

Single core performance models tend to range from relatively simple for in-order cores [65, 14, 27, 38, 16], to more complex for out-of-order cores [57, 58, 34, 21]. Karkhanis and Smith developed a CPI based model for out of order processors that assumes stalls for memory accesses and branches [57, 58] and has since been expanded to include additional stalls [34, 21]. At a high level, these models look very similar to the simpler in-order core models, although the inputs (e.g., number of concurrent critical paths) are more complex.

GPU performance models follow a similar stall based strategy, but with special features to accommodate the GPU SIMT instruction model [50]. Improved memory performance models in *CuMAPz* improved projections [61]. A more abstract model used GPU CUDA code to estimate performance without executing the code [6]. Analytic models that use the actual instruction trace have shown improved accuracy on simple benchmarks [89, 64]. Using microbenchmarks to timing information and extrapolate performance is also effective [103, 64, 81]. Energy models for GPUs have also been proposed [51, 72].

Multicore performance models range from the very high level Amdahl's law [3] to more complex models, such as the LogP model and its variants that include thread communication and synchronization overheads [24]. Multicore performance models like that from Guz et al. [42] and the Copernicus model [38] assume infinite parallelism, independent threads, and homogeneous threads. These models provide a way to predict the impact of resource contention at the core, cache, and memory levels while abstracting away parallelism and workload heterogeneity issues. Other work, including that of Sorin et al. [90], includes models adapted for heterogeneous workloads.

4.8 Summary and Open Questions

In this chapter, we have described a modular approach to both S-Core and M-core custom models. We have shown that these basic models can be made accurate for real processors by adding new modules that cover the additional complexity of real processors (e.g., μ-ops and pipeline constraints). Finally, we showed applications to an instruction set architecture study and as a part of a multicore model to find dark silicon projections. In particular, for the dark silicon projections application, we include projections using the Sandybridge processor which, due to microarchitectural improvements, is outside of the Pareto frontier used in Chapter 3.

The custom models have fine-grained architectural flexibility, so an architect could predict

the impact of making a small change (e.g., increasing the issue width or adding a new execution unit). However, the S-Core and M-Core models each require inputs generated on their respective hardware, include different key inputs, and use different equations to find performance. There is no opportunity to directly predict performance for both S-Cores and M-Cores using a single model, like we did in Chapter 3. A mechanism to predict S-Core performance from inputs generated on an M-Core, and vice versa, is the focus of the next chapter.

As motivation for the work in the next chapter, we re-consider Figures 4.3 and 4.5. We observe that at the highest levels, the two models look very similar. Further, the inputs vary from identical (e.g., number of instructions, number of cache misses) to surprisingly parallel (e.g., ILP and ITILP). These similarities are considered in more detail in the next chapter as we consider the possibilities of cross-architecture performance prediction.

5 TRANSLATION-BASED ARCHITECTURE EXTENSIONS

This chapter considers a single framework that can project performance across architectures.

Goals and Constraints: Our goal is to use performance characteristics measured on one architecture to predict performance on another architecture. In a broad sense, our approach is to take measurements on one chip, translate them to a generic set of workload characteristics, and then translate that set of generic characteristics into inputs to the analytic model for the second architecture. In this chapter, we show this approach applied to two architectures, S-Cores and M-Cores.

The modular custom models in the previous chapter had a significant limitation: any large changes to the architecture (for example, the transition from general purpose cores to throughput oriented ones) could not be modeled. In the upper-bound models from Chapter 3, we could model both S-Cores and M-Cores with a single model, but accuracy and flexibility within that approach was limited. In this chapter, we address these issues. Recognizing that a single model with sufficient accuracy would be either infeasible or so complicated as to be intractable, we instead develop the novel input translation approach.

Inputs: In this chapter, we assume that we have the following: (1) a known architecture with available hardware, performance counters, compiler, benchmark code, and access to either binary instrumentation tools or a simple emulator, and (2) an analytic model for a second architecture. We do not need to re-write or compile code for the second architecture to predict its performance (although, for validation, we do need to have both the compiled code and real hardware to execute it on).

Use-case: This general approach could be used to rapidly evaluate novel architectures where building the full design chain (e.g., optimized code, compiler, simulator/hardware) would be time-consuming. In our more practical S-Core and M-Core application, we have the ability to compare S-Core and M-Core performance for real applications without the overhead of actually rewriting the code. This is particularly useful as S-Core and M-Core architectures are rapidly evolving.

Architects would benefit from early analysis of new designs without full implementation as well as a way to observe the impact of future technology trends on current designs.

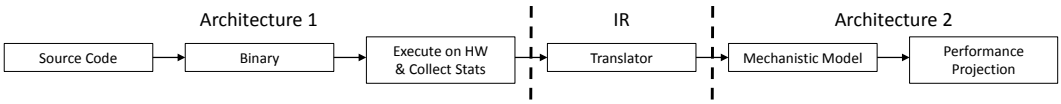


Figure 5.1: Translation Overview

Programmers would benefit from a tool to understand which code to port and to which architecture. The rapid evolution of hardware presents time-sensitive implementation challenges for both architects and programmers who want to exploit emerging accelerators. Our translation provides a mechanism for this.

This chapter opens with an overview of our approach and the rationale behind it. We then describe the models as we have implemented them to translate from S-Core measurements to M-Core performance projections and vice versa. We show the accuracy of our approach on the inputs to the second architecture's inputs, and then validate performance projection results for multicores built from the second architecture. We continue our running example with the application of this approach to dark silicon projections. The chapter concludes with related work and a discussion of open questions.

5.1 Overview

In this section, we give an overview of the approach that we use to predict cross-architecture performance and give some initial insights to motivate that approach.

In our efforts to build a single framework that can project performance across different architectures, we continue our use of analytic models for each architecture. Our inspiration is the somewhat tantalizing resemblance in the models' workload inputs, foreshadowed in the previous chapter and discussed next. When approached this way, the problem becomes somewhat simple. We briefly describe our approach for developing a model of translating the inputs from one model to another and the surprising accuracy of a naive implementation as inspiration for the detailed implementation and analysis that follows in the rest of the paper.

We leverage a first principles understanding of the inputs to the custom models from the previous section, and consider how those inputs could be derived from another system. To do so, we distill these inputs into their hardware independent characteristics. Given this perspective, our generic approach is show in Figure 5.1: using code written for a spe-

Table 5.1: CPU and GPU models: Workload Inputs and Translation Mechanism

	CPU	GPU	Translation Assumption
Compute	Number of instructions	Number of Instructions	x86 μops $\rightarrow$ ptx insts
	Instruction mix	Instruction Mix	Similar
	ILP	ILP	Low (≈ 1)
Memory	Cache and TLB miss rates	Number of off-chip accesses	L2 miss $\rightarrow$ off-chip access
	SSE Loads and Stores	Fraction of accesses coalesced	SSE load or store $\rightarrow$ coalesced access
		Memory Level Parallelism	Low (≈ 1)
Control	Branch mispredictions	Average warp occupancy	Diff. code paths $\rightarrow$ thread divergence
	# of outer loop iterations	Number of threads	Outer loop $\rightarrow$ thread
	# of inner loop iterations	Synchronization	Inner loop $\rightarrow$ sync point
		Block & Grid Configuration	Use model to find best config

cific architecture, measure key characteristics on that architecture. Then, use a translation mechanism to find key parameters for a second architecture and apply those parameters to a mechanistic model for the second architecture to find a performance projection.

We focus on two specific examples, M-Core performance from an S-Core and vice versa, to introduce the approach and for detailed analysis in this chapter. Table 5.1 depicts the workload based inputs needed for S-Core and M-Core models as discussed in the previous chapter. We group these inputs into three categories: computation, memory, and control. Computation refers to the number of instructions and also how much instruction level parallelism, within a single-thread, can be expressed; this may be thought of as the level of "near-parallelism". Control refers to the number of threads, outer-loop executions, and required synchronization instructions; this may be though of as the level of "far-parallelism". Finally, memory refers to the amount of data required by the benchmark and how much of that data can be cached.

There are clear parallels across the three categories of computation, memory, and control; we observe that we can translate inputs that describe a characteristic of a workload for one model to inputs for another model by considering the hardware's effect on the inputs. Workload characteristics fit into one of three categories: characteristics that we translate directly, characteristics that we translate based on dynamic trace analysis, or characteristics that we either ignore or assume have a set value. Thus to develop a cross-architecture model, we can start by measuring the typical S-Core model inputs, apply simple translation

mechanisms to find appropriate M-Core inputs, and then use the translated M-Core inputs to find M-Core performance. The last column in Table 5.1 describes the high level translation principles for this example.

We use two metrics to evaluate our prediction results:

- **Trends:** Does the translation model correctly predict when an architecture completes a benchmark more quickly than another architecture?
- **Accuracy:** How close are projected speedups to the measured speedup?

Using the very straightforward translation approach outlined in Table 5.1, we evaluate the approach according to the metrics outlined above:

- **Trends:** The naive translation correctly predict that a GPU will complete the benchmark more quickly than the scalar CPU implementation in all but two cases. The two benchmarks where the trend is incorrectly predicted, saxpy and spmv, both have GPU implementations that outperform the CPU implementation by less than 2×.
- **Accuracy:** Although three of the benchmarks with the smallest performance differences between the CPU and GPU are correctly predicted to have small differences, speedups can be greatly overestimated (e.g., by more than 28× in the case of convolution) or underestimated (e.g., by more than 15× in the case of sgemm) due to either mis-translated inputs or missing performance effects in the first order model.

However, the initial results are promising, and in the next section, we develop more refined models and translation mechanisms. Using those more refined approaches, we then we present our final, more refined, results. We also show how these more abstract inputs effect projections using the dark silicon running example.

5.2 Model Description

In this section, we describe the translation process from S-Cores to M-Cores and from M-Cores to S-Cores. The process is inspired by the idea that workload characteristics could be translated to a generic intermediate format that expresses key characteristics that impact performance on all platforms. This intermediate, architecture independent representation is listed briefly in Table 5.2. Although some of the characteristics are familiar from previous

Table 5.2: Intermediate Representation Translation Equivalents.

	IR	***S*-Core**	***M*-Core**
Compute	RISC-like instructions, R	M(R, x86) or M(R,ARM) → N_I,n_i	M(R, PTX) → N_I, n_i, N_{SFU}, N_{Mem}
	Per-R dependence trace	Trace analysis → ILP_{x86} or ILP_{ARM}	Trace Analysis → ILP_{PTX}
Memory	RISC-like instructions, R	n_{load}, n_{store}	n_{load}, n_{store}
	Memory Footprint, Memory Access Pattern	Simple Cache Model → N_{mL1}, N_{mL2}, N_{mL3}	Footprint analysis → $N_{offchip}$
		Access Pattern Analysis → SSE accesses	Access Pattern Analysis → Access Coalescing
			Accesses Per Loop Analysis → Memory Level Parallelism
Control	Minimum piece of Work	≈# of outer-loop iterations	≈ # of threads
	Reductions	≈ inner-loops, data reductions	≈ # of sync threads
	Optimization Search	Loop unrolling, loop ordering	Block/Grid configuration

work on *S*-Cores [52], we note that the addition of control flow information necessary for the transition between single-threaded and many-threaded architectures. In addition, the inputs that we collect on the *S*-Cores and *M*-Cores are microarchitecture independent; the microarchitectural features of the source architecture should not impact our results.

In the current implementation, we translate directly between measurements from cores and the intermediate layer is not explicitly generated. Adding this layer would make our tool-chain immediately applicable to additional architectures. An architect would be able to immediately model their new idea after inputting an analytic model of their architecture plus a set of mappings from the generic IR to their new architecture. For clarity, we describe the approach in terms of this IR.

In this section, we describe each component of the IR, how we find can find that component of the IR from both *S*-Cores and *M*-cores, and how we can translate the IR component to the necessary *S*-Core and *M*-core model inputs. The approach is summarized in Table 5.2, and our descriptions below follow the outline suggested by the table.

5.2.1 Computation

Both S-Core and M-Core models are dependent on the total number of instructions, the mix of instruction types, and dependences between instructions. The instruction types impact both the average instruction latency and any resource contention when the instructions are executing. Dependences between instructions impact which instructions can be issued with other instructions, which leads to the ILP used throughout this thesis.

Instructions: S-Core micro-ops and M-core `ptx` instructions can map to a single intermediate IR; translation is then a trivial process of mapping the IR to the target architecture's ISA. In our implementation, we directly map between x86 micro-ops and `ptx` instructions, skipping the intermediate step. This mapping works well and is straight-forward as both sides are RISC in nature (after x86 instructions are first mapped to micro-ops). After completing the mapping, we directly find the instruction count, instruction mix, and average instruction latency from the new set of instructions. The dictionary look-up for M-Cores also includes the unit where instructions are executed (e.g., SPs, SFU, or DPU) to facilitate counting instructions executed in the SFU.

ILP: The number of instructions that can issue per cycle from a single thread is a function of the ILP. In the case of out-of-order cores, the ILP is only a function of instruction dependences; for in-order cores, the ILP is a function of both instruction dependences and in-order sequencing requirements. The most generic IR-centric approach would be to maintain instruction dependence information as part of the IR and then compute ILP for the target architecture from that information. In our work, we find that the ILP computed for the Cortex-A8 and the Atom processor has similar enough constraints to the ILP on M-cores to consider them equivalent. When predicting performance for the out-of-order wide-issue S-Cores from M-Core measurements, we assume the average ILP on those cores over the entire benchmark suite.

5.2.2 Memory

Performance on both S-Core and M-Core architectures is directly limited by the frequency of data accesses, the latency of those data requests, and by how regular the data accesses

are. For the S-Core, the amount of data required and the regularity of the accesses to that data impacts the frequency of cache misses and the number of SIMD data accesses. For the M-Core, the caches (if any) tend to be less useful, and the memory footprint plus whether or not accesses are regular impact the number of off-chip memory accesses and whether or not accesses from different threads can be coalesced into a single access.

