Finding, Measuring, and Reducing Inefficiencies in Contemporary Computer Systems

Melanie Kambadur

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

COLUMBIA UNIVERSITY

2016

ABSTRACT

Finding, Measuring, and Reducing Inefficiencies in Contemporary Computer Systems

Melanie Kambadur

Computer systems have become increasingly diverse and specialized in recent years. This complexity supports a wide range of new computing uses and users, but is not without cost: it has become difficult to maintain the efficiency of contemporary general purpose computing systems. Computing inefficiencies, which include nonoptimal runtimes, excessive energy use, and limits to scalability, are a serious problem that can result in an inability to apply computing to solve the world's most important problems. Beyond the complexity and vast diversity of modern computing platforms and applications, a number of factors make improving general purpose efficiency challenging, including the requirement that multiple levels of the computer system stack be examined, that legacy hardware devices and software may stand in the way of achieving efficiency, and the need to balance efficiency with reusability, programmability, security, and other goals.

This dissertation presents five case studies, each demonstrating different ways in which *the measurement of emerging systems can provide actionable advice to help keep general purpose computing efficient.* The first of the five case studies is *Parallel Block Vectors*, a new profiling method for understanding parallel programs with a fine-grained, code-centric perspective aids in both future hardware design and in optimizing software to map better to existing hardware. Second is a project that defines a new way of measuring *application interference* on a datacenter's worth of chip-multiprocessors, leading to improved scheduling where applications can more effectively utilize available hardware resources. Next is a project that uses the *GT-Pin* tool to define a method for accelerating the simulation of GPGPUs, ultimately allowing for the development of future hardware with fewer inefficiencies. The fourth project is an *experimental energy survey* that compares and combines the

latest energy efficiency solutions at different levels of the stack to properly evaluate the state of the art and to find paths forward for future energy efficiency research. The final project presented is *NRG-Loops*, a language extension that allows programs to measure and intelligently adapt their own power and energy use.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I owe thanks to many people who have supported and enriched my experiences as a Ph.D. student. First, I want to thank my advisor Martha Kim for helping me to generate and refine research ideas, for meticulously reading paper drafts, for all of the long hours she spent with me leading up to deadlines, and for helping to improve my writing, presentation, and research skills. Martha was a great research partner!

Many professors at Columbia have given me invaluable instruction, advice, and feedback on my work. In particular, I would like to thank Al Aho, Simha Sethumadhavan, and Luca Carloni, who gave me the most feedback on my work and are all (not so coincidentally) serving on my thesis committee. I also want to thank Adrian Sampson for serving on my thesis committee, and a number of wonderful internship mentors from whom I received excellent guidance during my Ph.D., including Tipp, Rick, Nenad, C.K., Harish, Sunpyo, Matthai, Karin, and Gagan. Several of these people are also co-authors on the work presented in this dissertation. I had only one student co-author, Kui, who deserves my acknowledgement and thanks for his help with the PBV work. I was very fortunate to receive financial support from a number of companies, government agencies, professional groups, and individuals, and I am grateful to all of them as well. I also received wonderful administrative assistance from our department staff, in particular from Jessica, Daisy, Elias, and Cindy.

Next, I want to thank the students in the CSL and arch-reading groups who attended my talks and gave me project feedback. I especially want to thank Lisa, John, Robert, Kanad, Joel, Andrea, and Emilio, some of my fellow "fish bowl" inhabitants who crossed the line from colleagues to friends. I am also thankful to all of the WICS board ladies for helping me create lots of great memories as we planned our events, especially Arthi, Erica, and Heba with whom I spent the most time over the years, and our faculty advisor Julia. I have been lucky to make additional supportive friends at Columbia, including Jon, Eva,

Bingyi, Anna, Sasha, Cinar, and my "study group" — Sebastian, Paul, and Barbara — who helped me through the early years of my Ph.D.

My thanks as well to my three brothers and my parents for their support and encouragement, and in particular to my father for inspiring me to study computer science (and pursue a Ph.D.) in the first place. Of course, my biggest thanks must go to my amazing husband, Anju, who has been the most important person in my life since I met him nearly a decade ago.

Chapter 1

Introduction

A proliferation of computing uses, users, and producers in recent years has resulted in a diverse range of computing technologies at all levels of the computer system stack. While exciting for the computing industry, this ubiquity-induced complexity of computer systems has made it difficult to effectively match the requests of arbitrary software applications to general purpose hardware and operating systems. This dissertation will demonstrate the presence of resultant hardware-software mismatches and show that they cause runtime or energy efficiency losses. It will then show that new measurement technologies can be effective in maintaining reasonable efficiency in current systems and in achieving better efficiency in future systems.

The remainder of this introduction explains the diversity and complexity within the modern system stack, explains how efficiency issues arise in general purpose computing, and discusses the importance of efficiency as well as other system design considerations. Following this discussion is a brief overview of how measurement and analysis technologies can identify efficiency issues and improve them, a summary of the contributions of this dissertation, and an outline for the remainder of the document.

1.1 Computing Diversity

Computing has entered a new era of ubiquity. Over 3 billion people now purchase and use over 2 billion personal computers, tablets, and smart phones *per year* [67, 163]. There

are 5 billion "things" (i.e., devices such as cars or thermostats) currently connected to the *Internet of Things (IoT)* [68], and there are over 6.5 million technology workers in the United States alone [74]. The number of computer science researchers is also growing — in 2014 the United States produced around 1800 computer science Ph.D.s per year as opposed to 1000 per year in the 1990s and 200 per year in the late 1970s and early 1980s [237, 269].

In keeping pace with the expanding number of uses, users, and producers, computing technology has become increasingly specialized and diverse. As recently as the late 1980s, there were primarily two classes of computers, supercomputers and PCs, with a handful of simple embedded devices, such as pocket calculators. Today, there are many categories of computers. There are massive datacenters hosting cloud computing services and supplying storage space to record as much as possible of a "big-data" obsessed world. For personal computing, there are smartphones and a variety of of desktop servers, laptops, tablets, and "2-in-1's". In addition to phones, almost every form of electronics has become "smart" with the help of embedded computers, from watches, fitness monitors, and sports equipment to televisions, thermostats, and refrigerators, to cars, trains, and airplanes, to factory equipment and computerized "drones". Beyond this, today's computers live not just in electronics, but also on and in plants and animals, from GPS microchips, to pacemakers and cochlear implants, to nanotechnology.

To serve the myriad users and uses, and because so many different people are contributing to the development of modern computing technologies, diversity has percolated to nearly all of the components and layers of the computer system stack. On the architectural side, processing logic could be a basic general purpose CPU, a specialized processor such as a domain specific processor or micro-controller, and it could be programmable logic (i.e., FPGAs), or a fixed–function chip (i.e., ASICs). Beyond these processing categories, there are many different configuration options such as whether processing cores are in–order or out–of–order, whether multiprocessing is used (either in the form of simultaneous–multiprocessing [SMT] or chip–multiprocessing [CMP]), and whether a single chip contains multiple heterogeneous processors. Memory technologies are equally diverse and also frequently heterogeneous, utilizing SRAM, DRAM, FLASH and likely soon a variety of other types of Non-Volatile (NV)RAM. Memory and processors may be 2.5– or 3–D stacked tech-

nologies, or utilize near-data processing (NDP), where processing units are moved adjacent to or into data storage [13]. Languages are diverse as well, and may be objected oriented, compiled or interpreted, domain specific, dynamic or static. As with hardware, it is common to see *heterogeneous software*, with one application utilizing multiple languages. The number of higher-level languages has grown from about 200 in 1972 to 8500 in the present day, a count that exceeds that of known human languages [79]. Handling the translation between all of these languages and platforms has required that compilers become complex too; and operating systems face a similar challenge in providing a secure and simple interface to users on the diverse range of devices. For example, the Android operating system must be made to work on roughly 4000 distinct hardware platforms [248].

Dealing with computing diversity presents at least one notable problem: it has become difficult to effectively map arbitrary software requests to general purpose hardware resources. When hardware–software mappings are nonoptimal, the result is *inefficiency*, or the overuse of runtime, power, or both (i.e., energy).

1.2 Hardware-Software Mismatches Cause Efficiency Problems

The hardware-software matching problem is an issue where either 1) more resources are delivered by hardware than are needed by software, 2) more resources are requested by software than can be delivered by hardware, or 3) where both 1) and 2) occur simultaneously.

Though not always the case, the goals of programming languages and computer architecture are frequently at odds. At the kernel of each problem we want to compute, there is some hard-to-define amount of *necessary computational resources*. Translating from human intent to something a machine can understand — i.e., writing a computer program — sometimes results in resource requests beyond the essential. Similarly, making hardware delivery too potent sometimes means more resources are delivered than strictly needed. It is possible for software and hardware to over– or under–provision simultaneously: for example, a program might use a higher–precision data type (e.g., a 64-bit `float`) when a lower–precision data type would suffice (e.g., an 8-bit `float`), and hardware might supply

multiple processors when only a single processor is needed by the program. This (potentially dual) mismatch in requested resources and delivered resources can result in wasted execution cycles, wasted power, or both. Evidence of efficiency losses might include cache misses, pipeline stalls, processors that are idle but still drawing power, TLB misses, excess page swapping, task starvation, or tail latencies.

In a perfect world, computer programmers and their programs would make whatever requests they desire and hardware would deliver as many resources as it could within size, thermal, and cost constraints. Then, the compiler, the operating system, and middleware would make adjustments between the two to eliminate mismatches. Unfortunately, this is a job that is only partially solved by today's compilers and operating systems, and it is a job that is becoming increasingly difficult to automate as hardware and software complexity increase. In order to avoid unnecessary compromises to programmability and hardware design, and to reduce resource waste, we need to improve our understanding of the efficiency between hardware and software, and make coordinated efforts between applications, compilers, operating systems, and hardware to reconcile mismatched resource requests and delivery to the greatest extent possible.

1.3 Why Efficiency is (Still) Important

Over time, advances in computer architecture have dramatically increased computational efficiency. The 1946 Eniac computer drew 150,000 Watts of power to deliver 500 floating point operations per second (FLOPS), while modern smartphones consume around a single Watt to deliver a peak performance of billions of FLOPS [6, 197]. In fact, a single modern smartphone has more computational capabilities than the computers used to get Apollo 11 to the moon [205]. Today's fastest supercomputer, the Tianhe-2, has a processing rate of 33.86 *peta*FLOPS — that's nearly 34 million–billions of operations per second [53].

With such commendable advances, the need for even more runtime and power efficiency might not be immediately apparent, but there are a number of important reasons to care about continued efficiency advances. Some of these are:

- **To reduce computing's ecological footprint.** The IT-sector is already responsible

for around 2% of global carbon emissions, and with the expected growth in computing use and users, this number could soon be a lot higher if energy efficiency improvements are not made [44]. On a more local level, the energy costs of technology could become a barrier for individual users if not kept in check — today, an iPhone already has a higher monthly electric bill than a standard home refrigerator [253].

- **To solve new problems or put computers in new places.** Greater efficiency is required to reach the "exascale" era of computing (where computers can compute a billion–billion operations per second), which promises technological advances such as fully replicating the brain and providing global climate solutions such as controlled fusion. On the other side of computing scale, with greater efficiency, better miniaturization could be achieved and an entirely different set of important problems could be solved. For example, scientists are working on computers so small they can be swallowed in a pill and will transmit medical readings to doctors [177].

- **To make existing computing solutions more cost effective or accessible.** Efficiency is not just important for pushing the limits of the kinds of problems that computers can solve. It is also import for increasing the availability or decreasing the cost of existing computation. For example, partly due to increased computational efficiency, biotechnicians were able to reduce genome sequencing costs from $10 million in 2007 to under $1000 in 2011 [78, 180]. As another example, image recognition (i.e., computer vision) is performed today on powerful servers in the "cloud", but with better efficiency, image recognition could be performed on mobile devices to improve privacy and latency for users, and to decrease costs for service providers.

1.4 Considerations Besides Efficiency

The systems today that are closest to optimially efficient are called application-specific integrated circuits, or ASICs. ASICs are hrdware customized to specific, well-tuned applications to minimize inefficiencies, nearly eliminating undesirable events such as pipeline stalls and cache misses, and saving orders of magnitude of energy. Anton is one example of an ASIC that targets molecular biology applications, improving runtime performance by

nearly two orders of magnitude and considerably reducing power versus general-purpose computing solutions [209]. For many applications, however, complete specialization is not a panacea, and is instead impractical, undesirable, or both. Efficiency in computer system design must be balanced with a multitude of competing goals, such as:

- **Security** at both the software and hardware level, which can sometimes compete with efficiency by adding extra circuity or code to ensure a system is protected.
- **Programmability** or the ability to write programs quickly and without regard to efficiency, perhaps in languages that tend to request more resources than necessary. For example, object oriented programming — which helps developers reason about program structure and arguably write more readable code — has been shown to unnecessarily waste energy [20].
- **Reusability** of software functions and objects, and of instruction set architectures and functional units in hardware. Reusability saves human resources, such as developer time, but can be at odds with computational efficiency. For example, programs may link an entire statistics library when they only need one of its functions.
- **Accuracy and Verifiability**; extra precision or additional functionality such as test cases are often employed to ensure error-less programs, and there is value in being able to prove that a program is correct in all circumstances (e.g., in all of the non-deterministic runs of a multithreaded program). Both of these goals typically require an efficiency tradeoff.
- **Reasonable Area, Device Size, Durability** and other similar hardware goals also may require trading away computational efficiency.
- **Cost.** Many of the above are also associated with reduced expenses for computing producers or consumers. For example, buying one very computationally-powerful server is not typically as cost-effective as buying many wimpy servers at a lower price per unit. Similarly, while writing code in C++ and CUDA might provide the fastest execution time for an application, a company may find it cheaper to hire Python developers and pay for more machine time.

Another important and related issue is the presence of legacy code and devices. Often, it is prohibitively expensive for companies to replace obsolete devices, because efficiency is a second class goal to cost. Later in this dissertation, there are a couple of examples of the *legacy effect*. For example, in the study of application interference on Google servers (Chapter 4), we note that the search giant uses a wide variety of microarchitectural platforms. Also, we discuss a new trend in Chapter 5, where once specialized processors — Graphics Processing Units (GPUs) — have now been put to general purpose use because of their low price point and ubiquity.

1.5 Measurement's Role in Improving Efficiency

The efficiency of contemporary and emerging computer systems can be improved with the guidance of performance analysis techniques and measurement methodologies. Given the delicate balancing act of honoring competing system design goals, rather than striving to completely eliminate inefficiencies, the aim should be to understand them in the context of other tradeoffs, and to reduce them only as appropriate. Additionally, we need to determine when it is necessary to optimize or specialize applications or hardware, versus when it is possible to reconcile the mismatches with system tools. New measurement methods and tools can help locate potential efficiency issues, and can evaluate the potential tradeoffs of solutions to reduce them. Measurement and program analysis are important for a number of reasons:

- **To compare alternate solutions.** Measurement helps us compare different solutions' efficiency, and can determine which microarchitecture, which algorithm, which language, or which optimization technique is best. For example, in Chapter 6, we experimentally compare different energy efficiency techniques that span the computer system stack.

- **To account for real–time data.** Efficiency is often dependent on real–time events, such as user inputs or battery-levels. Our measurement–based methodologies in Chapters 4 and 7 are able to account for both.

- **To identify hotspots.** It is common for efficiency issues to be localized to small parts of the software or hardware. Measurement–based techniques are well-suited for pinpointing such hotspots, especially when their poor efficiency is contributed by more than one layer of the system stack. We demonstrate this in Chapter 3.

- **To test future designs.** When possible, proactive efficiency solutions are better than retroactive efficiency solutions. Measurement technologies can help direct the way towards more efficient hardware-software co-designs, as is shown in Chapter 5.

To summarize, measurement techniques are effective for addressing efficiency issues in modern general purpose computer systems because they help us account for dynamic events, help us understand unpredictable interactions between system layers, and because they help us to be proactive in avoiding efficiency issues even before they occur.

1.6 Summary of Contributions

As discussed above, the novel contributions of this dissertation support our thesis that *measurement techniques that work at the intersection of hardware and software can improve the efficiency of contemporary and emerging general purpose computer systems.* The contributions are divided into five case studies — one per chapter — that find inefficiencies within a variety of recent or emerging hardware and software paradigms. Most of this work has been previously written about in nine peer-reviewed publications and a technical report, each of which were primarily authored by the author of this dissertation [113–122].

The first contribution is *Parallel Block Vectors*, or PBVs, a new way of profiling program parallelism at very fine granularity. PBVs help identify opportunities to match future hardware to current software, or to optimize future software versions for current hardware. The second contribution is a new method of identifying minimal but representative regions within programs written for GPGPUs (general purpose graphics processing units). Finding these salient regions allows for the acceleration of cycle-accurate performance simulators, which in turn leads to GPUs with resource deliveries that are more adequately matched to software needs. The third contribution of this dissertation is a new method of profiling the interference between multiple applications in a datacenter that are forced to *co-locate* on

a single multicore for the sake of improving system efficiency. Unfortunately, co-location's attempt to ameliorate system throughput can sometimes negatively affect individual applications' latency. Our new profiling method looks for opportunities to co-locate applications that preserve both full system throughput and individual application latency.

The next contribution of this dissertation is a measurement–based language extension called *NRG-Loops*. NRG-Loops allow applications to react to power and energy measurements taken at runtime with on-the-fly adjustments to functionality, performance, and accuracy. Since the adjustments are conditionally enabled, NRG-Loops have the potential to allow a single piece of source code to match its resource requests to multiple types of underlying architectures, without significant effort on the part of the programmer. The final contribution is an *experimental survey* that quantifies the relative effectiveness of previously uncompared energy efficiency techniques across the system stack. The survey also combines different techniques to find whether their compound effects are negative or positive, and whether they are additive or synergistic.

1.7 Dissertation Outline

The remainder of this dissertation is organized as follows:

- **Chapter 2** provides background information on the terminology and components associated with computer systems, as well as on the measurement and analysis methods that existed prior to this dissertation.
- **Chapter 3** presents the new Parallel Block Vector profiling tool for examining program parallelism from a fine-grained, code-centric perspective.
- **Chapter 4** presents the new measurement method for quantifying (and mitigating) application interference on Datacenter CMPs.
- **Chapter 5** presents a new method for identifying minimal but representative regions of programs in GPGPUs, with the aim of accelerating microarchitecture performance simulators.

- **Chapter 6** presents an experimental survey that compares and combines energy efficiency solutions at different levels of the compute stack.
- **Chapter 7** presents the new NRG-Loop syntax extension that allows programs to intelligently react to their own power and energy use.
- **Chapter 8** summarizes the contributions of this work and suggests ideas for the community's future work.
- **Appendix 8.2** defines acronyms that appear throughout this dissertation.

Chapter 2

Background: Measuring the Intersection of Hardware and Software

Analysis tools and measurement techniques for understanding the behaviors that arise at the intersection of computer software and computer hardware are an essential part of improving computer systems' efficiency. This chapter begins with a discussion of the terminology and components associated with computer systems in Section 2.1 to give readers context for the holistic, whole–system approach that we need to take to improve system efficiency. Next, Section 2.2 gives a high-level overview of the main types of computing analyses available today. There are many analysis methods one might use to study computer systems, but only some are relevant to this dissertation's goal of identifying and reducing system-wide inefficiencies; Section 2.3 explores the most relevant types of analyses in more depth.

2.1 Computer Systems

When this dissertation refers to *computer systems*, it means all of the hardware and software technology that come together to form a computing device and its operations. Prior work is inconsistent in how it breaks down the layers in a computer system; we choose to divide the system into three coarse layers: *Hardware*, *System*, and *Application*. The

term *platform* is sometimes used in this and other works to encompass both the Hardware– and System–Layers, while the term *software* includes the System– and Application–Layers. The term *stack* or *system stack* is frequently used to describe an ordered set of the system layers, typically with the Hardware–Layer at the bottom of the stack, the System–Layer at the middle of the stack, and the Application–Layer at the top of the stack. Below is a description of the components or configuration options within each layer. As discussed in the introduction, computer systems are now vastly diverse, so the following list is not comprehensive, but it does cover all major characteristics of the computer systems explored later in this dissertation. By convention, layer components are denoted in **boldface**, while concepts and configuration options are *italicized*.

2.1.1 Hardware–Layer

The Hardware–Layer houses all of the tangible, physical elements of the computer system. These include **power supplies** and **peripherals**, that allow external communication to the computer such as **monitors** and other types of **displays**, **keyboards**, **microphones**, and **cameras**. The layer also includes everything inside of a computer **case**, such as **fans**, **I/O** and other **buses**, **batteries**, and the **motherboard**. The motherboard holds circuitry that makes up the computer's **microarchitecture**, or particular implementation of an **instruction set architecture (ISA)**'s **processor**, including **datapaths**, **execution units (EUs)** such as *arithmetic logic units (ALUs)*, *floating point units (FPUs)*, and *branch prediction units*, as well as *on-die* memory **caches** which today are typically implemented using *static random access memory (SRAM)* technology. The next largest layer of the *memory hierarchy* — the **main memory** — is today most frequently *dynamic random-access memory (DRAM)*, and is also on the motherboard. New types of memory technologies are of continual interest to the architecture research community, and recently a popular area of exploration for caches or main memory technology is *non-volatile RAM (NVRAM)*. NVRAM is persistent, meaning that it retains its memory state even if its power supply is cut. A few types of NVRAM currently being explored include Phase-Change RAM (PCRAM), *Magnetoresistive RAM (MRAM)* and *Resistive RAM (RRAM)* [193]. A particular type of NVRAM called *flash* is already widely utilized for the largest the largest layer of the

memory hierarchy — **storage**. Alternatively, storage may be on *hard disk drives (HDDs)*.

Processor technology long ago moved beyond the basic Von Neumann model of a single *central processing unit (CPU)*, and has since been undergoing continuous changes. In particular, computers today now frequently use *parallel* and/or *heterogeneous processing*. Parallel processing simultaneously executes multiple instructions, and may be implemented as *simultaneous multithreading (SMT)*, *chip–multiprocessing (CMP)*, or often, both. In the SMT model, one *superscalar* processor contains multiple independent *hardware threads* that each issue their own instructions within a single *cycle*. CMP multiprocessing integrates at least two (though possibly hundreds of) independent processors on a single *chip*. It is also common to see *server* computers with more than one *socket* — so that one motherboard actually contains multiple CMPs, each of which may be simultaneously multithreaded. Heterogeneous processing, or computer systems that use more than one kind of processor or core, typically each with unique ISAs, is also gaining popularity. Heterogeneous processing systems often include a CPU as well as some application specific processors such as *graphics processing units (GPUs)*. Cores may be implemented with custom logic (i.e., *application–specific integrated circuits [ASICs]*), or with reprogrammable logic (i.e., *field-programmable gate arrays [FPGAs]*). An alternative or supplement to heterogeneous processing is *asymmetric multiprocessing (AMP)*, where cores use the same ISA, but are of different sizes. AMP is currently popular because it allows users to exploit different choices of power– and runtime–efficiency tradeoffs for possible energy savings.