Number of Data Accesses: We find the number of data accesses directly from the number of data accesses in the source implementation. This assumes that register pressure on both the source and target architecture does not force extra data loads or stores.

Number of Off-Chip Memory Accesses (Source): The number of cache misses and off-chip accesses is a sensitive and difficult to predict input. We split our description into two parts: how we collect data on the source machine and how we translate that data to use as model inputs for the target machine. For S-Core source machines, we use a Pin-based MICA tool to collect the data footprint size and data access stride information [52]. For M-Core source machines, we lack such a tool, and instead determine footprint and stride information empirically. For the footprint size, we observe that optimized CUDA programs will place any shared data in the on-chip memory, and most off-chip accesses will be to unique addresses. We find a lower bound on the number of off-chip memory accesses using the memory footprint on the M-core and the on-S-core memory capacity: First, we find the number of off-chip memory accesses for the M-core ($F_{off-chip}$). Then, using $F_{off-chip}$ and the on-chip capacity ($C_{on-chip}$, in B), we find the size of the S-core footprint (F_{CPU}, in B):

$$F_{CPU} = F_{off-chip} + C_{on-chip} \tag{5.1}$$

Number of Off-Chip Memory Accesses (Target): We now describe how we translate from the footprint information to memory inputs for both S-Cores and M-Cores.

For S-Cores, we assume that all cache misses are due to limited cache capacity; this is reasonable since all of the workloads that we consider are data intensive and have minimal sharing. For each level of cache, we find how much of the memory footprint will fit in the cache. Any part of the footprint that does not fit into a level of the cache is assumed to miss in the cache; we assume only one cache miss per cache line. Although this does not model

the impact of prefetching, cache contention, or other second-order effects, we find that it is reasonable for workloads with primarily streaming data-use.

For M-Cores, we find a lower bound on the number of off-chip memory accesses using the memory footprint on the S-Core, the on-M-core memory capacity, and the maximum M-core memory transaction size:

First, we find the amount of required data that does not fit on the M-core ($F_{off-chip}$) using the size of the S-Core footprint (F_{CPU}, in B) and the on-chip capacity ($C_{on-chip}$, in B):

$$F_{off-chip} = \max(F_{CPU} - C_{on-chip}, 0) \tag{5.2}$$

For simplicity, here we assume that the on-chip capacity is the amount of shared memory per SM times the number of active SMs (64 KB/SM $\times$ 4 SMs). The approach could be modified for newer M-cores with on-chip caches; we ignore the impact of constant memory since the constant memory tends to be small and is read-only.

Then, the number of memory instructions per warp is the amount of off-chip memory divided by the memory access size (b, in Bytes) and the number of warps (N_{warps}):

$$\text{Memory Instructions Per Warp} = \frac{F_{off-chip}}{b \times N_{warps}} \tag{5.3}$$

We may be able to use the inner loop footprint and memory access strides to get improved predictions; this is future work.

Types of Accesses: We use data access stride information to predict how S-Cores and M-Cores can group data accesses for faster memory performance. Since SSE does not have gather/scatter capabilities, we assume that only benchmarks with completely coalesced M-Core accesses (e.g., sequential) are SIMD loads/stores. The percentage of S-core loads that are SIMD are the same as the percentage of M-core memory accesses that are memory coalesced. For the number of transactions per warp for the M-core (a measure of warp coalescing), we observe that, since SSE does not have gather/scatter capabilities, any SSE load is continuous in memory. Therefore, the percentage of S-Core loads that are SIMD are representative of how often the M-core data is laid out appropriately for memory coalescing. When SIMD-optimized code is not available, we use stride length information from the S-core execution and assume that only accesses with stride length equal to one are coalesced.

Note that regular stride patterns may indicate the possibility that algorithm re-organization could be used to increase the memory access coalescing rate; this is an area for future work.

5.2.3 Control

So far, we have assumed that the overall program structure does not change between code written for S-Cores and M-Cores. However, this is not true: S-Core kernels tend to be written as a single outer-loop that is executed many times, while M-Core kernels are written as, effectively, the interior of that loop, and launched by many threads. In the most direct translations, the number of threads in the M-Core implementation is equal to the number of outer-loop iterations in the S-Core implementation. In this section, we describe how we find the impact of this translation. These impacts include the computation's organization, branching effects, and any necessary synchronization.

As part of our commitment to using data only from the real hardware and either binary instrumentation or emulation, we primarily leverage a dynamic trace of the basic blocks executed and, in the case of the M-Core, the per-warp active threads masks associated with that basic block trace.

S-Core Outer-Loops: When translating from the M-core to the S-core, we must form the SIMD instructions and compute the number of loop iterations which must occur. Our model records the thread mask for each basic block, and breaks it into SIMD chunks (size 4 for SSE, size 8 for AVX). The number of outer loops, then, is the number of threads divided by the SIMD width. Since S-core code usually uses SSE instructions to implement masks and blend instructions to deal with control flow divergence, all diverged instructions are executed serially; the number of S-Core instructions may be under-counted due to mask management.

M-Core Threads: When translating from the S-Core to the M-core, we must form the threads and warps which perform synchronous independent computation. Our model combines sets of eight S-Core outer-loop executions into a single warp, reflecting the difference in vector to warp size. The number of warps, then, is the number of remaining outer-loop iterations on the S-Core. Since S-Core code usually uses masks and blend instructions to deal with control flow divergence, all diverged instructions are executed serially; this matches

with M-core execution of diverged code. If a warp does not experience control flow divergence, this may lead to an over-approximation of the number of instructions on the M-core. A challenge in this process was that the inner loop does not always represent the smallest granularity of work: some benchmarks, like `saxpy`, are unrolled by the programmer, even in the scalar version. The scalar version of `saxpy` actually works on 4 elements at a time, so everything is off by a factor of four and we miss how many threads could actually be worked on. This is actually a general problem: any loop unrolling will confuse the instruction counting process. Rather than modifying the code, take the number of unrolled bits per inner loop as input; even if such loop unrolling does is not done explicitly in the code, it may also occur at the compiler level. We could guide the SIMD-loop warp formation process by including the number of data elements (or maximum number of inner loop iterations) as an input.

Warp Synchronization: To model the required warp synchronization, we insert a sync operation at the end of any inner-loops observed in the code. This approach is conservative, but we found that most benchmarks required this because of shared memory use.

Optimization We could attempt to find the optimal number of blocks given the total number of threads and hardware capabilities, dividing into a number of blocks that will keep all SMs busy. We discuss the need for such a search in the next section.

In this section, we described how input translation works. In the next section, we will describe how the translation impacts our inputs; the following section describes the impact of those inputs on performance predictions.

5.3 Input Translation Accuracy

In this section, we present the model validation results. Model validation, now, includes three components: (1) the custom model, (2) the translated inputs, and (3) the custom model with the translated inputs. Since custom model accuracy was discussed at length in Chapter 4, in this section we only consider the accuracy of the translated inputs and of the custom models with the translated inputs.

Table 5.3: Errors in Translated M-Core Control Flow Inputs (N: Native, T: Translated)

	Warps		Blocks		# Syncs / Warp	
	N	T	N	T	N	T
raycasting	4096	76038	512	9505	1	0
saxpy	32000	8000	4000	1000	0	0
lbm	40000	31250	10000	3906	0	0
spmv	1852	1400	463	350	2	16
fft	2048	3072	64	768	0	0
convolution	8176	8176	1022	1022	0	0
sgemm	512	1040384	128	130048	128	0
montecarlo	8000	8000	1000	1000	0	12

(a) Control Flow

	# Insts/Warp		# SFU / Warp	
	N	T	N	T
raycasting	489	160	80	0
saxpy	17	20	3	4
lbm	554	336	106	81
spmv	304	54	87	18
fft	173	31	5	4
convolution	95	103	60	25
sgemm	18035	27	88	4
montecarlo	83	228	70	98

(b) Computation

	# Mem / Warp		Rate Coalesced	
	N	T	N	T
raycasting	3	0	1.0	0.0
saxpy	2	2	1.0	0.0
lbm	20	1	0.0	0.5
spmv	1	18	0.0	0.0
fft	2	1	0.0	0.3
convolution	5	2	0.0	0.3
sgemm	608	3	0.0	0.0
montecarlo	6	6	1.0	0.4

(c) Memory

5.3.1 S-Core to M-Core

We first consider the accuracy as we use S-Core inputs to predict M-Core performance. Note that since the inputs that are translated are platform independent, we do not need to consider a specific S-Core as our starting point.

In this section, we evaluate the accuracy of the input translation process, specifically from a Sandybridge core to a pre-Fermi M-Core. We classify the input translation process using the same categories used throughout this chapter: control flow, computation, and memory.

Table 5.3 shows the key native and translated inputs for the pre-Fermi M-Core; the Kepler M-Core has additional inputs for instruction level parallelism and caching effects. Below, we discuss key discrepancies in the translation process and why they occur. In the next section, we show how these inputs impact speedup predictions.

Control flow: In Table 5.3a, we see the measured and predicted number of warps, blocks, and synchronization instructions. We immediately note that we correctly predict the number of warps in the M-Core program in only two cases. For `sgemm`, we observe that the programmer chose to use the number of outer loops, rather than inner loops, as the number of warps. For `saxpy`, `sgemm`, and `spmv`, the optimized S-Core code includes loop unrolling; four values are computer per iteration instead of just one. For example, for `saxpy`, our prediction of 8,000 warps, instead of 32,000, thus makes sense. Given the large impact that the warp and block structure can have on M-Core performance, we expect these errors to have a large impact on our performance predictions in the next chapter.

Computation: In Table 5.3b, we observe that the predicted number of instructions per warp appears to be incorrect in most cases. However, we note that for `convolution`, a benchmark where the number of warps was predicted within 1% of the actual value, the predicted number of instructions is also within 8% of the measured number. After correcting for loop unrolling and other factors that impact the number of warps, we find that our instruction count predictions are also significantly improved. For five out of the eight benchmarks, the number of SFU instructions is predicted within 50% of the measured value. Additional accuracy would be achieved by recognizing code patterns on the S-Core that correspond to M-Core SFU instructions.

Memory: In Table 5.3c, we observe that the number of memory instructions per warp is, like the total number of instructions per warp, strongly influenced by the number of warps. Our approach for estimating how often memory accesses are coalesced is conservative, and tends to under-estimate how much co-coalescing can occur.

We have not discussed inputs that are used only in the Kepler model, the cache miss rate and the amount of instruction level parallelism. We will briefly discuss the impact of those inputs when we show the accuracy of our model on the Kepler architecture. We next discuss the accuracy of our translation approach when translating from M-Core inputs to S-Core inputs, and then show the speedup prediction accuracy after each translation approach.

5.3.2 M-Core to S-Core

We first consider the accuracy as we use M-Core inputs to predict S-Core performance. Note that since the inputs that are translated are platform independent, we do not need to consider a specific M-Core as our starting point (although we have found a pre-Fermi

Table 5.4: Errors in Translated S-Core Control Flow Inputs (N: Native, T: Translated)

	# Branch Misses	
	N	T
raycasting	7,274	29,612
saxpy	8	23,034
lbm	14,155	14,114
spmv	3,959	21,091
fft	1,279	242
convolution	547	27,498
sgemm	66,082	133,015

(a) Control Flow

	# Instructions		ILP	
	N×10^6	T×10^6	N	T
raycasting	33	7.7	3.3	6.2
saxpy	2.6	2.8	6.1	6.2
lbm	238	108	7.1	6.2
spmv	1.0	6.3	7.0	6.2
fft	1.9	1.1	7.7	6.2
convolution	10.4	17	6.8	6.2
sgemm	173	153	6.9	6.2

(b) Computation

	# L1 Data Misses		# L2 Misses		# LLC Misses	
	N×10^3	T×10^3	N×10^3	T×10^3	N	T
raycasting	0	12	0	2	0	0
saxpy	130	92	182	12	59,088	0
lbm	11,427	3,116	15	390	9,826	2,620,000
spmv	55	65	57	9	16,048	0
fft	51	56	3	8	0	0
convolution	24	37	2	5	2	0
sgemm	8,800	617	5,388	78	271	122,592

(c) Memory

architecture to be an easier starting point since its memory hierarchy is simpler). Results from `montecarlo` are not included as not all Ocelot profiling completed correctly.

In this section, we evaluate the accuracy of the input translation process, specifically from a pre-Fermi M-Core to a Sandybridge core. We classify the input translation process using the same categories used throughout this chapter: control flow, computation, and memory.

Control flow: In Table 5.4a, we show the natively measured and translation predicted number of branch mispredictions. For branch misses, we assume that 3% of all branches are mispredicted. This is almost exactly true for the `lbm` benchmark; for other benchmarks, it is an over-prediction because these programs are very regular.

Computation: In Table 5.4b, we show the natively measured and translation predicted instruction counts and ILP. First, for instructions, we not that we are within a factor of

two for all but one instruction count predictions. Our assumption that ILP is always 6.2 is actually reasonable for most benchmarks; only `raycasting` has a significantly different ILP.

Memory: In Table 5.4c, we show our predicted cache miss rates. For the L1 cache miss rate, our predicted cache miss rate is qualitatively close the to measured miss rate for all benchmarks except for `lbm` and `sgemm`. Interestingly, for these benchmarks, we over-predict the number of LLC misses, and for all other benchmarks, we find that there are no LLC caches.

5.4 Translated Core Accuracy

In this section, we find the impact of the translated inputs on our model predictions. We consider translation both from S-Core inputs to M-Core inputs and vice versa.