2.1.2 System–Layer

The System–Layer contains a piece of software called the **operating system (OS)** that connects users and applications to the Hardware–Layer, manages hardware settings, and attempts to ensure computer–wide security. In some computer layer taxonomies, the System–Layer is merely an analogue for the computer's operating system, but our classification also includes ***System* Virtual Machines (VMs)**, web **browsers**, the **BIOS** and other types of **firmware**.

- System VMs are essentially platforms within an OS; a system VM abstracts the hardware of a single machine into multiple virtual *partitions*, each of which may execute its

own operating system environment. To the overlying operating systems, the virtual machines appear identical to real hardware.

- Browsers are software that facilitates communication with the internet. They allow users to both post and receive information. Although browsers could be considered a part of the Application–Layer, we include them here because with the advent of technologies such as complex web applications and Google's Chromebook, the line between operating systems and browsers is blurring [254].

- A system's BIOS or Basic Input/Output System is a type of *firmware* — software that controls and monitors hardware, and is permanently stored in read–only memory. The BIOS handles the *booting* (i.e., power startup) process of computer hardware.

In addition to maintaining basic hardware functionality, the System–Layer is now taking on the relatively new role of tuning dynamically adjustable hardware configurations. An example of this that is discussed in detail later in this dissertation is dynamic voltage and frequency scaling (DVFS) — a process where the operating system adjusts voltage supplies or hardware frequencies to manage power usage.

2.1.3 Application–Layer

The Application–Layer contains **program source code** and the tools and mechanisms that connect that code to the platform layers. These include **programming languages** of which there are many types: *assembly* languages, *high-level* languages (which may be further divided into categories such as *imperative/functional*, *object-oriented/procedural*, *static/dynamic*), *scripting* languages, *domain specific* languages, and *visual* languages (i.e., those that are not text–based). These categories are not necessarily mutually exclusive, for example the `AWK` language could be considered both a scripting language and a domain specific language because it is primarily used for text data processing. This layer also includes the **interpreters**, **compilers**, and ***process* or *application* Virtual Machines (VMs)** that implement programming languages. Each of the three differs with respect to how this is done; for interpreters, source code is parsed and directly executed on the target platform, or occasionally first translated into an **intermediate representation**. Compilers

translate the source code to a platform-specific **target program**, usually incorporating both *machine-independent* and *machine-dependent* optimizations for a more efficient target program. Application VMs translate source to a *platform-independent* target language, then wrap the resultant program and their own functionality into a single process to execute. VMs often feature just–in–time (JIT) compilation, where most compilation is conducted at run time, in contrast to ahead–of–time (AOT) compilation. Also in this layer are the **assemblers** and **linkers**, that respectively (1) follow up after the compiler to convert *assembly programs* into *object files* and (2) merges the object files into a single *executable file*. After linking, executables are *loaded* by the operating system, which involves creating memory space for the program and starting its execution (the **loaders** that complete this operation could be considered a part of the System–Layer, but we place them here for continuity).

Finally, there are **application program interfaces (APIs)** and **libraries**. APIs specify how different computing components may interact via *functions*, object collections, or *protocols*. APIs work as generalized connectors, for example making it easy for applications to communicate with GPUs (e.g., OpenCL) or other hardware components (such as hardware disk drives or video cards), or with databases and web browsers. APIs are frequently implemented as libraries, which consist of a collection of functions written in a programming language, that have well–defined behavior and invocation procedures. Libraries may connect languages to system functionalities (an example being the `pthreads` library that allows C and C++ programs to spawn new threads), provide hardware–specific high–efficiency implementations (as do the CUDNN library for accelerating *deep neural networks* on GPUs or optimized BLAS libraries for *scientific computing kernels*), or simply abstract broadly used functions to aid software engineers (for example, the C++ Standard Template Library [STL] that contains generic implementations for objects such as hash maps and heaps).

2.1.4 Distributed Computing

Distributed computing is a relatively old technology with origins in the 1970s [4], that has much more recently become ubiquitous. Distributed computing connects multiple computers with a network, coordinating their communication and data sharing via some form of

message passing. It could be considered a part of the System–Layer, but we list distributed computing separately here because it differs from traditional single computer systems in three key ways. First, distributed computing systems contain multiple instances of individual computer systems, each autonomously operating and each with its own local memory. Second, distributed systems are *asynchronous* — the constituent computers do not share a global clock. Third, distributed systems can tolerate failures of entire computers and can still continue computation; a mechanism called *independent failure*.

There are a couple of circumstances responsible for the current popularity of distributed computing. The first is the need that some companies and agencies have for reliability greater than that of a single computer, and in particular that of a single hard drive. Another reason is that it is frequently cheaper to purchase multiple less powerful (in terms of operations per second, or RAM size) machines, than it is to purchase one machine with the same operational capabilities. Additionally, even the most powerful computer can sometimes not service one organization's data storage or I/O request bandwidth needs. Finally, outsourced server management (i.e., to distributed computer service providers such as Amazon Web Services[1]) is becoming a very popular business decision.

2.2 An Overview of Computing Analyses

Before getting into the specifics of the measurement and analysis techniques that this dissertation will use, we need to understand the larger context of all the tools and methods available to analyze computer systems.

To organize the available tools and methodologies, we consider the general type of analysis conducted. The three main types of analysis are:

1. **Static Program Analysis**
2. **Hardware Synthesis, Emulation, and Simulation**
3. **Dynamic Analysis and Measurement**

[1] https://aws.amazon.com/

Static program analysis is a set of methods that examines source code or assembly code without executing it. Analysis may be done by humans (for example, Big-O notation for estimating runtime), or by tools (for example, compilers checking for type correctness or optimization opportunities). Static analysis considers code alone, in isolation of the platform and hardware, so it is difficult to extrapolate the analysis data into information that represents whole system behavior. Due to its lack of suitability for examining complex interactions between system layers, static analysis is not a primary focus of this dissertation.

Synthesis, emulation, and simulation are techniques used to design and optimize hardware. Hardware synthesis takes an algorithmic description of a problem and implements the behavior it describes in hardware. For the sake of optimization, the synthesis process performs different types of analyses. Emulation is when one piece of hardware is imitated with another piece of hardware, typically to analyze how a not–yet–existing piece of hardware might perform. Simulation has a similar goal to emulation, but models in detail the internal states of the non-existing hardware. While emulation and simulation should in theory report the same final performance numbers when analyzing a given piece of hardware, only simulation will be able to report detailed and continuous information about the internal processes of the hardware in question. We do not use hardware synthesis, emulation, or simulation techniques in this dissertation, in part because they are not fully accurate when it comes to understand hardware–software interactions [37] or, if accurate, they are prohibitively complex and extremely slow Except in Chapter 5 — which uses dynamic analyses to improve the speed of a certain type of simulation — synthesis, emulation, and simulation are not further discussed in this dissertation.

Dynamic analyses and measurements involve recording information about software as it runs on an existing platform. Since dynamic analysis tracks actual execution, it is the method best suited to capture complex hardware-software interactions, and thus the general method used throughout this dissertation. We devote the next section of this chapter to discussing when, where, how, and for what to use dynamic performance analysis techniques.

2.3 Dynamic Performance Analyses

There are many kinds of dynamic performance analyses for studying computer systems. Rather than try to comprehensively catalogue the many open–source, commercial, and academic measurement and analysis tools, this section's aim is to distinguish different types of analysis techniques based on how, when, where, and what is measured. Later, we explicitly name relevant tools in the "Related Work" sections of individual chapters.

2.3.1 Instrumentation

One distinction that can be made to differentiate dynamic analysis techniques is in *how* measurements are initiated, beginning with whether *instrumentation* is used or not. Instrumentation is the process of dictating system monitoring through code instructions inserted directly into the program being measured. The instruction code may be inserted at many different points, such as:

- within program source code or library code;
- within program binary code;
- within assembly or machine code;
- within a compiler; or
- within an operating system.

The code could also be inserted *by* different agents, such as:

- directly by the programmer (e.g., `printf` calls), with or without the support of programming language–level instrumentation *directives*;
- by a compiler; or
- by a system driver.

The code can also be injected at different times, for example:

- during or after a program is written;

- during or after compilation, either pre- or post-linking; or
- during execution, for example via JIT re-compilation.

One possibly confusing nomenclature detail to note is that when instrumentation is inserted pre-compilation or at compilation or link time, it is said to be *static instrumentation*, whereas if instrumentation is inserted post-compilation at execution time, it is said to be *dynamic instrumentation*. However, the static– or dynamic–prefixes refer only to the time of instrumentation, and both types of instrumentation are still *dynamic* analyses, because in both cases the collection of data will occur at runtime.

Deciding when, how, and where to instrument a program is a delicate tradeoff that must take into account the required efforts of the analysis tool developer, the efforts of the end-user, and the collection goals. For example, instrumentation inserted via dynamic instrumentation directly into binaries is attractive from a user-standpoint, because it requires neither program rewrites nor recompilation. However, dynamic instrumentation tools can require significantly more effort on the part of the analysis tool developer — in part because dynamic instrumentation tools must be specific to the intermediate representation (frequently an ISA) of the programs they instrument, thus requiring not only a detailed understanding of this representation by the tool developers, but also multiple versions of the tools for different representations.

2.3.2 Independent Measurement

When instrumentation is not used, that is, when monitoring instructions are external to the program being monitored, we say that the program is *independently monitored*. Monitors could be a standalone application that reads hardware counters (such as cache miss rates or instructions retired) or operating system statistics (such as number of child processes spawned or percentage of processor utilization). Independent measurement could also be built into some part of the system, including the hardware, the operating system, or the compiler. Finally, independent measurement may be completely external to the computer system being measured, for example on another machine in a distributed network, or with an external meter, such as the "Kill A Watt" electricity monitor that connects to a computer's

power supply to account various statistics such as voltage, current, and power draw.

2.3.3 Collection and Recording Policies

Orthogonal to whether they are instigated by instrumentation or by independent monitoring, dynamic analyses may utilize one of two types of recording policies, *continuous* monitoring or *periodic* sampling. Continuous monitoring, as the name suggests, continues to collect data for the duration of an application's execution. Periodic sampling instead collects data only for short lengths at a time, then pauses for an interval, then collects data again, and repeats. The sampling periods may be punctuated by random– or timer–based intervals, or may be based on machine statistics such as processor cycles or instructions retired. Another analysis policy decision that is orthogonal to the method of measurement is whether data is recorded as *trace* or as a *profile*. The difference between traces and profiles is that traces save detailed consecutive and often time–stamped lists of events (such as an ordered function execution log), whereas profiles summarize the data with aggregate event counts (such as a list of function call frequency counts). While traces can potentially provide users with more information, they may also take more resources to record, store, and post-process.

2.3.4 What to Collect

The options for what information dynamic analyses can be used to track are almost endless. One major distinction that can be made about what kinds of information dynamic analyses collect is whether that information relates to *performance* or *correctness*. Performance information includes quantitative metrics such as runtime, energy, power, device size or hardware area, or system financial cost. Correctness information includes data on the reliability of systems, on program bugs such as data races, on security issues, or on testing–related issues such as test case code–coverage. In a few cases, the two categories may overlap; for example data that shows a program spending significant time in mutex locks may be considered both performance–related (because of the excess runtime incurred) or correctness–related (because of the potential underlying concurrency bug). In line with the goal of identifying and reducing inefficiencies, the dynamic analyses in this dissertation

focus almost entirely on collecting performance information. Some examples of performance information that may be collected include:

- *processor metrics*, such as processing frequencies, cycle counts, instruction execution counts, or instruction mixes (i.e., what proportions of different types of instructions are issued);
- *memory metrics*, such as cache misses, page faults, or I/O bus cycles;
- *synchronization and concurrency metrics*, such as number of locks, time spent waiting at barriers, or the number of active threads or processes;
- *programmatic metrics*, such as function calls or basic block counts;
- *scheduling metrics*, such as response time, throughput, or latency;
- *speed metrics*, such as instructions retired per cycle (IPC), floating point operations per second (FLOPS), processor idle time, or processor utilization rates;
- *power and energy metrics*, such as peak power, minimum power, TDP, or performance per watt.

2.3.5 Dynamic Analysis Pitfalls

There are a couple of downsides or potential problems with dynamic analyses.

Overheads and Perturbation One major potential pitfall of dynamic analyses is *perturbation*, or the possibility that the overheads of the measurement process will affect the actual data being measured. This is especially a concern of instrumentation, given that extra instructions are added to the programs to be measured. To ensure result integrity, perturbation must be carefully monitored, reigned in, and if necessary, corrected. Non-perturbative overheads — those that do not affect the measurements — can also be problematic if they present as significant compilation time or runtime increases. For example, Chapter 5 must deal with the excessive (but non-perturbative) overheads involved in simulating GPUs.

Other Pitfalls There are other precautions one must take to avoid measurement inaccuracies. Due to the nature of dynamic analyses measuring real events on real systems, they can be *unsound*, that is not representative of all possible variations. Nondeterminism in parallel programs, different user inputs, or program–external events such as operating system interrupts, battery power levels, or processor frequency variations can result in different measurement results across program executions. Typically, reasonable accuracy of measurement can be assured with repeated trials (until a statistically significant result is achieved), a varied supply of inputs, and control over as many external events as possible (such as disabling dynamic processor frequency tuning).

Chapter 3

Parallel Block Vectors

The advent of multi-core processing and its subsequent rise is largely a consequence of a decline in transistor scaling. Facing limits to performance per area due to heat thresholds, architects needed a way to deliver speed without excess power consumption. One solution that they came up with is the multicore processor, which substitutes the previous single large, powerful, and power–hungry processing core for multiple smaller, less powerful but also less power–hungry cores. The multiple cores use *thread level parallelism* or TLP — executing instructions simultaneously on multiple cores. There are a few reasons why multiple cores can increase net performance with lower net power consumption; one is that the smaller cores can be run at a lower frequency than the single large core and still deliver faster net processing speeds due to TLP. Sometimes, opting out of the expensive wide superscalar out–of–order (OOO) cores used in uniprocessing (because the *instruction-level parallelism (ILP)* they provide is no longer needed with TLP) further helps multiprocessing come out ahead on the performance–per–power spectrum.

Given the efficiency opportunities that multiprocessing can provide for general purpose computing, the technology seems like a clear win. There is a major downside of multiprocessing, however: writing and debugging the programs that can run on multiprocessors, i.e., *multithreaded* programs, is a challenge and an ongoing area of research. There are many hurdles, including legacy single–threaded code that must be rewritten, and added difficulty in debugging programs that now exhibit *nondeterminism* (different behaviors across different executions). Also, it is difficult for programmers to reason about parallel algorithms,

and some parts of programs simply cannot be parallelized. Finally, even when a program is well-parallelized, overheads often arise, such as the need to share resources or communicate between threads, that prevent full parallel *scalability*, meaning software threads do not fully fill available hardware threads. When software is not as parallel as hardware, the result is potentially massive inefficiencies leading to significant losses to runtime and energy.

To assist in the efforts of tracking down inefficiencies in the form of poor software– and hardware–parallelism matches, this chapter presents a new form of parallel program analysis called *Parallel Block Vector (PBV)* profiling.[1] Existing research and industrial tools analyze parallel performance by combing through program source or thread traces for pathologies including communication overheads, data dependencies, and load imbalances, but this work takes a new approach. PBV profiling ignores any underlying pathologies, and instead records basic block execution profiles per concurrency phase (e.g., the block execution profile of all serial regions of a program). This information provides a direct and fine-grained mapping between an application's runtime parallel phases and the static code that makes up those phases, pointing users to potential inefficiencies in source code. PBVs also uncover opportunities for improved architectural design, for example revealing information that could help architects produce more effective serial-phase accelerators to speed through the portions of programs that cannot be made to effectively utilize multiprocessing.

3.1 Introduction

As multi-cores have come to dominate programmable architectures from mobile to the datacenter, efficiency in parallel programming has seen significant attention from both research and industry. Parallel profilers and measurement tools have helped application parallelization, often by exposing hard to identify parallel performance issues. Intel's VTune [96] and the gprof-based Kremlin [66] are examples of such tools. While these tools are certainly useful to software engineers, they don't capture the whole picture of a parallel program's execution.

[1]This work was introduced in a conference publication [119], and was also discussed in three other publications [120–122].

Parallel block vectors profiles establish a mapping between static basic blocks in a multithreaded application and the degrees of parallelism exhibited by the application each time a basic block executes. These profiles enable the discovery of two previously unseen characteristics of parallel programs: they tease apart serial and parallel portions of a program for individual analysis, and they track the changes in parallelism of fine-grained code regions. A parallel block vector consists of an array of counters where each counter $counter_{b,t}$ indicates how many times basic block b was executed when the application had t threads running. From this profile it is easy to find blocks that executed at a particular thread count t (e.g., all b such that $counter_{b,t} > 0$), or the thread counts each time a particular block b was executed ($counter_{b,t}$ for all values of t).

This chapter shows that with careful engineering effort, parallel block vectors are neither complex nor expensive to gather even at such fine granularity. We demonstrate Harmony, an LLVM compiler pass that instruments a multithreaded application to gather parallel block vectors. For eight Parsec benchmarks, instrumentation using Harmony incurs an average of 16% application slowdown and has minimal resource overhead as measured by register spills and cache miss rates.

The new parallel program characteristics uncovered by parallel block vectors can be used to improve multithreaded execution in a variety of ways. For example in Section 3.4.2, we use parallel block vectors to separate the parallel and serial code portions of several applications, discovering that the instruction mixes for these subsets of code differ, often significantly, from the overall program instruction mix. In the context of heterogeneous processors, such as those analytically motivated by Marty and Hill [85], this information can be applied to tailor heterogeneous cores to better suit their anticipated parallel and serial workloads. This chapter will also show how PBVs can be used to improve parallel software's performance in Section 3.5, by identifying very tiny regions of code that take up the majority of multithreaded execution, through the use of a PBV-descendant metric called *ParaShares*.

In summary, this chapter makes the following three contributions:

- Defines parallel block vectors, a novel way of measuring parallel program performance that can reveal previously unseen multithreaded program features.

- Describes Harmony, a tool that allows fast (only 16% slower than runtime) and accurate collection of parallel block vectors via compiler inserted instrumentation and dynamic profiling.

- Demonstrates four applications of parallel block vectors, discovering that: (1) In many cases the black and white scenario of Amdahl's Law, in which code is either purely serial or purely parallel, does come to pass, with blocks displaying strong affinities for either serial or parallel execution. However, there are also exceptions in which substantial numbers of basic blocks run both serially and in parallel across different executions. (2) Program features, such as instruction mix and basic block size, vary across blocks that can be categorized into different degrees of parallelism. Notably, features of identifiably serial blocks often differ significantly from whole program features. (3) The frequency of execution of a block does not necessarily correlate to parallelism or serialism. This suggests that when "hotspot" analysis is used in the context of processor design, architects should consider parallelism as a factor in their analysis. (4) PBV–produced ParaShare metrics can be used to pinpoint opportunities for reducing software-inefficiencies, allowing users to speed benchmarks with micro–changes that have macro impact to the tune of 14-92% runtime improvements.

The remainder of this chapter discusses these contributions in further detail. Section 3.2 defines parallel block vectors and shows a sample parallel block vector for a simple matrix multiply application. Section 3.3 describes Harmony, a static instrumentation tool to collect the profiles, Section 3.4 uses analysis of parallel block vectors to make our three architectural discoveries, and Section 3.5 discusses the ParaShare metric.

3.2 Parallel Block Vector Profiles

Many profiling tools collect runtime statistics from the perspective of processes or threads [95, 96, 101, 231], reporting the number of threads running for the duration of a process or the breakdown of serial and parallel execution time. Parallel block vectors report on a program's parallel behavior from the perspective of a basic block. A parallel block vector consists of one histogram for each basic block, indicating the degree of parallelism exhibited

by the application each time the block was executed.

Figure 3.1 shows a parallel block vector profile for a simple, unoptimized matrix multiplication program. The profile shown in the table is a matrix with one row for each static basic block and one column for each possible degree of concurrency. In this example, the program created four threads, which in addition to the initial thread, makes at most five concurrent threads. Each cell in the profile gives the number of dynamic executions of the given block at the given degree of parallelism. To help survey large applications, we also use the heatmap visual representation shown by the shading in Figure 3.1.

Parallel block vectors create two new opportunities for better understanding parallel programs. First, they allow identification of specific basic blocks that run at a particular thread count. For example, examining the first column in Figure 3.1 reveals that fourteen blocks make up the serial phases of matrix multiplication's execution, with main:6 and worker:5 dominating the dynamic mix. Second, a user can monitor regions of interest in a program to see the phase or phases in which the code executed. For example, blocks worker:4-6 in Figure 3.1 correspond to the inner multiplication loop, a critical region in terms of performance. As one would hope, the profile reveals that this code is largely executed at high thread counts.

There are multiple ways to count threads when determining the parallel phase of an application. *Nominal thread count* includes all created threads regardless of whether they are running or blocked. *Effective thread count* excludes blocked threads and counts only runnable threads. *Running thread count* includes only the runnable threads that have actually been granted access by the operating system to a processor. We collect profiles for nominal and effective thread counts, but do not count running threads for two reasons. First, running thread count is strongly dependent on the availability of hardware resources and the behavior of the scheduler, thus revealing more about those two aspects of the system than the application. Second, counting running threads requires polling the OS, which is likely to substantially slow and perturb the execution of the program under measurement.

```
#define NDIM 1000
double a[NDIM][NDIM];
double b[NDIM][NDIM];
double c[NDIM][NDIM];

void worker(int me,int p,int n) {
  int    i,j,k;
  double sum;
  i = me;
  while (i < n) {
    for (j = 0; j < n; j++) {
      sum = 0.0;
      for (k = 0; k < n; k++)
        sum = sum+a[i][k]*b[k][j];
      c[i][j] = sum;
    }
    i += p;
  }
}
```

```
int main(int argc, char *argv[]) {
  // Variable declaration and
  // initialization ommitted
  n = // number of threads, here 4
  threads = (pthread_t*)
    malloc(n*sizeof(pthread_t));
  pthread_attr_init(&pthread_custom_attr);
  for (i = 0; i < n; i++)
    pthread_create(&threads[i],
                   &pthread_custom_attr,
                   worker, ...);
  for (i = 0; i < n; i++)
    pthread_join(threads[i], NULL);
  free(arg);
  return 0;
}
```

Nominal Thread Count

	1	**2**	**3**	**4**	**5**
main:9	0	0	0	0	1
worker:7	0	14	14	16	956
worker:6	606	146K	12K	16K	955K
worker:5	607K	14.6M	12.8M	16.1M	955M
worker:4	607	146K	12K	16K	955K
worker:3	1	14	14	16	955
worker:2	0	1	1	1	1
worker:8	0	1	1	1	1
worker:1	1	1	1	1	0
worker:0	1	1	1	1	0
main:7	3	0	0	1	0
main:6	1M	0	0	0	0
main:5	1K	0	0	0	0
main:4	1K	0	0	0	0
main:3	1	0	0	0	0
main:2	1	0	0	0	0
main:1	1	0	0	0	0
main:0	1	0	0	0	0

Figure 3.1: **Parallel block vector for matrix multiplication.** For each basic block in an application, top, the profile, bottom, indicates the block execution frequency at each possible thread count (i.e., degree of parallelism).