As discussed at the beginning of this chapter, we have the following goals:

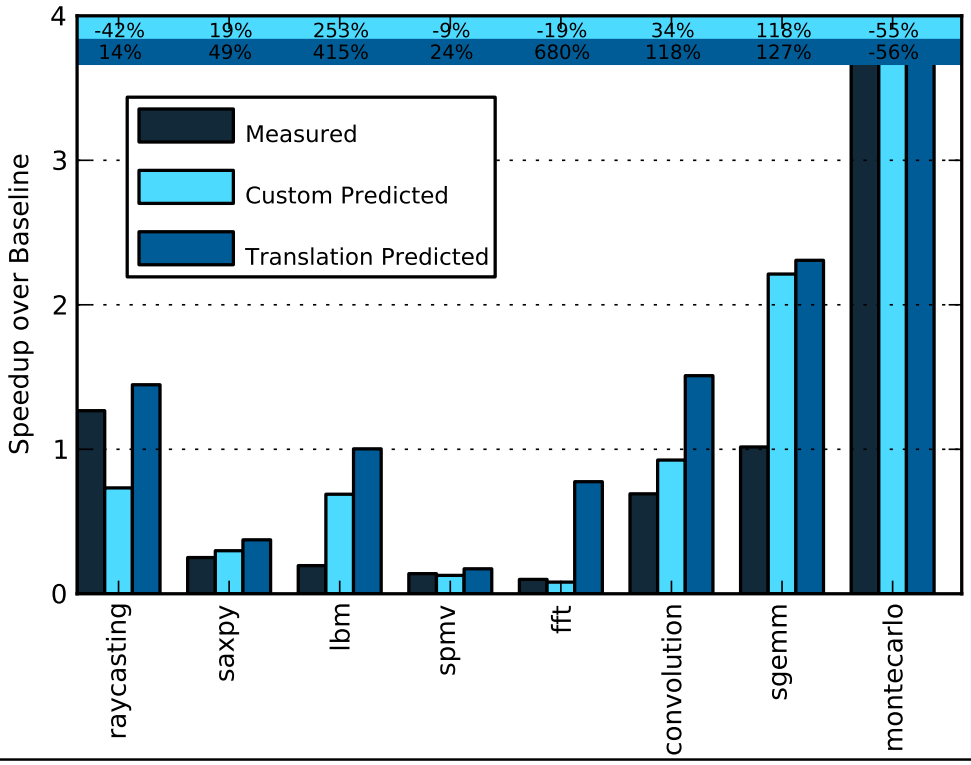
- **Trends:** Does the translation model correctly predict when an architecture completes a benchmark more quickly than another architecture?
- **Accuracy:** How close are projected speedups to the measured speedup?

This section evaluates each of those goals for both input translation and speedup predictions for the S-Core to M-Core and the M-Core to S-Core cross-accelerator performance predictions using the CAB benchmark suite. Additional results for the CAB benchmark suite are in Appendix F; results using the Rodinia benchmark suite are in Appendix G.

5.4.1 S-Core to M-Core

In this section, we evaluate the accuracy of the input translation process, specifically from a Sandybridge core to a pre-Fermi M-Core. In addition to speedup graphs, we provide the accuracy range and average, the input-based error, and limitations below each figure. Input-based error, here, is calculated as the error in the translated speedup prediction that is beyond the error in the custom speedup prediction from the last chapter. This is the additional error introduced due to the errors in our inputs.

In the input translation validation section above, we found two key sources of error in the inputs: loop-unrolling on the S-Code, and warp-block configurations that did not match

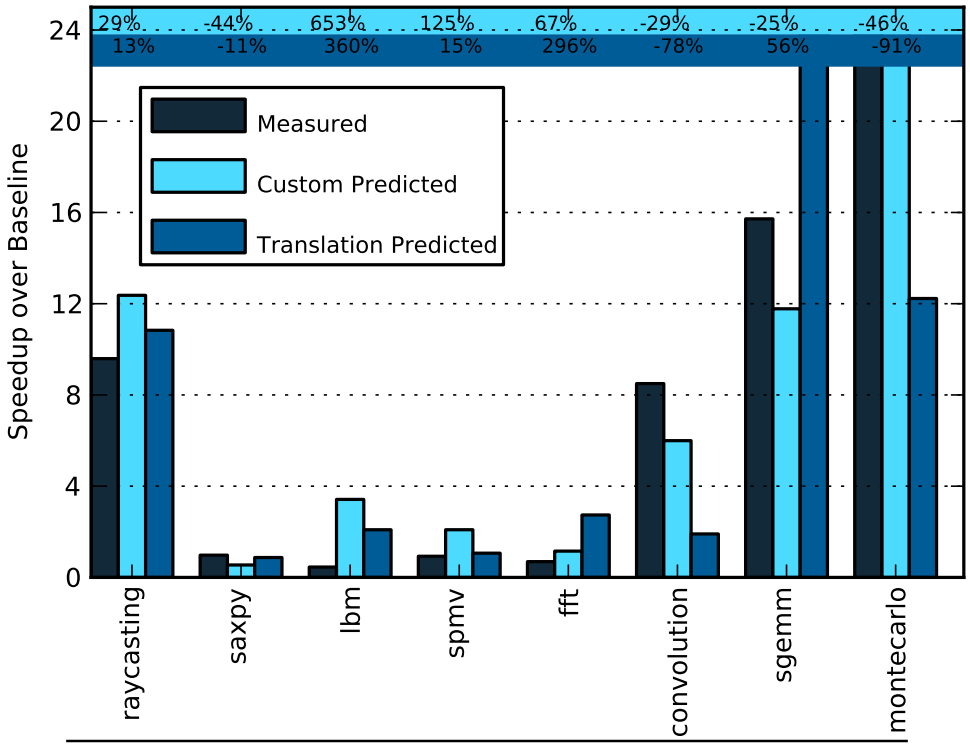


Accuracy	-56% to 680%, average 171%
Input-based Error:	0% to 671%, average 122%
Limitations	Model error, memory coalescing predictions, # of SFU instructions

Figure 5.2: *S*-Core to *M*-Core Translation: pre-Fermi

our predictions. Since loop-unrolling can be easily detected, and warp-block configurations can either be given by the programmer or found through a search process using our model, we assume below that these error sources have been fixed.

In Figure 5.2, we show the predicted speedups using the *S*-Core to *M*-Core translation approach for the pre-Fermi core, along with the measured speedup and the predicted speedup using the custom model and native inputs. In this initial approach, errors range from -56% to 680%. As shown in the table, we find that although some errors can be attributed to input errors, on average, the errors in our inputs contribute significantly to speedup prediction errors. We note that `convolution`, in particular, is impacted by the number of SFU instructions that we predict (25) versus the number that are actually executed (60). Correcting this input reduces the error in our predicted speedups for `convolution`



Accuracy	-91% to 360%, average 70%
Input-based Error:	0% to 229%, average 44%
Limitations	Model error, # of SFU instructions

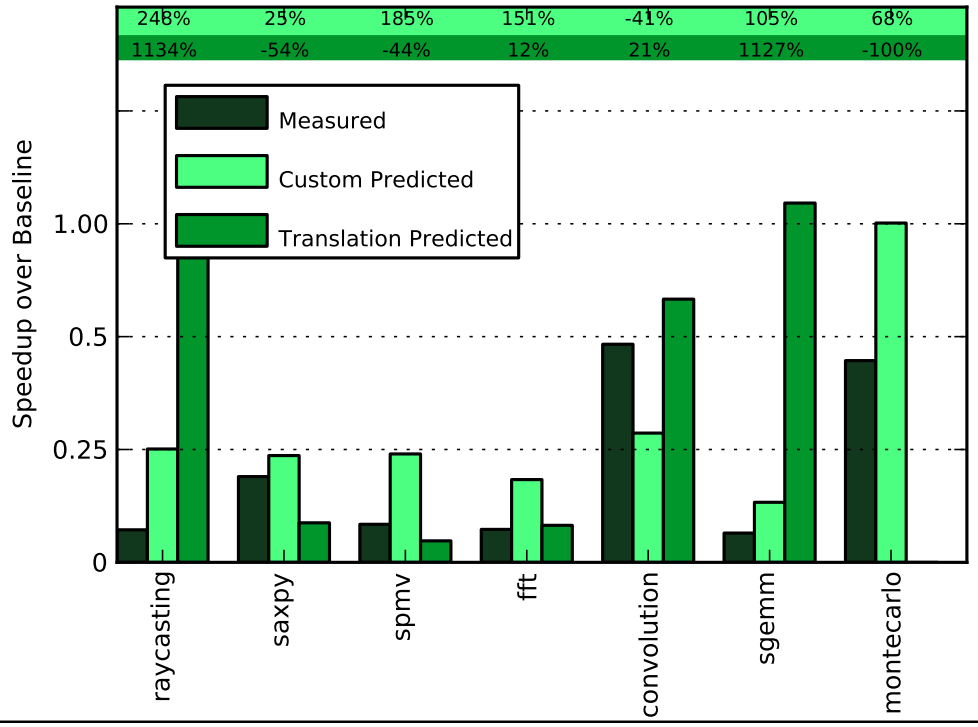
Figure 5.3: *S*-Core to *M*-Core Translation: Kepler

significantly.

In Figure 5.3, we show the predicted speedups when we use Sandybridge measurements to predict performance on a Kepler SM. We find improved accuracy results compared to the translation to the Pre-Fermi SM. We believe this is related to the improved memory behavior on the Kepler *M*-Core.

5.4.2 *M*-Core to *S*-Core

In this section, we evaluate the accuracy of the input translation process, specifically from a pre-Fermi *M*-core to an Atom core and a Sandybridge core. Note that due to the *S*-Core's more advanced single-thread optimizations, we expect accuracy to be lower than when we used cross-accelerator prediction in the other direction.



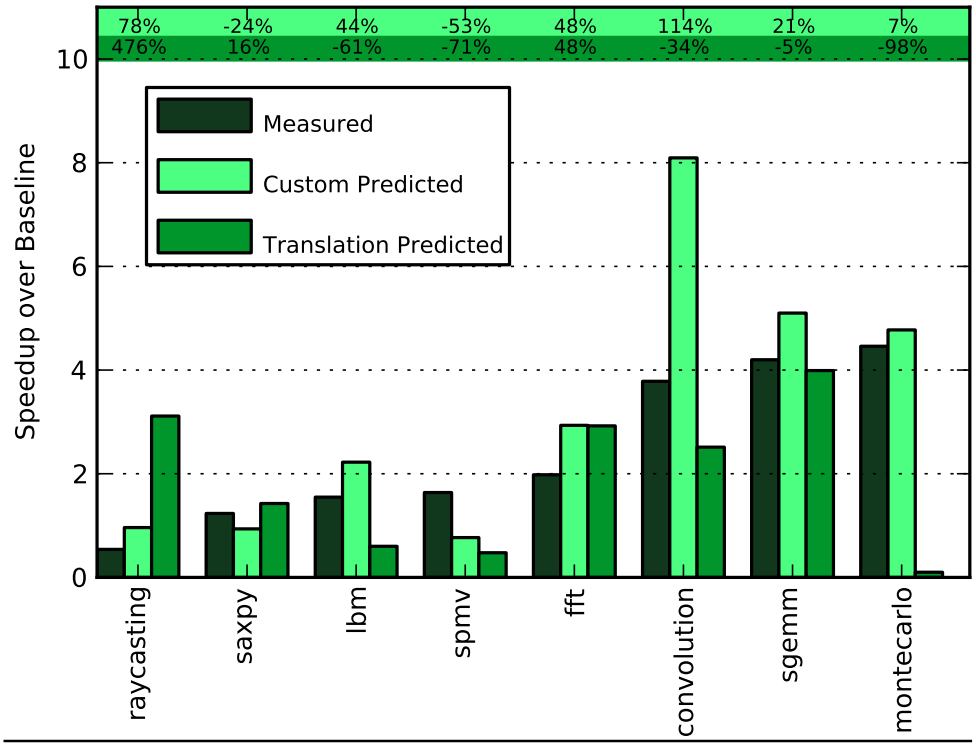
Accuracy	-44% to 1134%, average 313%
Input-based Error:	0% to 1134%, average excluding outliers 33%
Limitations	Model error, algorithmic changes, memory contention

Figure 5.4: M-Core to S-Core: Atom

In Figure 5.4 and 5.4.2, we show the predicted speedups using the M-Core to S-Core translation approach, along with the measured speedup and the predicted speedup using the custom model and native inputs. Note that `montecarlo` translation failed due to an Ocelot error and is not included.

In this initial approach, errors range from -71% to 476%. We correctly predict whether or not a benchmark is sped up using the SIMD unit on the Sandybridge in all but two cases. Of particular note is the `raycasting` benchmark, which has high speedup prediction errors for both the Atom and the Sandybridge cores. We note that the M-Core implementation has algorithmic changes compared to the S-Core implementation which our approach cannot predict.

On the Sandybridge, the errors in the inputs are small enough such that we have reasonable accuracy, are mostly optimistic about speedups, and predict trends correctly in most



Accuracy	-71% to 476%, average 52%
Input-based Error:	0% to 398%, average excluding outliers 7.5%
Limitations	Model error, algorithmic changes

Figure 5.5: M-Core to S-Core: Sandybridge

cases. Atom predictions are reasonable, but we are further limited by the Atom's memory constraints, which lead to cache misses that our simple capacity based model cannot predict.

5.5 Application to Dark Silicon Projections

In this section, we consider the impact of using these models on dark silicon projections for future technology nodes, continuing our running example from the last two chapters. The framework for this running example was given in Chapter 2, and results using the previous two models are toward the end of Chapter 4 for reference.