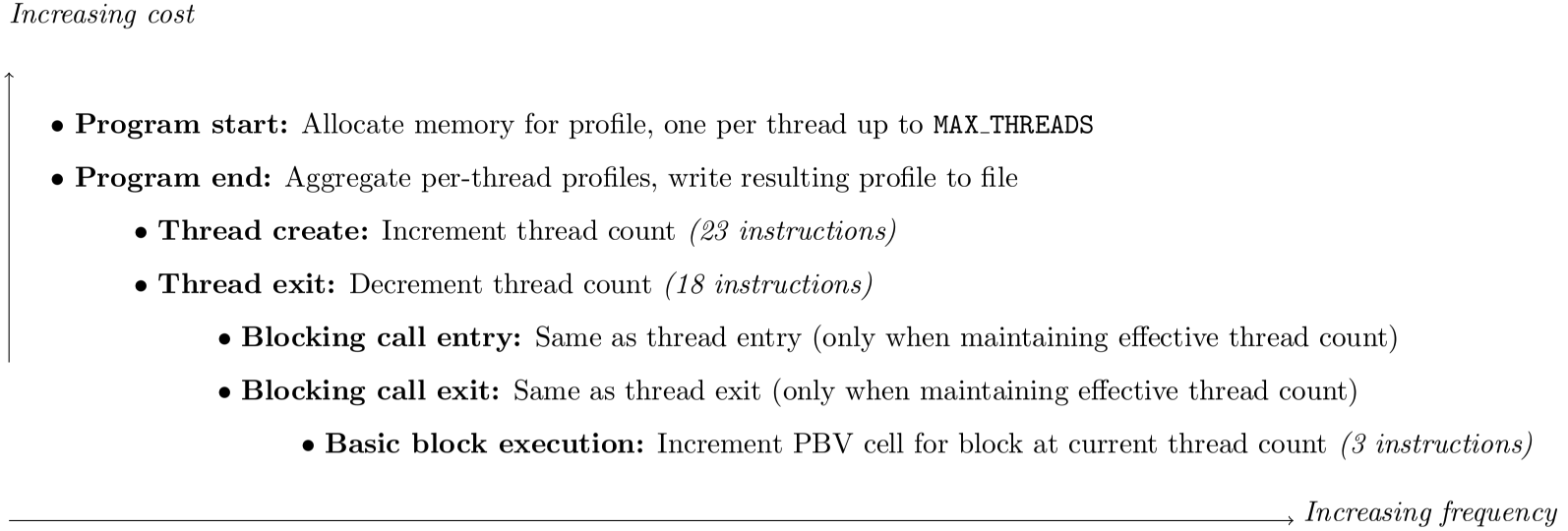


Figure 3.2: **Harmony instrumentation points.** Profiler action is taken upon various runtime events. Careful engineering offloads expensive work to the least frequent events, in particular program start and finish which do not overlap with the execution of the program itself. This results in minimal profiling work at the most frequent events (i.e., basic block executions), reducing the profiling overhead and minimizing perturbation.

3.3 Harmony: Efficient Collection of PBVs

We now describe Harmony, an instrumentation pass for LLVM [137] to generate parallel block vectors. We selected compile-time instrumentation for Harmony for three reasons. First, parallel block vectors require dynamic information such as basic block execution frequency, thread count, and timing information which is not available via static analysis. Second, unlike dynamic instrumentation frameworks such as Pin, compile-time instrumentation adds no additional runtime overhead beyond the instrumentation code itself. It is particularly important to keep overheads low when profiling parallel applications as shifts in the relative timing of events can perturb the behavior of the program. Finally, with compile-time instrumentation, portability comes for free, making it trivial to collect profiles on any architecture or language supported by the compiler.

This section describes the architecture of Harmony and discusses the efforts undertaken to minimize profile collection overhead thereby maintaining profile accuracy. The pass is intended to be the last pass executed, after the program has been fully optimized and the final program control-flow graph (CFG) has been set. Harmony is available as an open-source tool at `http://arcade.cs.columbia.edu/harmony`.

3.3.1 Injecting Instrumentation

To collect parallel block vectors, Harmony must take action at several program events, as summarized in Figure 3.2. At program start the profiler must allocate and initialize a profile, and at program finish the profile must be written to a file. At thread creation and exit, Harmony must inject code to increment and decrement the nominal thread count. When tracking effective thread count, the counter must also be decremented upon entry and incremented upon exit from any blocking call, such as a lock acquire. Lastly, each basic block execution must be accompanied by an increment of the appropriate entry in the profile matrix.

Harmony injects instrumentation in two different ways. For tracking basic block executions, Harmony adds instructions directly into the body of a basic block, as illustrated in Figure 3.3. The same goes for program entry and exit, where Harmony inserts calls to

profile initialization and cleanup routines (not shown). For the remaining events, Harmony interposes on relevant thread library calls as illustrated in Figure 3.4. At present, the tool supports only Pthreads library calls, and requires only that programs include `harmony.h` in place of `pthreads.h`.

3.3.2 Strategies for Minimizing Perturbation

Adding instrumentation to a parallel program risks perturbing program behavior, potentially compromising the accuracy of the profile. While some perturbation is unavoidable, we found that careful engineering significantly reduces the overhead of profile collection.

As basic block executions are by far the most frequent event the profiler instruments, we focused our optimization efforts there. Each time a basic block executes, the instrumentation must read the current thread count and use that value along with the basic block ID to index the profile matrix and increment one counter (i.e., `profile[currentThreadCount][bbid]++`). Harmony takes the following steps to streamline this computation:

- Because `bbid` will be changing much more frequently than `currentThreadCount`, the profile matrix is laid out in a cache-friendly, column-major fashion that places profile entries for different basic blocks at the same degree of parallelism at adjacent addresses in memory.

- Significant portions of the address calculation are factored out of the basic blocks themselves. Specifically, the column address offset for the current thread count need only be re-calculated each time the thread count changes and not for each basic block execution. All that remains of the address calculation for each basic block is to compute the offset within the profile column.

- Finally, because the target programs are parallel, multiple threads will be updating the profile concurrently. Rather than guarding each counter in the profile matrix with a lock, which would introduce substantial synchronization overhead, we allocate a private profile matrix for each thread, and aggregate the per-thread profiles only after the program has finished executing.

```
void sample(uint32_t bb_id) {
  bucket_t *b = *ptr_specific_col +
              bb_id*PROF_BUCKET_SIZE;
  (*b)++;
}
```

```
# load the pointer to pointer to my column
movl %gs:ptr_specific_col@NTPOFF, %edx
# load the pointer to my column
movl (%edx), %edx
# increment counter for BBL1 at specific_col+4
incl 4(%edx)
```

Figure 3.3: **Direct instrumentation example.** Each basic block is augmented to record its execution at the current degree of parallelism. The additional three instruction use only one register and do not induce any register spills.

```
// intercept potentially blocking call
#define pthread_mutex_lock(a...) \
    BLOCKING_CALL(pthread_mutex_lock(a))

// effective thread count drops on entry
// and rises on upon completing
#define BLOCKING_CALL(exp) ({ \
    int rv; \
   __sync_sub_and_fetch(&(effectiveThreadCount), 1); \
    rv = exp; \
   __sync_add_and_fetch(&(effectiveThreadCount), 1); \
    rv; })
```

Figure 3.4: **Thread library wrapper example.** Here the instrumentation decrements and increments the effective thread count upon upon entry to and exit from of a blocking call respectively.

Collectively, these optimizations result in the small per-basic block overhead of the three instructions shown in Figure 3.3.

3.3.3 Mapping Profiles Back to Application Code

To ensure that profiles can be mapped back to the original application code, Harmony annotates both the profiles and the LLVM assembly file with unique basic block IDs. In post-processing these two files can be cross-referenced for further analysis as in our instruction mix case study (Section 3.4.2). Though we do not implement it for these studies, this labeling scheme could be coupled with debug symbols to link the profile all the way back to source code.

3.3.4 Runtime Impact of Harmony

Dynamic analysis risks altering the timing, and with it the behavior, of a parallel program in a way that may compromise the accuracy of the gathered information. For example, slowing critical sections will increase lock contention, and, conversely, slowing non-critical sections will reduce lock contention. It is thus important to carefully examine profiling's impact on the original program.

In his 1991 chapter, *Event-based Performance Perturbation: A Case Study*, Allen Maloney [151] listed the three primary sources of program perturbation: execution of additional instructions and their resulting execution slowdown, changes in memory references patterns, and register pressure. As outlined in Section 3.3.2, Harmony takes a number of steps to minimize the impact on program behavior. In this section we evaluate the profiler's impact on each of these three metrics.

3.3.5 Execution Time Overhead

First, we compare the execution time of applications compiled with and without Harmony's instrumentation. In both cases the `-O3` flag is set to turn on maximal compiler optimizations. The machine used for these experiments and those described later in Section 3.4 has 4 2.0 GHz cores, 3.3GB of RAM, and is running Linux Ubuntu version 8.04. We use Harmony

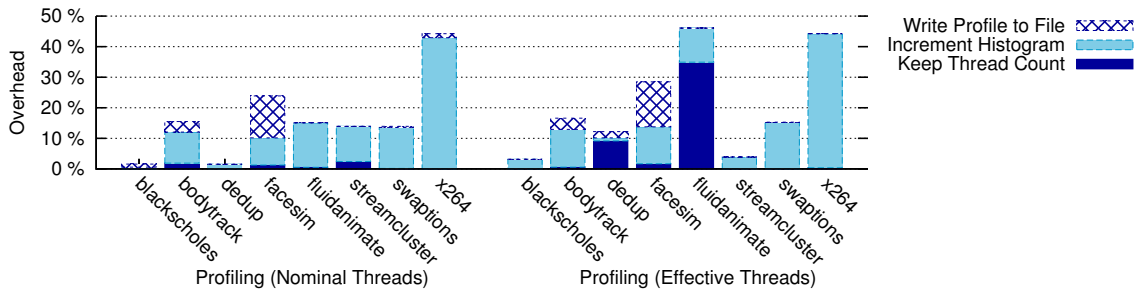


Figure 3.5: **Low overhead of instrumentation.** Program slowdown due to profile collection ranges from 2% to 44% with an average overhead of 18%.

.

to collect parallel block vectors for eight applications from Parsec [21], a suite of non-HPC multithreaded benchmarks.

All runtimes are the average of 20 program runs. For each application the profile collection times were normalized to an uninstrumented baseline. Figure 3.5 plots the profile collection overheads. Nominal thread count profiling added 16% on average while effective thread count profiling added slightly more overhead at 21%. The additional overhead is expected due to the additional thread counter activity. As Figure 3.5 indicates, 3% of these totals are attributable to time spent writing the profile to a file after the program has finished. Thus, the effective overheads *during program execution* are 13% and 18%, respectively.

Relative to similar tools, these overheads are modest. For example, ThreadScope, a tool for tracing runtime parallel events using the Haskell GHC compiler [111], incurs 10%-25% overhead. Quartz, a gprof like tool that uses sampling to monitor threads, increases program run times by 70% [3]. The popular (but heavier-weight) runtime binary instrumentation platform, Pin, incurs a $100 - 400\%$ increase in execution time for basic block counting alone [10].

It is interesting to note that two applications, `dedup` and `fluidanimate`, spend significantly more time maintaining an effective thread count than maintaining a nominal thread count. This differential in activity between the two counters becomes significant when we compare the resulting profiles later in Section 3.4.1.

3.3.6 Storage Resource Contention

The last two sources of perturbation in Maloney's list address increased resource pressure caused by instrumentation. For Harmony we see a slight — 7.5% on average — increase in register spills. However, for these applications, the additional spills were confined to the profile setup and cleanup activities which occur prior to and after the execution of the program itself. Most importantly, the instrumentation code in each basic block did *not* induce spills.

As measured by Cachegrind [244], the instrumentation introduces negligible cache perturbation. In the L1 instruction and data caches the miss rate increased by at most 0.06% and 0.2% respectively. There was no measurable impact on the hierarchy beyond the L1 structures (i.e., L2 miss rates were unchanged).

3.4 Architectural Design Applications of PBVs

We now carry out three novel analyses of our benchmarks, each enabled by parallel block vectors. Figure 3.6 shows visual representations of the profile of each of the eight Parsec benchmarks. Recall from Figure 3.1 that each row corresponds to a static basic block and each column to a nominal thread count. For space reasons we show the full heatmaps only for nominal thread counts, though in the following sections we will analyze both nominal and effective thread count profiles.

From these profiles, we see that in several applications (namely `bodytrack`, `dedup`, `facesim`, `fluidanimate`, and `streamcluster`) basic blocks display a strong affinity for either serial or parallel phases. The remaining applications (`blackscholes`, `swaptions` and `x264`) by contrast have significant portions which execute at mixed thread counts, during both serial and parallel phases.

It is well known that the serial portions of an application limit parallel speedups [85], but what exactly do those serial portions look like? Are they amenable to acceleration? We will explore these questions in the following sections, before closing with a discussion of other applications of parallel block vectors.

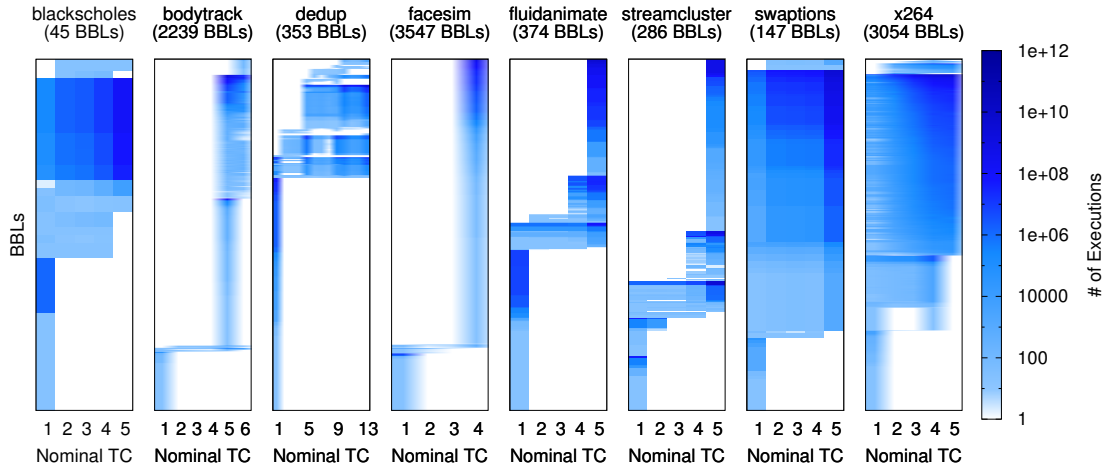


Figure 3.6: **Parallel block vectors for Parsec.** These heatmaps are a visualization of the profiles produced by Harmony. For the given application, they show the number of times (shading) each static block (row) was executed at each degree of parallelism (column).

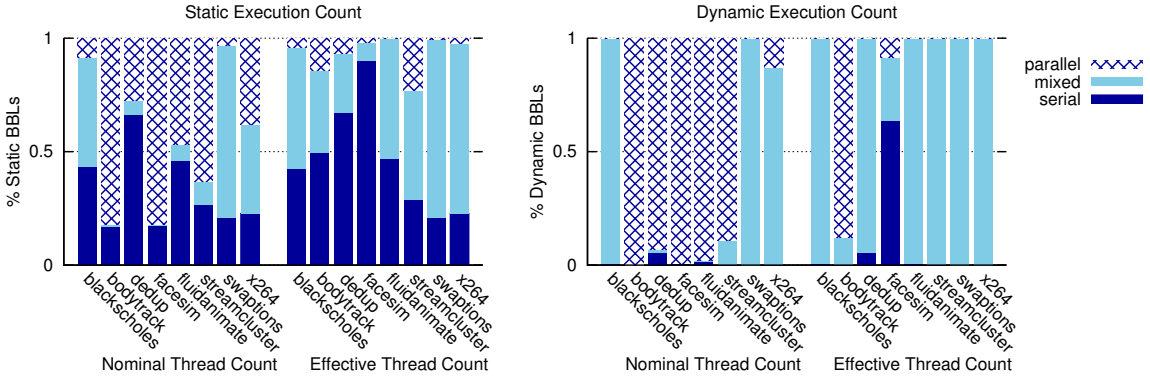


Figure 3.7: **Classifying basic blocks by parallelism.** These graphs show the percentage of blocks which execute only serially (serial), blocks which execute both serially and in parallel (mixed), and blocks which only execute in parallel (parallel) for each application, for both nominal and effective thread counting, and for both static and dynamic block executions.

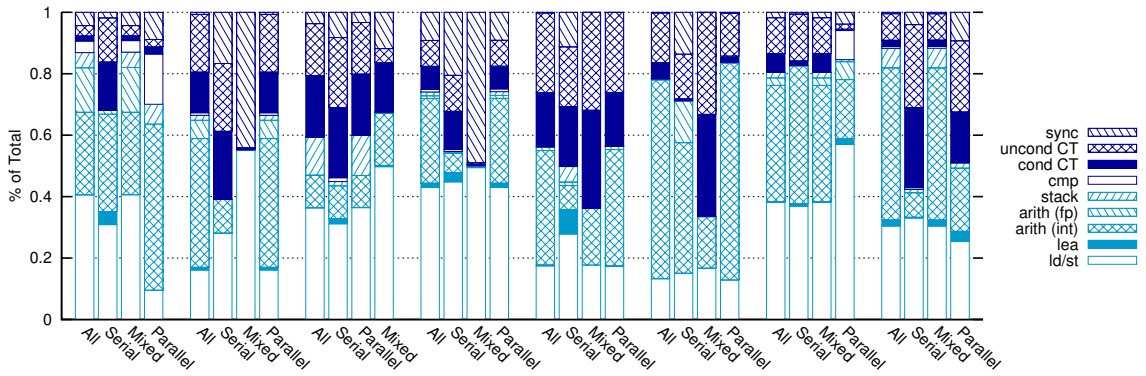


Figure 3.8: **Opcode mix by class.** Instruction mixes for the entire program compared with the mixes for each basic block class (serial, parallel, and mixed). In all applications, the instruction mixes for both purely serial and purely parallel blocks differ significantly from whole program mixes.

3.4.1 Serial and Parallel Application Partitions

Knowing how much of a program runs in parallel and how much runs serially is useful for many purposes. Tools such as Intel's VTune Amplifier XE [96] identify the serial fraction of an application's runtime so that software engineers can improve the parallelization of their programs. Such metrics are also useful when estimating the scalability of a particular parallelization according to Amdahl's Law [85].

Parallel block vectors make it possible not only to quantify the serial portion of a program, but to map that region back to the specific basic blocks that comprise it. To get this information we classify each basic block into one of three categories: *serial* (i.e., never executed with a thread count greater than one), *parallel* (i.e., always executed with a thread count greater than one) or *mixed* (i.e., sometimes executed in serial regions, other times in parallel regions).

From an architect's perspective, the pure serial blocks make natural targets for specialized serial processors (further discussed in Section 3.4.2) or accelerators (Section 3.4.3). The mixed blocks, which run both in parallel and serially, are likely of interest to all system designers. They might represent areas in the application where there were communication overheads or other forms of architectural resource contention. Identifying the mixed blocks allows their execution to be improved with better scheduling algorithms, additional

hardware resources, or code transformations.

Figure 3.7 shows the breakdown of static and dynamic basic blocks by class (serial, mixed, or parallel). We observe that significant portions of several applications are neither purely serial nor purely parallel, but rather belong to both regions (the mixed class). This is true of both nominal and effective thread counts. The trends are similar but even more pronounced when counting dynamic basic block executions. This means that when we talk about Amdahl's law and serial and parallel phases of a program, those phases often *do not* correspond to different portions of the application. One hypothesis is that such blocks are the result of library code which is called both from the serial and parallel phases.

Returning to the serial/mixed/parallel classifications, we can also clearly tell which applications are the most parallel. For example, from the static nominal view, `bodytrack` and `facesim` seem to be equally parallel. However, from the dynamic effective profiles, we see that `bodytrack` has more blocks actually running in parallel, whereas `facesim` apparently suffered from blocking threads and its parallel blocks were less frequently executed than its serial blocks. In the following section, we will look in more depth at the content of blocks in each of these classes.

3.4.2 Program Features by Degree of Parallelism

Recent interest in heterogeneous multicore architectures spans not only the architecture community but operating systems, high-performance computing, programming languages, and others [7, 75, 98, 127, 176, 206, 268]. The principle idea behind heterogeneous processing is specialization: different cores on a heterogeneous machine can address the varied compute needs of modern workloads while maximizing hardware performance and efficiency. For example, when portions of a program cannot be adequately parallelized, an aggressive, out of order, no holds barred processor might be employed to reduce execution time.

If heterogeneous cores are meant to address the specialized needs of certain portions of the application, it is naturally important to understand what these processors should be specialized to. Parallel block vectors can assist by distinguishing features of parallel and serial phases. For this analysis, we will continue to use the *always serial*, *always parallel*, or *mixed* classification introduced in Section 3.4.1 in which every block belongs to exactly one

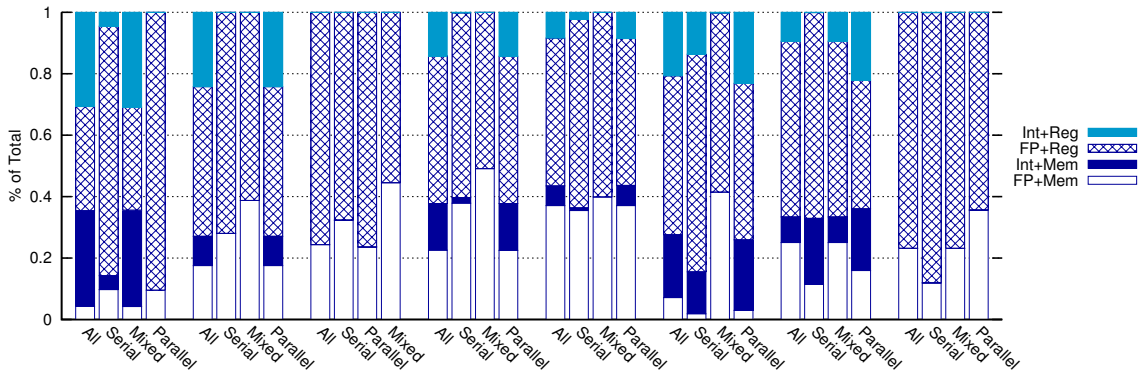


Figure 3.9: **Memory interaction by class.** The proportion of memory operations for serial and parallel basic blocks differ from the proportion in the program as a whole.

class.