In this section, our goal is to show that the projections, even with significantly less accurate inputs, are still useful. This motivates the idea that cross-accelerator projections using the approach in this chapter, even for more diverse accelerators, could be used to

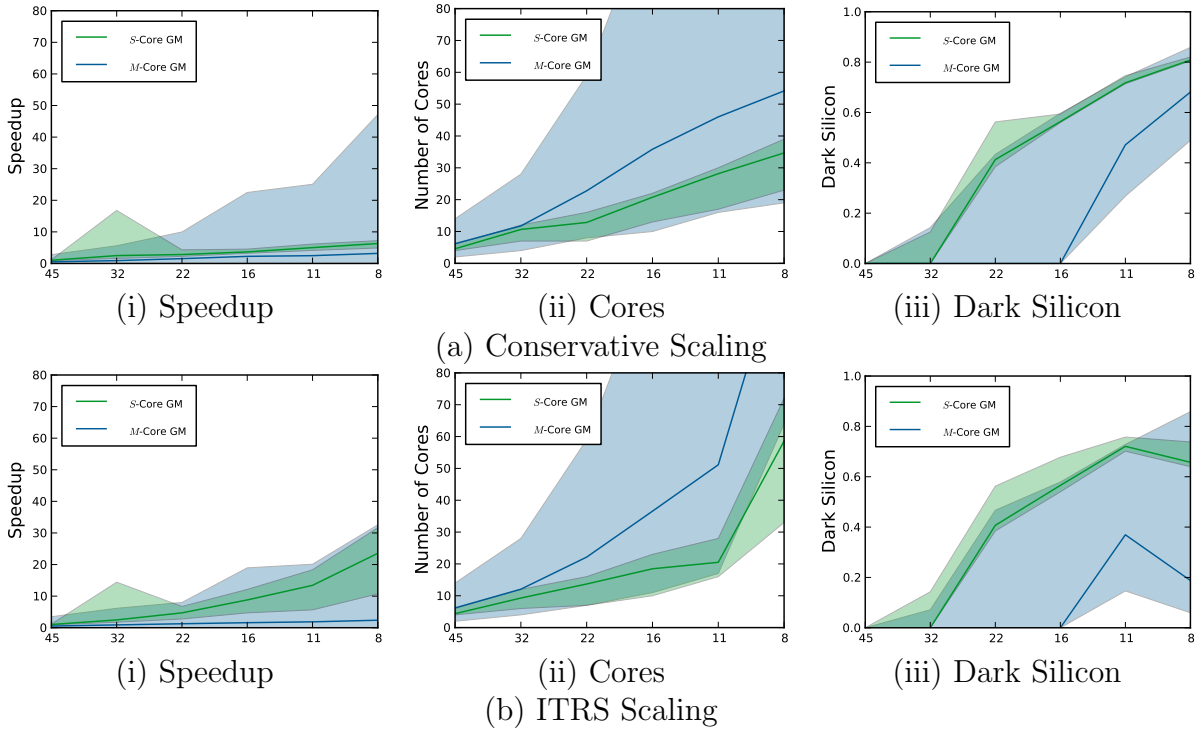


Figure 5.6: Dark Silicon Projections using Custom Models.

predict the impact on future technology generations.

In Figure 5.6, we show the dark silicon projections when inputs have significantly more error. Rather than using the translation model to generate these inputs, we add a random amount of error between -20% to +20% to each term. This approach is conservative - in many cases the errors added by the translation process are less than 20%. Using this approach, we find that at 8nm, the geometric mean speedup for S-Cores is between 3.2× and 22× and for M-Cores is between 2.4× and 6.3×. The number of cores ranges from 35 to 119, and the amount of dark silicon ranges from 19% to 81% at 8nm, depending on scaling assumptions and the type of core used.

These results, on average, are almost identical to the results in Chapter 4. The key difference is that at each point, there may be higher or lower speedups due to the random error. On average, however, the results remain the same. Note that if we had observed biased errors in the translation process (e.g., translation systematically led to speedup over-predictions), these systemic errors would also be introduced into the speedup projections.

The utility of the translation model in the dark silicon projection framework is that we

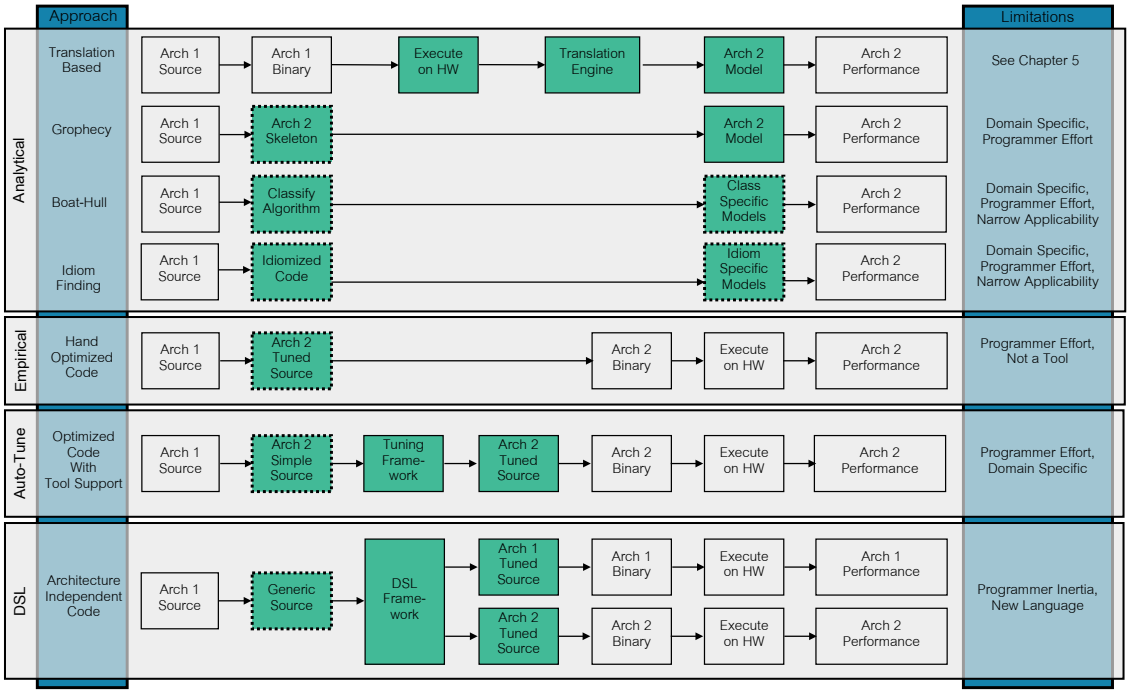


Figure 5.7: CAB overview contrasted with related work. Shaded regions show novel components and dotted borders imply programmer effort.

can now project speedups for cores without implementing them (or the software for them). Given the CAB benchmark suite implemented for S-Cores, for example, we could predict whether the speedups at 8nm for M-Cores would justify rewriting the benchmark suite for M-Cores.

5.6 Related Work

In this section, we consider other approaches for finding performance of a target architecture given code written for the source architecture. These approaches are summarized in Figure 5.7, and grouped into four categories: analytical, empirical, auto-tuning, and DSL-based approaches.

In all cases, the goal is to start with source-code implemented for one accelerator (CPU) and develop intuition on performance on another accelerator (GPU). The limitations of the prior work is also outlined in the figure. The Upper-Bound model in Chapter 3 and related work from Guz et al. [42] are both related to the translation model in this Chapter, but

those abstract models for measuring architecture tradeoffs do not include microarchitecture characteristics and have limited accuracy.

We summarize the work in each of the four areas below.

5.6.1 Analytic Approaches

Several analytic approaches have been previously proposed; we specifically consider a statistical translation-based approach, Grophecy, Boat-Hull, and Idiom-Finding, all described below.

The approach most related to our work is a statistical approach to translate between CPU and GPU performance using key metrics. Such statistical approaches include [59], who find that collecting metrics on four GPUs and three CPUs can predict performance on other GPUs or CPUs, although for reasonable accuracy, they can only predict performance within the same class (GPU or CPU).

Grophecy [74] is a recently proposed novel framework with similar goals as ours. However, they require programmers to skeletonize their CPU code, in a way that inputs can be extracted for a GPU analytical model and then project performance. These code skeletons require heavy programmer involvement, unlike our push-button approach. Further, their approach cannot predict performance for CPUs with SIMD acceleration.

The Boat-Hull approach [78] is an even more recently proposed approach. This work predicts performance on either a CPU or GPU based on both the hardware characteristics and software inputs. However, the software inputs include classification of the benchmark into a specific class; each class of benchmarks then has a specific analytic model to produce performance estimates. This approach produces useful predictions, but again requires programmer effort.

Finally, the idiom-finding approach [18, 75] presents one of the first works in this area of cross-accelerator tools. Their tool focuses on program idioms (specifically scatter/gather) and develops models to project their performance if ported to FPGAs or GPUs. Our work looks at full applications which are are well beyond the scope of idioms and also look at translation both from CPUs to GPUs and vice versa.

5.6.2 Empirical Approaches

Empirical studies have been performed porting suites of applications on CPU and GPUs to answer questions on portability [68, 77, 41].

Of particular note among the empirical works is [87], which argues that established compiler technology and pragmas are sufficient to effectively utilize throughput accelerators and only "low programming effort" is necessary. Their study is an empirical result for a suite of benchmarks and hence of limited use for new workloads and application developers. Maleki's [73] controlled and rigorous study of vectorizing compilers makes the opposite claim: " only a few loops from the real applications are vectorized by the compilers (GCC (version 4.7.0), ICC (version 12.0) and XLC (version 11.01) we evaluated." We argue that given the programmer expertise necessary to insert useful pragmas and to verify that all useful parallelism has been extracted, auto-generated approaches for empirical performance comparisons are still labor intensive.

5.6.3 Auto-Tuning

Mainstream techniques include a plethora of code profiling and analysis tools like gprof, VTune, valgrind, which allow detailed analysis of code written for one architecture to be further tuned for it. A body of literature has focused on analyzing sequential code to inform programmers on multi-core parallelization strategies [37, 63, 2, 95]. Without programmer involvement, the tools provide the required information to programmers. As with the empirical approaches above, heavy programmer involvement is still required, and many of the tools are domain specific.

Somewhat similar to the auto-tuning approach, many recent works provide frameworks for automatically or semi-automatically generating efficient GPU code. Examples include PGI compiler [100], OpenMPC [67], Mint [97], and C-to-CUDAs [8]; compared to our focus, these works are meant for one accelerator require pragmas inserted by programmer, and are typically restricted to one domain of problems, e.g., stencil computation, affine transformation. Other works present a performance analysis framework for suggesting further optimizations for GPU given CUDA code as input [89, 6].

5.6.4 DSL

Finally, we consider architecture independent code that can be compiled for multiple platforms. Specifically, we consider OpenCL, which can be compiled for both CPUs and GPUs. However, OpenCL tools are still evolving and have not yet achieved the same performance as native code [35].

In summary, though good tools exist for understanding performance bottlenecks within a particular domain, which could serve as the underpinnings of a more general approach, *a high-fidelity cross-accelerator performance projection/profiling programmer-uninvolved tool has remained elusive.*

5.7 Summary and Open Questions

In this chapter, we considered the use of inputs measured on one architecture to predict the performance of a benchmark on another architecture. This approach is extremely powerful as it allows both architects and programmers to explore new designs without completely re-writing their programs (and compilers, simulators, etc.).

Although accuracy is not as high as with custom models, we feel that these results are useful as a first-order tool while developing new accelerators. The key value of this tool is in its application to future accelerators that have not yet been designed. We discuss the tool's application to a more sophisticated accelerator with functionality specialization in our conclusions, and observe that increasing functionality specialization leads to simpler, easier to predict benchmarks.

6 CONCLUSIONS AND FUTURE WORK

In this section, we conclude the dissertation with a summary of contributions and a discussion of future work.

6.1 Summary of Contributions

We began this dissertation with a summary cartoon showing the architectural trade-offs and anticipated accuracy for our three models in relation to previous work. In Figure 6.1, we revisit this trade-off figure with real data from the dissertation.

As a metric for architectural flexibility, we count the degrees of freedom for each model. To give weight to the difficult problem of processor-threading, we give the ability to model both single-threaded and many-threaded architectures with a single model a weight of ten. Other characteristics, like cache miss rate, core frequency, and issue width, each have a weight of one. As a metric for accuracy, we consider the Average Absolute Error across a benchmark suite. So that our accuracy ranges from zero (very poor) to one (exactly correct, assuming errors are at least 1%), we plot 1/`Average Absolute Error`.

Using these metrics, we find that our initial cartoon under-estimated the utility of several of each model in different respects. We did not anticipate that the modular microarchitecture extensions, in addition to increasing the microarchitectures that we could model with a single model, would also increase accuracy on previously modeled architectures. Using the measure of architectural flexibility above, the translation model is actually the most flexible modeling approach that we have considered. Finally, the upper-bound core model, although it has limited flexibility, can actually be very accurate for processors that it has detailed information for.

As discussed in the introduction, the key contributions of this thesis are the following:

- Distillation of mechanistic models to unified upper-bound models that cover a diverse architecture space
- General modular approach to architecture-specific mechanistic models
- New mechanistic model modules to accurately predict performance on real hardware

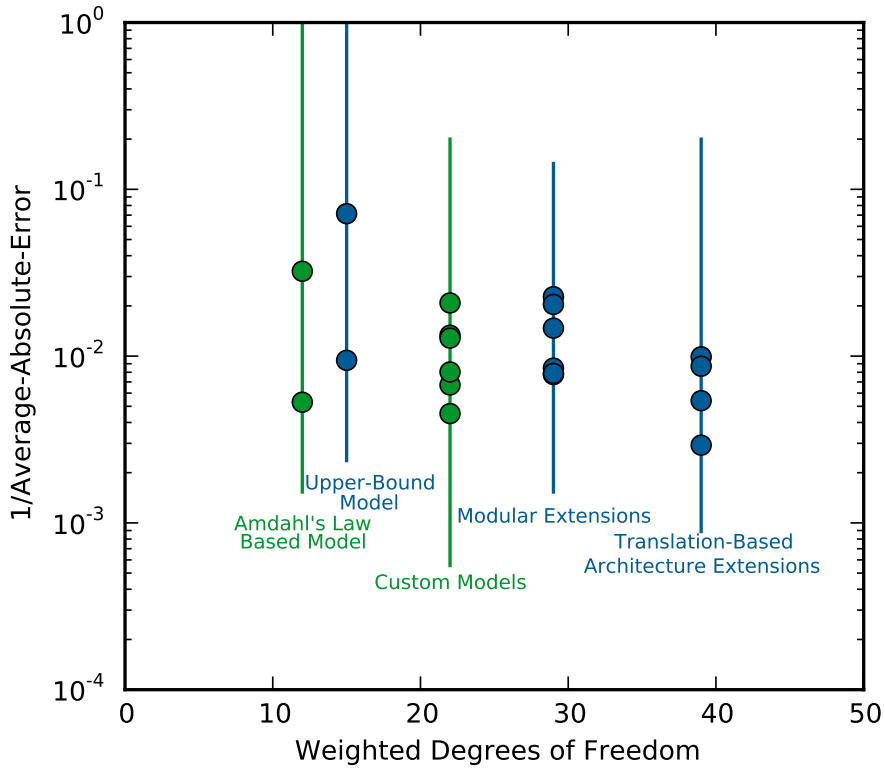


Figure 6.1: Detailed Trade-offs. Green models are from prior work and blue models are from this dissertation. Dots show per-core averages; lines show range of prediction errors.

- Identification of architecture-independent qualities of important performance characteristics
- Cross-architecture performance prediction using architecture-independent measurements and no user intervention
- Application of the three different models to dark silicon projections to improve understanding of future technology challenges

6.2 Future Work

This dissertation leaves several open questions. We consider three, in particular: applying the model to functionality specialization, adding power and energy predictions, and approaches to further improve accuracy. Below, we discuss each of these areas and ideas for future work.

6.2.1 Functionality Specialization

Throughout this dissertation, the emphasis has been on specialization through data-level parallelism with our consideration of M-Cores and S-Cores with SIMD units. We now consider how this work could be applied to architectural improvements that use functionality specialization - approaches with custom hardware targeted to improve application execution. As an example, we briefly consider the Dyser architecture, which support both functionality specialization and data-level parallelism [39].

Before discussing how we would add the Dyser architecture to our infrastructure, we briefly describe the architecture. The Dyser (Dynamically Specializing Execution Resources) architecture consists of a set of heterogeneous functional units, connected via simple switches, which are integrated with a general-purpose architecture. A compiler detects commonly executed regions, and those regions are ported to the Dyser unit. The general purpose processor's standard pipeline is used for all un-ported instructions, as well as for cache, prefetch, and memory functions.

First, we discuss what would need to be done to create a Dyser mechanistic model, and then we discuss what new inputs and translation mechanisms we would need to add to our translation methodology.

For the Dyser mechanistic model, we observe that the Dyser unit is integrated in an S-Core, and so we begin with the S-Core model. By sending regions of code to a specialized unit, the approach effectively replaces code segments with long latency instructions. In Figure 4.3, we included a module for N_{accel}. This module counts the number of accelerator (Dyser) instructions and the number of S-Core instructions that the accelerator instructions replace. The long latency accelerator instructions, then, can be modeled like any other long latency instruction in the S-Core. The latency of these instructions would be based on the longest dependency chain of instructions in the Dyser unit and the latencies of each; this could be further refined using additional information.

In addition to knowing how many instructions are mapped to Dyser, we also need to know how many load-slices are generated. This requires predicting the region of code that is mapped to Dyser and how it is sliced; we can find an upper-bound from the number of backward load-slices and number of loads. Further work would be required for refinements including handling loop unrolling and vectorization optimizations [40]. We assume that memory accesses and cache usage would not change since Dyser fairly directly replaces the

S-Core code.

Finally, we consider how adding the Dyser accelerator would impact our translation approach. Specifically, we consider modeling Dyser performance from M-Core inputs. We observe that, as in the mechanistic model description above, the new component is predicting the number of instructions that would be mapped to the Dyser and how they would be arranged into load-slices. This could be done similarly to as above after mapping the M-Core instructions to an intermediate format, as discussed in Chapter 5.

6.2.2 Improving Accuracy

Throughout this dissertation, we have commented on areas where our estimations could be improved through additional data to improve model accuracy. These comments are summarized below for each model.

Upper-Bound Model: The upper bound model's generality makes it interesting for basic scaling design space exploration. In addition, it would be interesting to generate new Pareto frontiers at current technology nodes, and for both S-Cores and M-Cores. Exploring the M-Core design space through Pareto frontiers could be particularly interesting as M-Cores become more general purpose. Finally, adding more details to the parallelism model used in this work, perhaps by applying a critical sections model [33] could add interesting refinements as to the importance of parallelism and how it is quantified.

Custom Model Extensions: In the results that we presented, a key source of error was the fact that we did not model disk accesses; this played a large part in our errors for the Cortex-A8 and Atom processors. Adding this component would greatly improve our S-Core prediction accuracy. For the M-Cores, we used Fermi timing information in the Kepler model due to availability; refined data would improve results.

Translation-Based Extensions: There is still space for refinements in this approach to improve accuracy. We have included a very simple cache miss model for S-Cores that is based only on the memory footprint that is sufficient for our mostly regular applications, but recognize that a more detailed model based on memory re-use and access stride patterns would be more accurate, especially for more irregular applications. Similarly, our simple

memory access coalescing model could be improved with further memory modeling development. Implementing an optimization pass that considers different layouts for the translated code could also improve accuracy, and partially addresses concerns about simple algorithmic changes. Although we detected them by hand, adding a pass to auto-detect loop unrolling would also improve push-button accuracy.

6.2.3 Power and Energy

Throughout this dissertation, we considered a running example in which we computed the number of cores that could fit on future multicore chips given area and power constraints and scaling factors. In Chapter 3, we considered a model that included resource trade-off curves with the performance achievable given an area or power budget. In Chapters 4 and 5, however, we considered modelling approaches for architectural improvements that could not be readily included in those resource trade-off functions.

Very interesting future work would be to extend the performance models in this dissertation to include power and energy estimates. The most straight-forward approach would be to compute activity factors for these architectures and use them in a modified power estimation tool such as McPAT [69].

6.3 Closing Remarks

In this dissertation, we have three modeling approaches to expand the architectural trade-off and accuracy space. These models, an upper-bound model, custom model extensions, and translation-based architectural extensions, each have specific use-cases and strengths. Using the models, we are able to generate dark silicon projections as a running example of the models' utility and strengths.

The goal throughout this work is to expand the tool-set available to architects for rapid exploration of novel ideas. The strengths of this work lie in the ability to model new ideas without having to implement the hardware or software necessary to use that hardware. Accuracy at this early stage does not need to be within 10%, and as a result the approaches discussed in this dissertation can be useful to architects.

Analytic mechanistic models have a long history in computer architecture research. In the first five years of ISCA proceedings, they were applied to problems as varied as instruction

design to minimize program size [99], pipeline buffering [82], memory hierarchy design [85], understanding memory and CPU bandwidth trade-offs and addressing efficiency [45], multi-core workloads and distributed parallel machines [94, 62], and input/output latency hiding using multi-programmed workloads [96]. At the time, they provided an approach to quantify design impacts when full system simulation was difficult, due to both computation and infrastructure limitations. Today, with the increasing complexity of both hardware and software and the highly optimized stacks that require changes at all levels to implement new ideas, we expect a renaissance of the utility of analytic mechanistic modeling as a tool for computer architects.

The increasing need for more new architectural ideas that do not rely on increasing energy use has forced interest in new ways to rapidly evaluate ideas. The models in this dissertation are one step toward a new set of tools in architects' toolboxes.

A CUSTOM BENCHMARKS

SGEMM: SGEMM is the $C = AB^T$, column-major version of dense matrix multiply. We vectorize the two inner-loops, and perform standard locality, unrolling, and alignment optimizations. The optimized CUDA SGEMM is based on the register-tiling GPU code by Stratton et al. [91], and is one of the fastest known implementations. It uses per-thread register caches for access to A to reduce total shared memory accesses.

Convolution: Convolution is a widely used image filtering technique that has applications in image processing and edge detection. We use a 5x5 filter, applying it to each pixel of a padded image. The CPU version efficiently vectorizes pixel computation, but relies on unaligned loads to fetch contiguous data. The CUDA implementation uses shared memory to load image regions, reducing the required memory bandwidth, and uses the constant memory for the filter.

FFT: Traditional Fast Fourier Transform is a fundamental algorithm in signal processing and differential equation solving. Our SSE implementation interchanges loops of the standard FFT butterfly formula for better temporal locality, enabling efficient, aligned vectorization. The CUDA version uses a similar iteration pattern, first shuffling the data, then performing the butterfly algorithm in separate kernels. Both versions use pre-generated look-up tables for sine and cosine coefficients.

SAXPY: SAXPY (Single-Precision AX Plus Y) performs a scalar multiplication and vector addition, and is a standard function in the Basic Linear Algebra Subroutines (BLAS) library. The SSE vectorization of SAXPY is straight-forward and utilizes software prefetching for marginal performance improvements. The CUDA version is similarly simple, using one thread per element.

SPMV: Sparse matrix vector multiplication is a primitive in sparse linear algebra with an irregular access pattern, making it memory bound. We first implement non-blocked versions for SSE and CUDA, which either perform no vectorization, or attain no memory coalescing. Next, our blocked version for both architectures use the BCSR [53] data structure, which

coarsens the granularity of sparsity, and enables vectorization/coalescing at the cost of some redundant computation.

Histogram: Our histogram bins 32-bit integers into 64K bins. We found no vectorization potential in the histogram computation. The CUDA version of histogram applies a simple parallelization, and uses an atomic memory update.

Monte Carlo: Monte Carlo is a class of stochastic algorithms that rely on repeated random sampling to compute their result, and our version has applications in economics. Different algorithmic phases show varying degrees of parallelism, so we cannot vectorize the entire kernel for SSE. Some optimization is achieved through loop unrolling and loop fission. Likewise, the CUDA version requires somewhat different parallelization strategies for each phase.

LBM: LBM is a computational fluid dynamics benchmark that requires a variety of arithmetic operations and control flow. Our vectorization strategy avoids vector-scalar and scalar-vector operations, but requires vector masking to deal with vector-path based control-flow in the benchmark. The CUDA version of LBM is based on the Parboil [91] version. Since LBM is bandwidth bound, optimizations focus on addressing the layout of the lattice data, and include using a tiled structure-of-arrays that achieves both good coalescing and parallel memory bank usage.

Ray Casting: Ray Casting is a graphics benchmark with significant control flow divergence. For this reason, vectorization of this benchmark requires too many vector mask instructions to be profitable. The CUDA version of this benchmark exploits massive thread level parallelism for increased performance.

B PARSEC CHARACTERISTICS FOR UPPER-BOUND MODEL

Table B.1: PARSEC Characteristics Used in Upper-Bound Model

			m_{L1} Parameters		m_{L2} Parameters	
	f_∞	r_{ls}	α	β	α	β
Blackscholes	0.984	0.325	1.5	100	2.0	8025
Bodytrack	0.75	0.326	3.0	27272	5.0	351302
Canneal	0.865	0.386	1.6	1536	1.6	1652000
Dedup	0.87	0.400	1.6	269	1.6	57810
Facesim	0.86	0.480	2.5	16243	2.5	365727
Ferret	0.95	0.362	1.5	227	2.4	348595
Fluidanimate	0.95	0.423	1.9	710	1.6	337750
Freqmine	0.94	0.495	2.0	1540	2.0	134851
Streamcluster	0.99	0.429	1.5	391	1.4	52934
Swaptions	0.9999	0.442	1.8	1819	2.0	39677
Vips	0.85	0.267	1.5	83	3.0	325143
X264	0.99	0.06	2.0	5224	2.0	84626