Figure 3.8 compares the dynamic instruction mixes of each of these three categories, as well as for the program as a whole. All of the X86 opcodes that occurred in the application were classified into one of eight categories: loads and stores, loads of effective addresses, integer arithmetic, floating point arithmetic, comparisons, conditional control transfers, unconditional control transfers, and synchronization.

We observe that for most applications, serial basic blocks display significantly different instruction mixes from the overall program. This indicates an opportunity for architects to exploit, when designing the microarchitecture of aggressive cores for heterogeneous CMPs. Consider the `blackscholes` application. Across the whole program, floating point operations account for more than 20% of the dynamic instructions. If this were the only instruction mix considered, as is currently the case, then the aggressive processor for serial regions might waste space and expense unnecessarily on floating point units, when we can see from the graph that the serial blocks actually require fewer floating point operations than the program as a whole. Instead, the serial phases of `blackscholes` have a higher concentration of control and integer arithmetic, suggesting that resources would be better spent on the branch predictor, for example.

The data in Figure 3.8, shows such a pattern in each of the benchmarks. In every case, either the serial or parallel portions (and sometimes both) have substantially different

instruction mixes than the application as a whole. However, across these applications, there does not appear to be a consistent pattern of *how* the instruction mixes change. For example, the serial portions in `blackscholes` had reduced need for floating point units, while the serial portions of `x264` show increased rates of unconditional control transfers. It is not immediately obvious how hardware can or should exploit such patterns. We believe that this direction merits further investigation, starting with a more comprehensive review of the application space.

Just as opcodes vary, the state upon which the serial and parallel portions of a program operate varies relative to the overall program. Figure 3.9 shows the memory interactions of the three parallelism classes. As with opcodes, the component parts of the application show different mixes than the application as a whole.

3.4.3 Hotspot Analysis Using Parallel Block Vectors

The previous section suggests an approach for using parallel block vectors to determine the applicability of a specialized processor to particular code regions. An extreme form of specialized processor, accelerators have shown great promise in reducing power, saving space in embedded systems, and improving performance for target programs. The following case study explores how Harmony can help architects quantify the potential performance gains of their accelerator designs, in particular how parallel block vectors can enhance hotspot analysis for parallel applications. Hot basic block analysis has traditionally been used for a variety of purposes, including JIT translation [241], garbage collection optimizations [92], simulation points analysis [192], code cache management [218], and parallel performance debugging, for example, in Intel's VTune Amplifier [95].

Figure 3.10 plots *average degree of parallelism* against dynamic basic block executions for block in each application. The average degree of parallelism of a block is simply the average thread count for each block weighted by the block's execution frequency at each count. The scatter plots reveal that the hottest blocks are not always the most parallel ones. In `streamcluster` for instance, many of the hottest blocks have an average degree of parallelism of one. Generally, the hottest blocks seem to be split between blocks which execute exclusively serially and blocks which execute at or near the maximum degree of parallelism.

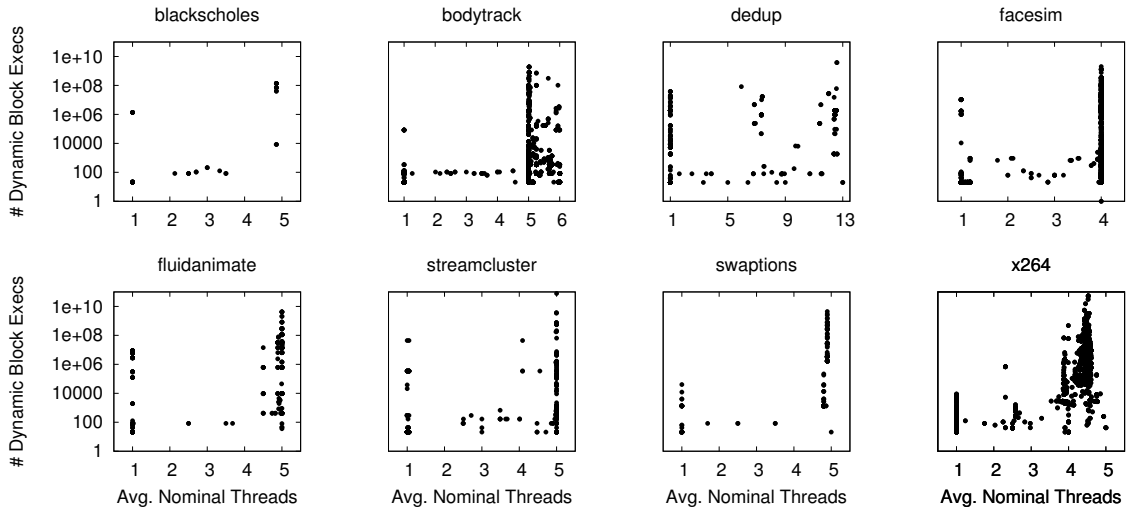


Figure 3.10: **Hottest blocks are not always the most parallel blocks.** Each static basic block's weighted average nominal thread count was calculated and then plotted against its total number of dynamic executions. The graphs show that the hottest blocks are primarily split between those that execute only serial and those that execute near the max degree of parallelism.

This data indicates that not only are there hotspots, possibly amenable to acceleration, but that one should not assume anything about whether the hotspots belong to parallel or serial phases. Some code simply cannot be parallelized. As multicore architectures scale to larger core counts, these serial portions of runtime dominate total execution times. The acceleration of serial sections then becomes critically important. So, as a special case of hotspot analysis, we look more closely at the serial blocks, and ask the question, *are serial code segments amenable to targeted accelerator optimizations?*

Taking the serial basic blocks identified in Section 3.4.1 (see Figure 3.7), we attribute dynamic serial execution frequencies to different percentages of the serial blocks. Figure 3.11 (left) shows that for six of the eight applications, 75% of the serial execution is attributable to less than 10% of the basic blocks. This data corroborates what other projects [250] have seen, that accelerators can effectively accelerate the serial parts of a parallel application.

Processor designers might also be interested in how amenable parallel blocks are to targeted hardware optimizations. Figure 3.11 (right) shows the execution coverage of parallel phases by purely parallel basic blocks. With the exception of `blackscholes` and `swaptions`,

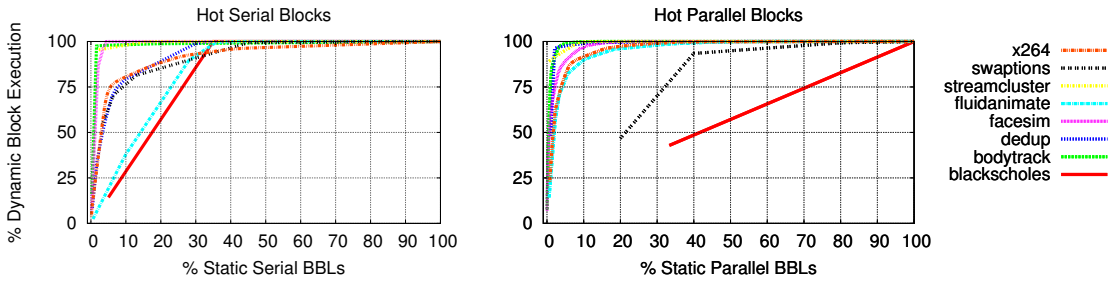


Figure 3.11: **Few basic blocks represent large portions of serial and parallel runtime.** For basic blocks that were determined by parallel block vectors to always execute serially (left) or in parallel (right), percentages of runtime execution are attributed to static basic blocks. For most applications, a small number of blocks represents a large fraction of the total runtime.

approximately 5% or fewer of parallel blocks are responsible for 75% of dynamic parallel blocks. The reason that `blackscholes` and `swaptions` do not show a steep hotspot curve is that they had very few parallel blocks to begin with; three and five, respectively. As with serial code, we find that parallel parts of the applications exhibit pronounced hotspots.

In the above experiments, we examine hotspots in terms of basic blocks, but only because this was the most natural first choice given that it matched our profile granularity. We note that similar experiments can easily be run at hot function or hot instruction granularity if we statically analyze the basic blocks and source program after running Harmony. It would also be possible, with some additional effort, to map hot call graph or dataflow paths to parallelism.

3.5 Pinpointing Software Performance Issues with PBVs

ParaShares identify very tiny regions of code that take up the majority of multithreaded execution, and they are purposefully agnostic to the type or cause of underlying performance pathologies. Their goal is to precisely point programmers to the lines in their program that would benefit most from optimizations. A ParaShare is a rankable score that measures each basic block's share of a total parallel program's execution. The rankings are similar to hot block analyses that report the most frequently executed basic blocks and their CPU use. However, ParaShares factor in the degree of program parallelism at each block execution, providing a more accurate reflection of a block's contribution to execution time. The weighting scheme downgrades the importance of blocks that execute during highly parallel program phases. As a result, it ranks blocks that mostly run during serial phases, and thus tend to consume a greater fraction of runtime, relatively higher in importance. As a program executes, some blocks execute frequently and others may execute rarely or not at all. The frequently executed blocks are called "hot" and are important optimization targets as they constitute a large share of an application's dynamic work.

Figure 3.12 illustrates how ParaShare ranking works. On the left, a program trace highlights the execution patterns of two blocks of interest, A (gray) and B (black). For simplicity, we assume that both blocks have the same number of instructions and equal execution times, though in real ParaShares we do not assume this. Simple counting reveals that B executes 9 times whereas A executes only 4, giving B a higher rank of importance. However, A may consume more of the program's execution time because its executions

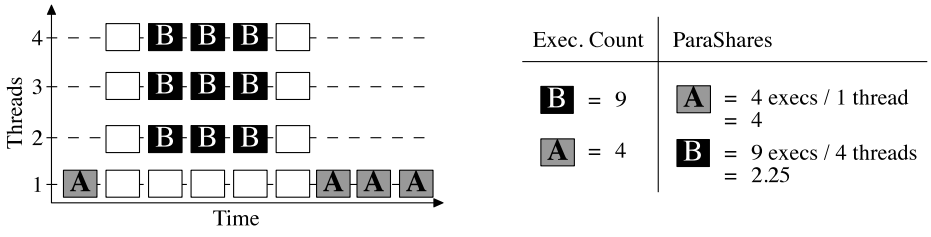


Figure 3.12: **ParaShares rank basic blocks to identify those with the greatest impact on *parallel execution***, weighting each block by the runtime parallelism exhibited by the application each time the block was executed.

occur during serial phases of the program. To account for this nuance, ParaShares divides the executions by the degree of parallelism at execution time, in this example dividing B's 9 executions by the 4 threads that ran while B executed, and dividing A's 4 blocks by 1 for the single running thread. As a rule, parallelism is counted at the start of a basic block's execution to resolve any overlaps in block executions between threads. The resulting scores capture parallel execution shares more effectively, and in this case rank A and B in the opposite order of importance versus traditional execution counts.