C ADDITIONAL m-CORE UPPER-BOUND MODEL RESULTS

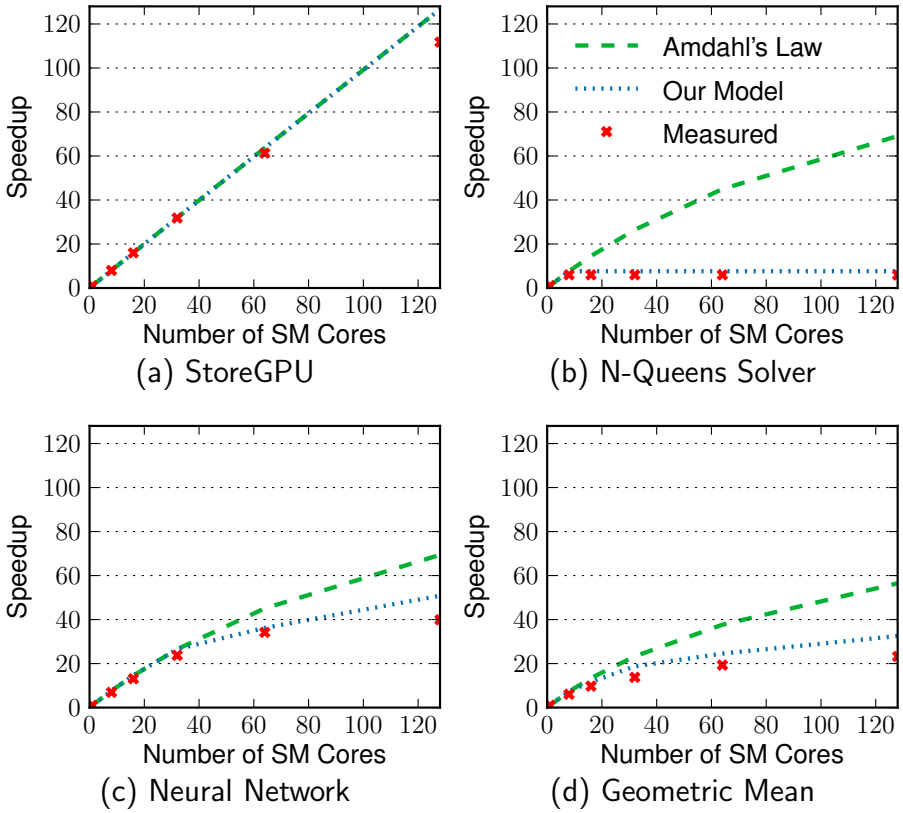
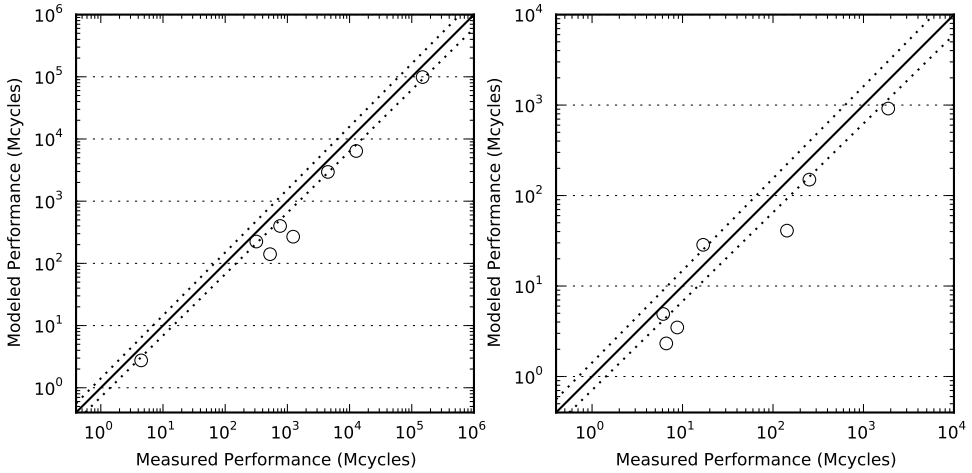


Figure C.1: M-Core Validation: speedup over one SM

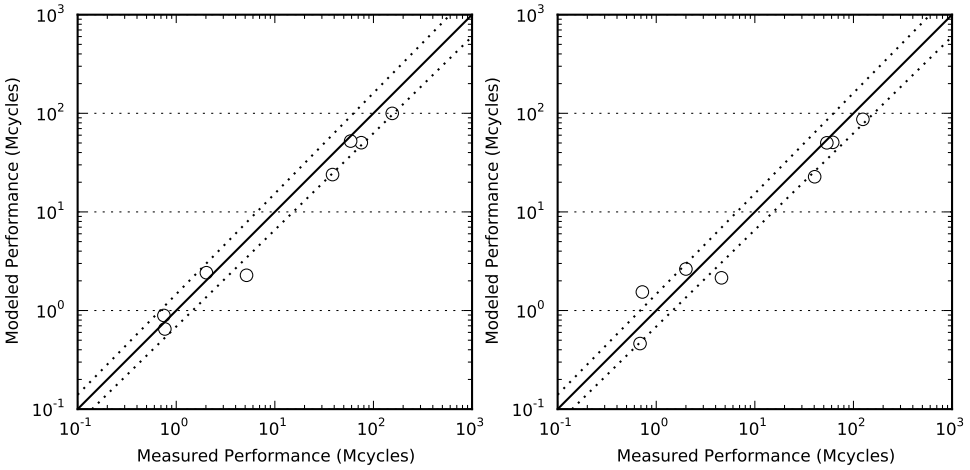
D CYCLE COUNTS USING CUSTOM MODELS

In this appendix, we show the predicted and measured cycle counts for each CAB benchmark on each of the cores considered in Chapter 4. We include a dotted line with ±40% error. Points that fall within the dotted lines thus have accuracy within 40%. Analysis of these results was included in Chapter 4.



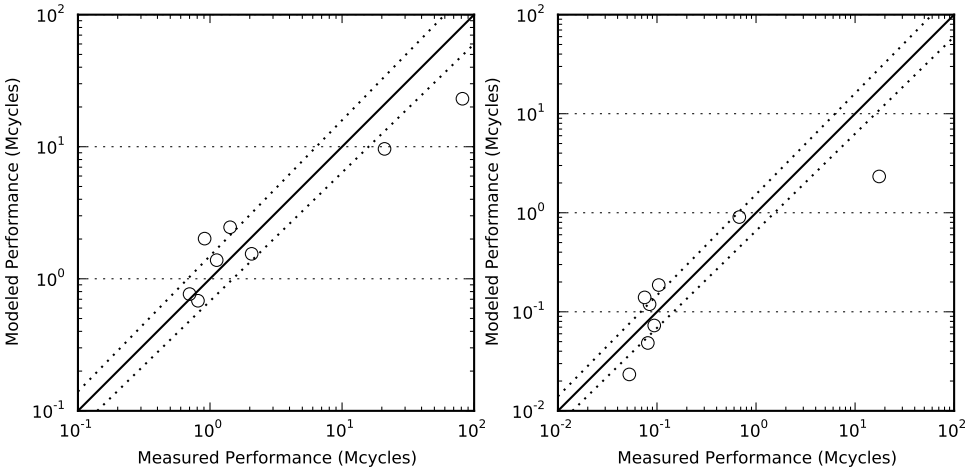
(a) $S1$: Cortex-A8 (b) $S2$: Atom

Figure D.1: In-order S-Cores



(a) $S3$: Xeon (b) $S4+$: Sandybridge

Figure D.2: Out-of-Order S-Cores



(a) $M1$: Pre-Fermi (b) $M3$: Kepler

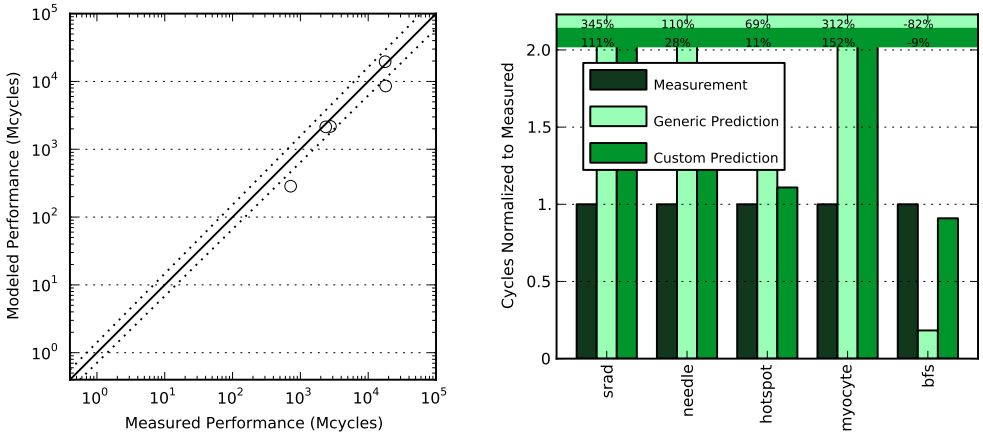
Figure D.3: M-Cores

E CUSTOM MODELS ON CHALLENGE BENCHMARKS

In this appendix, we consider a more challenging benchmark suite, a sub-set of the Rodinia suite. Note that these benchmarks are not optimized for the S or M cores that we consider. Due to some non-ideal behaviors, we do not expect our models to achieve the same accuracy as we did with the CAB benchmarks in Chapter 4. For the Rodinia benchmarks, which were auto-vectorized, most benchmarks do not see a significant speedup. The largest speedup is for `bfs`, which is sped up by nearly six times.

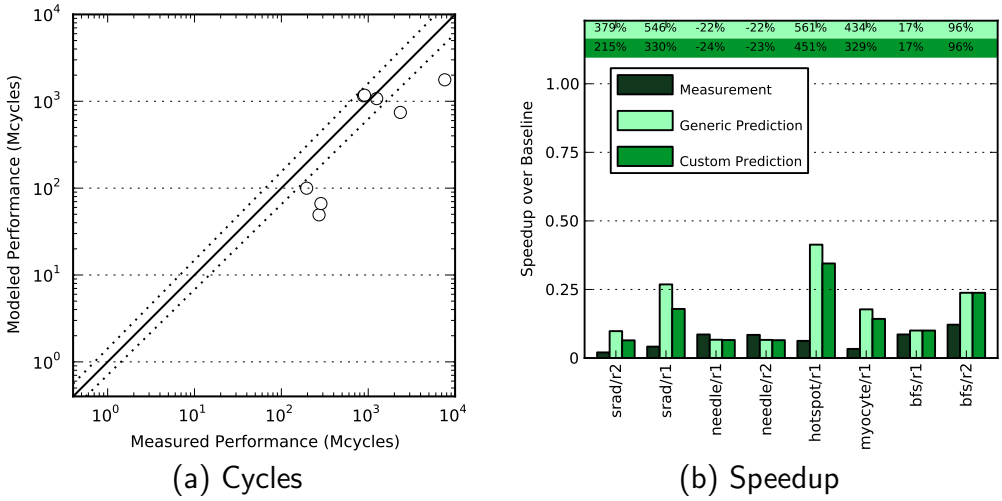
We include both a cycle count graph and a bar graph with speedups and percent errors in speedups. On the cycle count graphs, we include a dotted line with ±40% error. Points that fall within the dotted lines thus have accuracy within 40%. A brief nalysis of these results was included in Chapter 4.

For the Rodinia benchmarks, we include five benchmarks, three of which have multiple kernels. In the figures, "r1" and "r2" refer to different regions, or kernels, in the code.



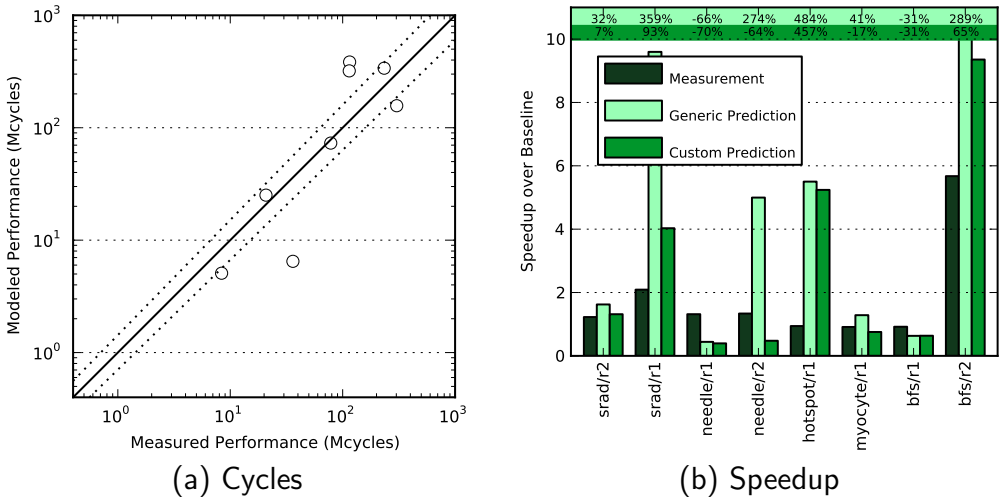
(a) Cycles (b) Speedup

Figure E.1: S1-Core: Cortex-A8



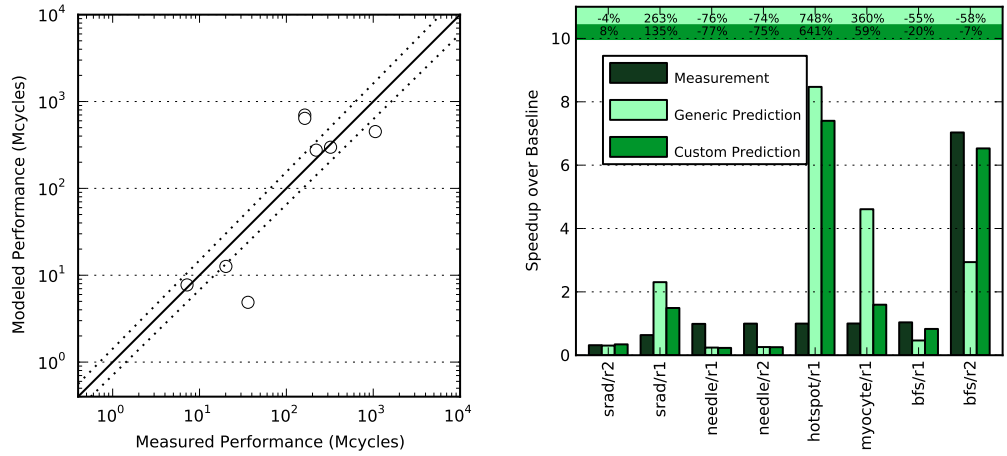
(a) Cycles (b) Speedup

Figure E.2: $S2$-Core: Atom



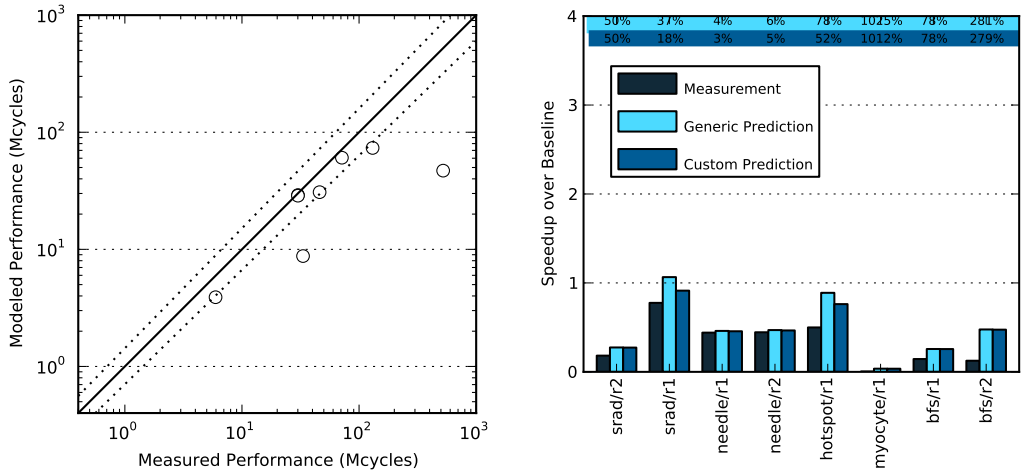
(a) Cycles (b) Speedup

Figure E.3: $S3$-Core: Xeon



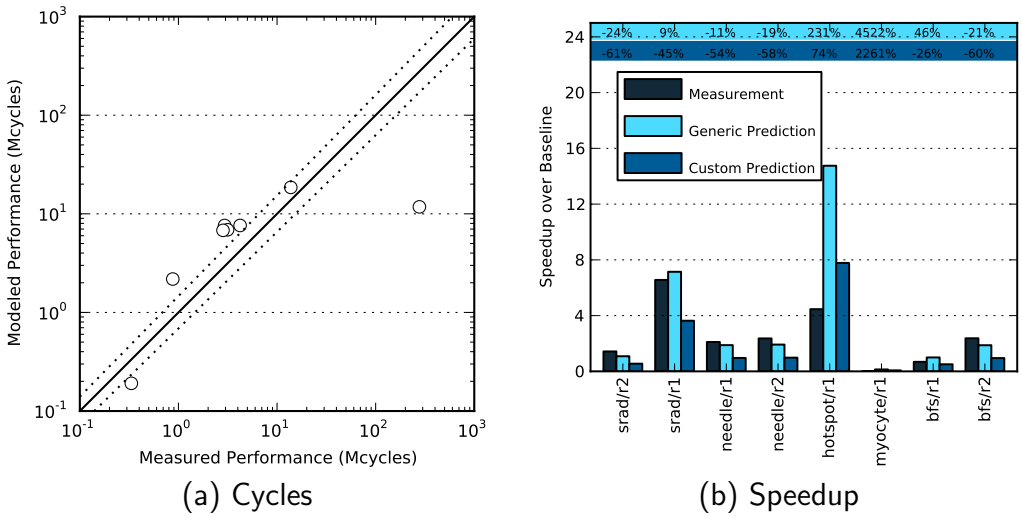
(a) Cycles (b) Speedup

Figure E.4: $S4$+-Core: Sandybridge



(a) Cycles (b) Speedup

Figure E.5: $M1$-Core: pre-Fermi

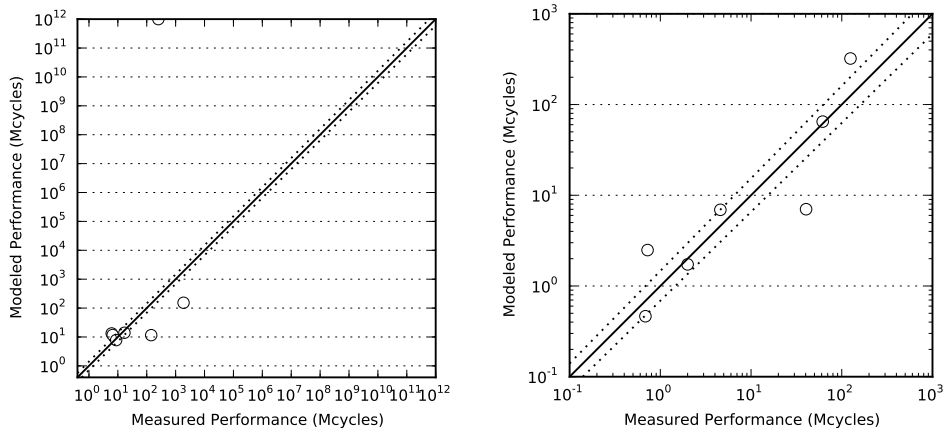


(a) Cycles (b) Speedup

Figure E.6: $M3$-Core: Kepler

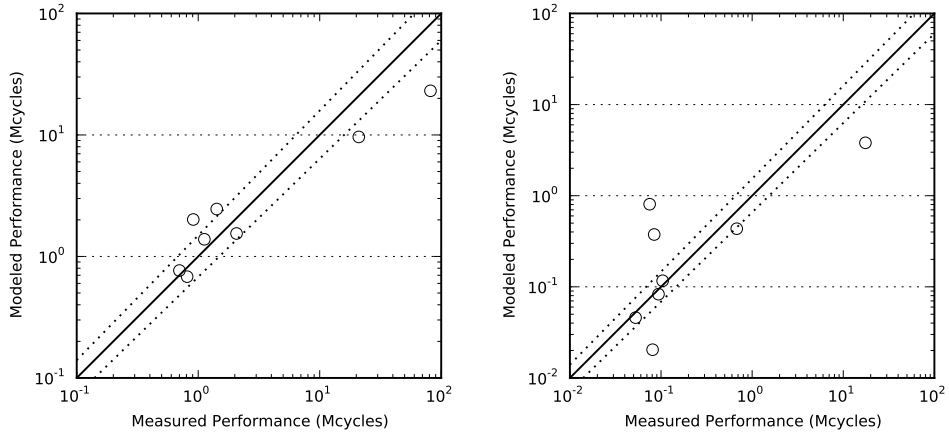
F CYCLE COUNTS USING TRANSLATION MODELS

Below, we show the predicted and measured cycle counts for CAB benchmarks on the two S-Cores and two M-Cores considered in Chapter 5. We include a dotted line with $\pm 40\%$ error. Points that fall within the dotted lines thus have accuracy within 40%.



(a) $S2$: Atom (b) $S4+$: Sandybridge

Figure F.1: S-Cores



(a) $M1$: Pre-Fermi (b) $M3$: Sandybrdige

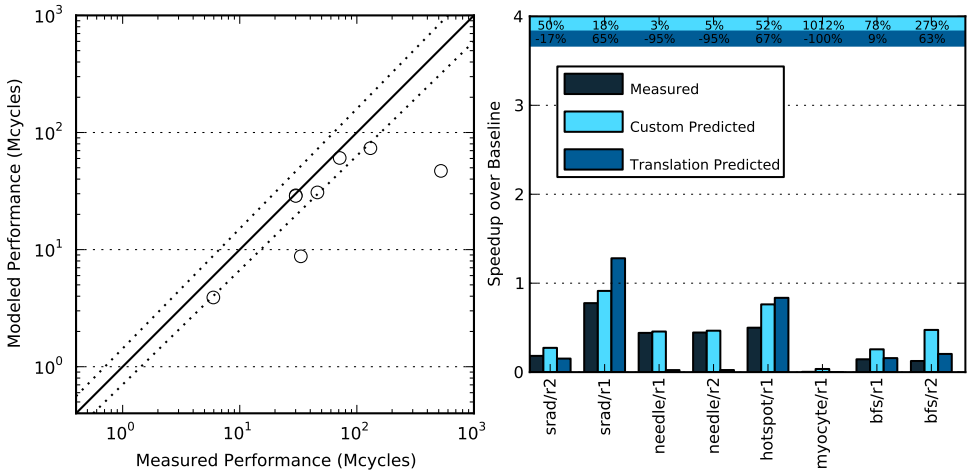
Figure F.2: M-Cores

G TRANSLATION MODELS ON CHALLENGE BENCHMARKS

In this appendix, we consider a more challenging benchmark suite, a sub-set of the Rodinia suite. Note that these benchmarks are not optimized for the S or M cores that we consider. Benchmarks are not necessarily well-optimized in either the S-Core on M-Core source code. As a result, we do not expect our models to achieve the same accuracy as we did with the CAB benchmarks in Chapter 5.

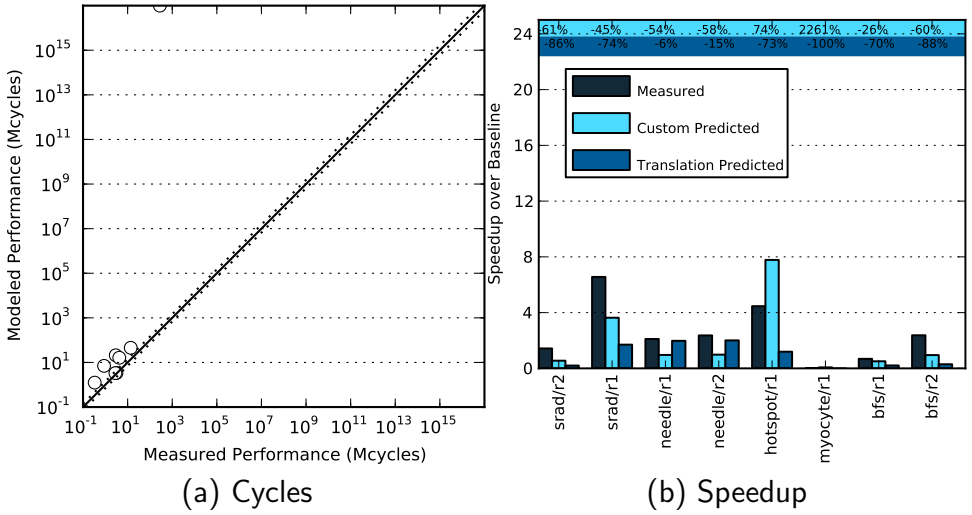
We evaluate the Rodinia benchmarks only for S-Core to M-Core translation direction, as the benchmarks were not compatible with our Ocelot-based tools. We include both a cycle count graph and a bar graph with speedups and percent errors in speedups. On the cycle count graphs, we include a dotted line with ±40% error. Points that fall within the dotted lines thus have accuracy within 40%. A brief nalysis of these results was included in Chapter 4.

We do not do any hand-tuning or warp/block size analysis beyond our push-button tools. Given the expected inaccuracies, the results are surprisingly accurate.



(a) Cycles (b) Speedup

Figure G.1: M-Core: pre-Fermi



(a) Cycles (b) Speedup

Figure G.2: M-Core: Kepler

REFERENCES

[1] Ajeetha, Saktheesh Subramoniapillai. 2012. Architectural analysis and performance characterization of nvidia gpus using microbenchmarking. Master's thesis, Ohio State University.

[2] Allen, F., M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. 1988. A framework for determining useful parallelism. In *Proceedings of the 2nd international conference on supercomputing*, 207–215. ICS '88, New York, NY, USA: ACM.

[3] Amdahl, Gene M. 1967. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring joint computer conference*, 483–485. AFIPS '67 (Spring), New York, NY, USA: ACM.

[4] Azizi, Omid, Aqeel Mahesri, Benjamin C. Lee, Sanjay J. Patel, and Mark Horowitz. 2010. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. In *Proceedings of the 37th annual international symposium on computer architecture*, 26–36. ISCA '10, New York, NY, USA: ACM.

[5] Azizi, Omid, Aqeel Mahesri, John P. Stevenson, Sanjay J. Patel, and Mark Horowitz. 2010. An integrated framework for joint design space exploration of microarchitecture and circuits. In *Proceedings of the conference on design, automation and test in europe*, 250–255. DATE '10, 3001 Leuven, Belgium, Belgium: European Design and Automation Association.

[6] Baghsorkhi, Sara S., Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. 2010. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN symposium on principles and practice of parallel programming*, 105–114. PPoPP '10, New York, NY, USA: ACM.

[7] Bakhoda, A., G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE international symposium on performance analysis of systems and software. ISPASS 2009.*, 163–174.

[8] Baskaran, Muthu Manikandan, J. Ramanujam, and P. Sadayappan. 2010. Automatic c-to-CUDA code generation for affine programs. In *Proceedings of the 19th joint European*

conference on theory and practice of software, international conference on compiler construction, 244–263. CC'10/ETAPS'10, Berlin, Heidelberg: Springer-Verlag.

[9] Bhadauria, M., V.M. Weaver, and S.A. McKee. 2009. Understanding PARSEC performance on contemporary CMPs. In *Proceedings of the 2009 IEEE international symposium on workload characterization*, 98–107. IISWC '09, Washington, DC, USA: IEEE Computer Society.

[10] Bienia, Christian, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on parallel architectures and compilation techniques*, 72–81. PACT '08, New York, NY, USA: ACM.

[11] Blem, Emily, Hadi Esmaeilzadeh, Renee St. Amant, Karu Sankaralingam, and Doug Burger. 2012. Multicore model from abstract single core inputs. *IEEE Computer Architecture Letters* 99(RapidPosts):1.

[12] Blem, Emily R., Jaikrishnan Menon, and Karthikeyan Sankaralingam. 2013. A detailed analysis of contemporary arm and x86 architectures. Tech. Rep. TR1783, Department of Computer Sciences, University of Wisconsin - Madison.

[13] ———. 2013. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *19th IEEE international symposium on high performance computer architecture*, 1–12. HPCA '13.

[14] Bolch, Gunter, Stefan Greiner, Hermann de Meer, and Kishor S. Trivedi. 1998. *Queuing networks and Markov chains*. Wiley-Interscience.

[15] Borkar, Shekhar. 2010. The exascale challenge. Keynote at International Symposium on VLSI Design, Automation and Test (VLSI-DAT).

[16] Breughe, Maximilien, Stijn Eyerman, and Lieven Eeckhout. 2012. A mechanistic performance model for superscalar in-order processors. In *Proceedings of the 2012 IEEE international symposium on performance analysis of systems & software*, 14–24. ISPASS '12, Washington, DC, USA: IEEE Computer Society.

[17] Burger, Doug, and Todd M. Austin. 1997. The SimpleScalar Tool Set Version 2.0. Tech. Rep. 1342, Computer Sciences Department, University of Wisconsin-Madison.

[18] Carrington, Laura, Mustafa M. Tikir, Catherine Olschanowsky, Michael Laurenzano, Joshua Peraza, Allan Snavely, and Stephen Poole. 2011. An idiom-finding tool for increasing productivity of accelerators. In *Proceedings of the international conference on supercomputing*, 202–212. ICS '11, New York, NY, USA: ACM.

[19] Chakraborty, Koushik. 2008. Over-provisioned multicore systems. Ph.D. thesis, University of Wisconsin-Madison.

[20] Che, Shuai, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE international symposium on workload characterization*, 44–54. IISWC '09, Washington, DC, USA: IEEE Computer Society.

[21] Chen, Xi E., and Tor M. Aamodt. 2008. Hybrid analytical modeling of pending cache hits, data prefetching, and MShrs. In *Proceedings of the 41st annual IEEE/ACM international symposium on microarchitecture*, 59–70. MICRO 41, Washington, DC, USA: IEEE Computer Society.

[22] Cho, Sangyeun, and Rami Melhem. 2008. Corollaries to Amdahl's law for energy. *Computer Architecture Letters* 7(1).

[23] Chung, Eric S., Peter A. Milder, James C. Hoe, and Ken Mai. 2010. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd annual IEEE/ACM international symposium on microarchitecture*, 225–236. MICRO '43, Washington, DC, USA: IEEE Computer Society.

[24] Culler, David, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. *LogP: towards a realistic model of parallel computation.* PPOPP '93, New York, NY, USA: ACM.

[25] Diamos, Gregory Frederick, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference*

on parallel architectures and compilation techniques, 353–364. PACT '10, New York, NY, USA: ACM.

[26] Eeckhout, Lieven. 2010. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture* 5(1):1–145. `http://www.morganclaypool.com/doi/pdf/10.2200/S00273ED1V01Y201006CAC010`.

[27] Emma, P.G., and E.S. Davidson. 1987. Characterization of branch and data dependencies in programs for evaluating pipeline performance. *IEEE Transactions on Computers* 36:859–875.

[28] Esmaeilzadeh, Hadi, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on computer architecture*, 365–376. ISCA '11, New York, NY, USA: ACM.

[29] ———. 2012. Dark silicon and the end of multicore scaling. *IEEE Micro* 32(3):122–134.

[30] ———. 2012. Power limitations and dark silicon challenge the future of multicore. *ACM Trans. Comput. Syst.* 30(3):11:1–11:27.

[31] ———. 2013. Power challenges may end the multicore era. *Commun. ACM* 56(2): 93–102.

[32] Esmaeilzadeh, Hadi, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. 2011. Looking back on the language and hardware revolutions: measured power, performance, and scaling. In *ASPLOS '11*.

[33] Eyerman, Stijn, and Lieven Eeckhout. 2010. Modeling critical sections in Amdahl's law and its implications for multicore design. In *Proceedings of the 37th annual international symposium on computer architecture*, 362–370. ISCA '10, New York, NY, USA: ACM.

[34] Eyerman, Stijn, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2009. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.* 27(2):3:1–3:37.

[35] Fang, Jianbin, Ana Lucia Varbanescu, and Henk J. Sips. 2011. A comprehensive performance comparison of CUDA and OpenCL. In *2011 international conference on parallel processing*. ICPP '11.

[36] Fog, Agner. *The microarchitecture of intel, AMd, and via CPUs: An optimization guide for assembly programmers and compiler makers*. Copenhagen University College of Engineering.

[37] Garcia, Saturnino, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor. 2011. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation*, 458–469. PLDI '11, New York, NY, USA: ACM.

[38] Govindaraju, Venkatraman, Peter Djeu, Karthikeyan Sankaralingam, Mary Vernon, and William R. Mark. 2008. Toward a multicore architecture for real-time ray-tracing. In *Proceedings of the 41st annual IEEE/ACM international symposium on microarchitecture*, 176–187. MICRO 41, Washington, DC, USA: IEEE Computer Society.

[39] Govindaraju, Venkatraman, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying functionality and parallelism specialization for energy efficient computing. *IEEE Micro* 33(5).

[40] Govindaraju, Venkatraman, Tony Nowatzki, and Karthikeyan Sankaralingam. 2013. Breaking SIMD shackles with an exposed flexible microarchitecture and the access execute PDG. In *The 22nd international conference on parallel architectures and compilation techniques*. PACT '13.

[41] Gregg, Chris, and Kim Hazelwood. 2011. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *Proceedings of the IEEE international symposium on performance analysis of systems and software*, 134–144. ISPASS '11, Washington, DC, USA: IEEE Computer Society.

[42] Guz, Zvika, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C. Weiser. 2009. Many-core vs. many-thread machines: Stay away from the valley. *IEEE Computer Architecture Letters* 8:25–28.

[43] Gwennap, Linley. 2010. Sandybridge spans generations. *Microprocessor Report*.

[44] Halfhill, Tom. 2008. Intel's tiny Atom. *Microprocessor Report*.

[45] Hammerstrom, Dan W., and Edward S. Davidson. 1977. Information content of cpu memory referencing behavior. In *ISCA '77*, 184–192.

[46] Hardavellas, Nikos, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2011. Toward dark silicon in servers. *IEEE Micro* 31(4).

[47] Hempstead, Mark, Gu-Yeon Wei, and David Brooks. 2009. Navigo: An early-stage model to study power-contrained architectures and specialization. In *Workshop on modeling, benchmarking, and simulations*. MoBs '09.

[48] Hennessy, John L., and David A. Patterson. 2011. *Computer architecture: A quantitative approach*. Morgan Kaufmann.

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

[50] Hong, Sunpyo, and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on computer architecture*, 152–163. ISCA '09, New York, NY, USA: ACM.

[51] ———. 2010. An integrated GPU power and performance model. In *Proceedings of the 37th annual international symposium on computer architecture*, 280–289. ISCA '10, New York, NY, USA: ACM.

[52] Hoste, K., and L. Eeckhout. 2007. Microarchitecture-independent workload characterization. *Micro, IEEE* 27(3):63–72.

[53] Im, Eun-jin, and Katherine Yelick. 2001. Optimizing sparse matrix computations for register reuse in SPARSITY. In *ICCS '01*.

[54] Ipek, Engin, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. 2007. Core fusion: accommodating software diversity in chip multiprocessors. In *ISCA '07*, 186–197.

[55] ITRS. 2011. International technology roadmap for semiconductors, 2010 update.

[56] Jacob, B.L., P.M. Chen, S.R. Silverman, and T.N. Mudge. 1996. An analytic model for designing memory hierarchies. *IEEE ToC '96*.

[57] Karkhanis, Tejas S., and James E. Smith. 2004. A first-order superscalar processor model. In *ISCA 2004*, 338–.

[58] ———. 2007. Automated design of application specific superscalar processors: an analytical approach. In *Proceedings of the 34th annual international symposium on computer architecture*, 402–411. ISCA '07, New York, NY, USA: ACM.

[59] Kerr, Andrew, Gregory Diamos, and Sudhakar Yalamanchili. 2010. Modeling GPU-CPU workloads and systems. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, 31–42. GPGPU '10, New York, NY, USA: ACM.

[60] Kim, Changkyu, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. 2007. Composable lightweight processors. In *MICRO '07*, 381–394.

[61] Kim, Yooseong, and Aviral Shrivastava. 2011. CuMAPz: a tool to analyze memory access patterns in cuda. In *Proceedings of the 48th design automation conference*, 128–133. DAC '11, New York, NY, USA: ACM.

[62] Kinney, Larry L., and R. G. Arnold. 1978. Analysis of a multiprocessor system with a shared bus. In *ISCA '78*, 89–95.

[63] Kulkarni, Milind, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. 2009. How much parallelism is there in irregular applications? In *PPoPP '09*.

[64] Lai, Junjie, and André Seznec. 2011. TEG: GPU Performance Estimation Using a Timing Model. Rapport de recherche RR-7804, INRIA.

[65] Lazowska, E., J. Zahorjan, G.S. Graham, and K. Sevcik. 1984. *Quantitative system performance*. Prentice-Hall.

[66] Lee, Benjamin C., and David Brooks. 2010. Applied inference: Case studies in microarchitectural design. *ACM Trans. Archit. Code Optim.* 7:8:1–8:37.

[67] Lee, Seyong, and Rudolf Eigenmann. 2010. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *SC 2010*.

[68] Lee, Victor W., Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA 2010*.

[69] Li, Sheng, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture*, 469–480. MICRO 42, New York, NY, USA: ACM.

[70] Loh, Gabriel. 2008. The cost of uncore in throughput-oriented many-core processors. In *Workshop on architectures and languages for throughput applications*. ALTA '08.

[71] Luk, Chi-Keung, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*.

[72] Luo, Cheng, and R. Suda. 2011. A performance and energy consumption analytical model for GPU. In *2011 IEEE ninth international conference on dependable, autonomic and secure computing*, 658–665. DASC '11.

[73] Maleki, Saeed, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An evaluation of vectorizing compilers. In *PACT '11*.

[74] Meng, Jiayuan, Vitali A. Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D. Uram. 2011. Grophecy: GPU performance projection from CPU code skeletons. In *SC '11*.

[75] Meswani, M.R., L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole. 2011. Modeling and predicting performance of high performance computing applications on hardware accelerators. In *IISWC '11*.

[76] Navada, Sandeep, Niket K. Choudhary, and Eric Rotenberg. 2010. Criticality-driven superscalar design space exploration. In *Proceedings of the 19th international conference on parallel architectures and compilation techniques*, 261–272. PACT '10, New York, NY, USA: ACM.

[77] Nickolls, John, and William J. Dally. 2010. The GPU computing era. *IEEE Micro* 30(2).

[78] Nugteren, Cedric, and Henk Corporaal. 2012. The boat hull model: enabling performance prediction for parallel computing prior to code development. In *Proceedings of the 9th conference on computing frontiers*, 203–212. CF '12, New York, NY, USA: ACM.

[79] NVIDIA Corporation. 2009. NVIDIA's next generation CUDA compute architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[80] ———. 2012. NVIDIA's next generation CUDA compute architecture: Kepler GK110. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[81] Parakh, A.K., M. Balakrishnan, and K. Paul. 2012. Performance estimation of GPUs with cache. In *2012 IEEE 26th international parallel and distributed processing symposium workshops phd forum*, 2384–2393. IPDPSW '12.

[82] Patel, Janak H. 1978. Pipelines wth internal buffers. In *ISCA '78*, 249–255.

[83] Pharr, Matt, and William R. Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *InPar 2012*.

[84] Pollack, F.J. 1999. New microarchitecture challenges in the coming generations of CMOS process technologies. MICRO Keynote.

[85] Rege, S. L. 1976. Cost, performance and size tradeoffs for different levels in a memory hierarchy. In *ISCA '76*, 64–67.

[86] Sandhu, Tarinder. 2010. NVIDIA's GeForce GTX 480 finally unleashed. *HEXUS*. Contains GTX480 die photo.

[87] Satish, Nadathur, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. 2012. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *ISCA '12*.

[88] Shrout, Ryan. 2012. NVIDIA revelas GK110 GPU - kepler at 7.1b transistors, 15 smx units. *PC Perspective*. Contains GK110 die photo.

[89] Sim, Jaewoong, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 2012. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proceedings of the 17th acm sigplan symposium on principles and practice of parallel programming*, 11–22. PPoPP '12, New York, NY, USA: ACM.

[90] Sorin, D. J., V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. 1998. Analytic evaluation of shared memory parallel systems with ILP processors. In *Proceedings of the 25th international symposium on computer architecture*, 380–391. ISCA '98.

[91] Stratton, John A., Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commerical throughput computing. Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign.

[92] Suleman, Aater M., Onur Mutlu, Moinuddin K Qureshi, and Yale N. Patt. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS '09*, 253–264.

[93] Suleman, M. Aater, Moinuddin K. Qureshi, and Yale N. Patt. 2008. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. *SIGARCH Comput. Archit. News* 36(1):277–286.

[94] Sullivan, Herbert, and Theodore R. Bashkow. 1977. A large scale, homogeneous, fully distributed parallel machine. In *ISCA '77*, 105–117.

[95] Tournavitis, Georgios, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. 2009. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI '09*.

[96] Towsley, Donald F. 1978. The effects of CPU: I/O overlap on computer system configurations. In *Proceedings of the 5th annual symposium on computer architecture*, 238–241. ISCA '78, New York, NY, USA: ACM.

[97] Unat, Didem, Xing Cai, and Scott B. Baden. 2011. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In *ICS*.

[98] Venkatesh, Ganesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation cores: reducing the energy of mature computations. In *ASPLOS*.

[99] Wade, James F., and Paul D. Stigall. 1974. Instruction design to minimize program size. In *ISCA '74*, 41–44.

[100] Wolfe, Michael. 2010. Implementing the PGI accelerator model. In *GPGPU '10*.

[101] Wong, H., M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *2010 ieee international symposium on performance analysis of systems software*, 235–246. ISPASS '10.

[102] Woo, Dong Hyuk, and Hsien-Hsin S. Lee. 2008. Extending Amdahl's law for energy-efficient computing in the many-core era. *Computer* 41(12):24–31.

[103] Zhang, Yao, and J.D. Owens. 2011. A quantitative performance analysis model for GPU architectures. In *2011 IEEE 17th international symposium on high performance computer architecture*, 382–393. HPCA '11.