3.5.1 Collecting ParaShares

From a user's perspective, ParaShares are straightforward to collect. They require recompilation, a single program run with the usual inputs and outputs, and the execution of a post-processing script. Here are the steps required to collect ParaShares:

Step 1. Compile the source program with Harmony.

Step 2. Execute the program once to collect a PBV.

Step 3. (Optional) Tune machine specific parameters. Optionally, ParaShares can incorporate machine specific instruction weights to account for differences in opcode processing or memory access times. If used, these weights should be stored in a dictionary mapping instruction types to latency factors. Opcode-dependent latency factors are often already available online; for example, latency factors for our machine are available in [72]. These latency factors suggest, for example, multiplying conditional operations by two, add instructions by one, and divide instructions by 30. Due to the overwhelming significance of total instruction count, our applications' ParaShare rankings showed minimal sensitivity to these latency factors. However, latency factors could have more of an effect for other applications and architectures.

Step 4. Calculate weighted, per block static instruction counts. Next, the total (possibly weighted) dynamic instruction count per basic block is calculated. The instruction contents of each block are available in the annotated assembly file produced earlier by Harmony. With weighting, a sum of the weights of each instruction in the block produces a total block weight ($Weight_b$). As an unweighted alternative, a simple count of the instructions per block suffices.

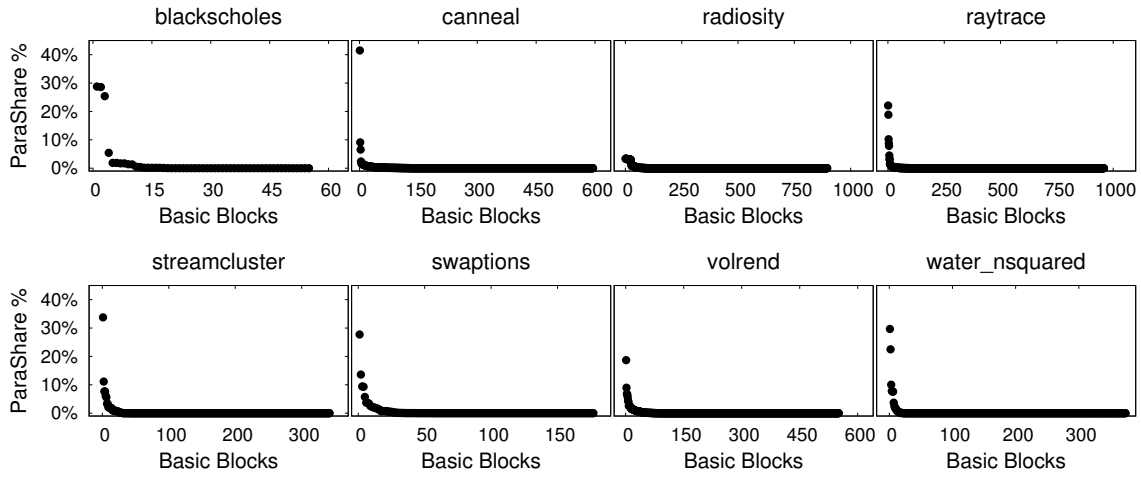


Figure 3.13: **ParaShare rankings identify important blocks to target for multithreaded performance optimizations.** These graphs show the ParaShare percentages (ordered from greatest to least share) of all the basic blocks in eight benchmark applications.

Step 5. Calculate ParaShare rankings. The ParaShare for each block b is computed using the block's static instruction weight and dynamic thread weight. Specifically, the sum of each block's executions at thread count t ($Execs_{b,t}$) are divided by t. This formula is related to the runtime calculation used in Quartz [3], but we apply it here at a much smaller granularity and for a different purpose. The ParaShare of block b is the product of this dynamic thread weight and the static block weight:

$$ParaShare_b = \sum_{t=1}^{max} \frac{Execs_{b,t}}{t} \times Weight_b$$

As necessary for further analysis, the absolute ParaShare for each basic block can be normalized to the program's total ParaShare (the sum of ParaShares across blocks).

Step 6. Use the ParaShare rankings for performance optimizations or other analyses. Finally, ParaShares can be mapped back to the source code via compiler debug information in the assembly code.

3.5.2 Using ParaShares

Figure 3.13 gives a first look at ParaShare block rankings for real applications—eight programs from the Parsec Version 3.0 [21] and Splash-2 [257] benchmark suites, namely `blackscholes`, `canneal`, `radiosity`, `raytrace`, `streamcluster`, `swaptions`, `volrend`, and `water_nsquared`. The Splash2x variant of Splash that is packaged with Parsec was used for its provision of multiple input sets. All of the applications are written in C and C++ and parallelized using pthreads with a variety of design patterns, including a mix of data and task parallelism. Each program was run using 24 threads and native input set sizes on a Dell PowerEdge R420 server. The server is dual socket with Intel Sandybridge E5-243 chips, each with six cores and two-way hyper-threading for a total of 24 effective cores. The system has 24GB of DRAM and runs Ubuntu 12.04.2 with the 3.9.11 version of the Linux kernel. The graphs show that just a few basic blocks (on the x-axis) per program dominate the ParaShare rankings (on the y-axis). The small number of important blocks is no surprise, however ParaShare's ability to find the correct important blocks makes it possible to massively improve program performance with just minor code changes, as demonstrated later in Section 3.5.2.

Benefits of Fine Granularity The well known 90-10 rule of thumb says that 90% of program execution time resides in just 10% of code. For our benchmarks, the rule holds: functions that consume roughly 90% of the execution represent 2.3-17.3% of the lines in the overall programs, or an average of 7.7%. Table 3.1 shows the exact code line counts per benchmark, as well as line counts for the functions consuming 90% of the execution based on ParaShare computations.

The table also shows the number of lines of code contained in the basic blocks that are responsible for 90% of the ParaShare execution. The differences in line counts, particularly for the scientific benchmarks with lengthy functions, strongly motivate the use of basic block granularity over function granularity for examining hot spots. By examining block-granularity hotspots rather than function granularity hotspots, programmers can save themselves from looking at an average of 289 lines per benchmark. In fact, basic block hotspots enough that we could coin a new 90-2 rule of thumb, because 90% of the parallel

Benchmark Application	**Total Lines**	**90% Exec By Func Lines**	**90% Exec By Block Lines**	**50% Exec By Block Lines**
blackscholes	564	68	34	21
canneal	1362	204	70	6
radiosity	11836	276	42	4
raytrace	10963	431	51	8
streamcluster	2539	439	12	5
swaptions	1550	359	28	10
volrend	4227	585	133	89
water_nsquared	2079	338	29	18

Table 3.1: **A case for fine-grained identification of performance inefficiencies.** To examine the functions that take up 90% of the parallel execution, a programmer must examine an average of 338.5 lines per program. To examine the basic blocks that consumed the same amount, they would need to look at an average of only 50 lines per program.

execution is taken up by just 2.4% of the program source lines according to our precise ParaShares analysis. The top 50% of program execution could be covered by searching an even more targeted set of code; programmers would only need to look at 20 source lines per application, or 1% of the overall program lines. The block versus function savings is particularly important when examining unfamiliar applications with lengthy functions and lots of loops. For example, volrend has one function with three sets of doubly nested loops, and we found more than a few instances where a single function contained four or more loops.

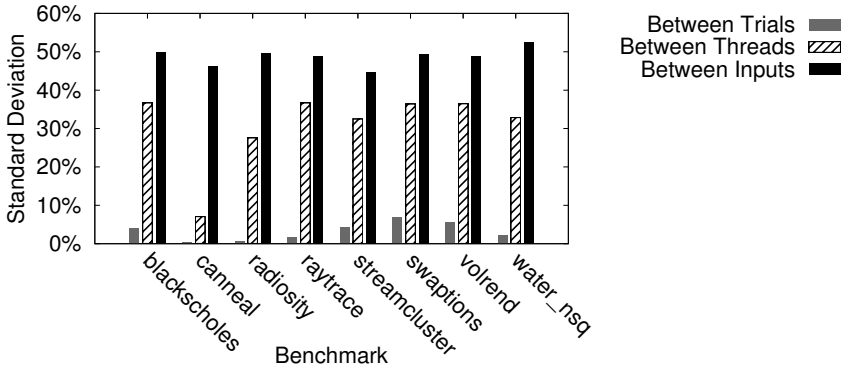


Figure 3.14: **Robustness of the metrics.** Runtimes and basic block execution counts can change across program trials, but the differences are small relative to differences in ParaShares collected across varying thread counts or input sizes.

ParaShare Robustness A program's parallel behavior may be inconsistent across runs, changing block execution counts or overall program runtime. Despite these variations, a single profiling run can produce representative ParaShares, particularly if the purpose of collection is to examine and optimize the hottest blocks with the highest ParaShares. Figure 3.14 plots the standard deviations of program total ParaShares as a fraction of the maximum program total ParaShare across three trials. The standard deviation across runs with the same thread count and input was never more than 7% and averaged only 3.2%. This variation is small when compared with variations between trials given different maximum thread counts (31% on average) or different input sizes (48%). In addition to the magnitude of the overall program ParaShare staying consistent between trials, the ranking

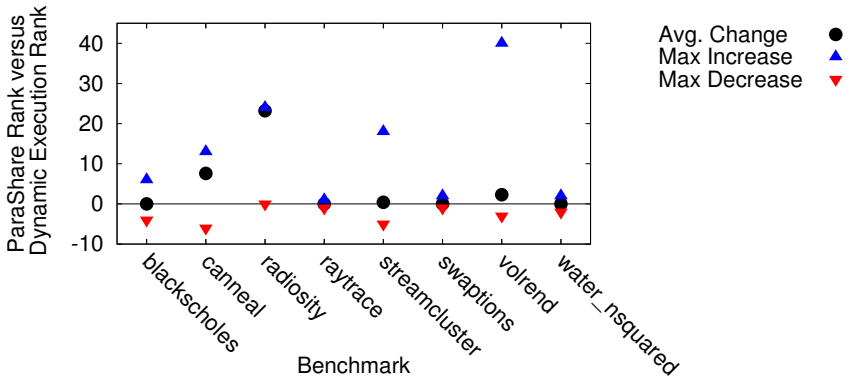


Figure 3.15: **ParaShares versus unweighted rankings in top 20 blocks.** ParaShares do not always highlight new 'hot' blocks, but can often significantly impact the relative importance of a block versus dynamic instruction count rankings *not weighted by parallelism*.

of individual basic blocks varies minimally, and only changes in lower ranked blocks with ParaShares of 2% or less.

Impact of ParaShare Weights ParaShare's utility is not just to locate small regions of significant source code, but to locate significant code that other tools may not highlight. Figure 3.15 shows differences in the top 20 blocks identified by ParaShares versus by dynamic instruction counting that is *unweighted by parallelism*. For a few of the applications (raytrace, swaptions, and water_nsquared), instruction count dominates parallelism and the difference in rankings is negligible. For others, the difference in rankings is profound. For example, one of the top 20 blocks in volrend moves up 40 places in ParaShare rankings versus dynamic execution rankings. In radiosity, the average shift in rankings between the two profiling methods is over 23 places per top 20 block.

Performance Tuning Using ParaShares to target particularly important lines of source code, we made extremely simple and short source code changes to reduce application runtimes by 14-92%. Figure 3.16 shows the effect of optimizations to blackscholes, streamcluster, and swaptions. Both optimized and unoptimized versions were compiled with LLVM's `-O3` optimization set. The optimizations improve computation time, but do not make any algorithmic or parallelization changes. As a result, the savings shrink as thread counts

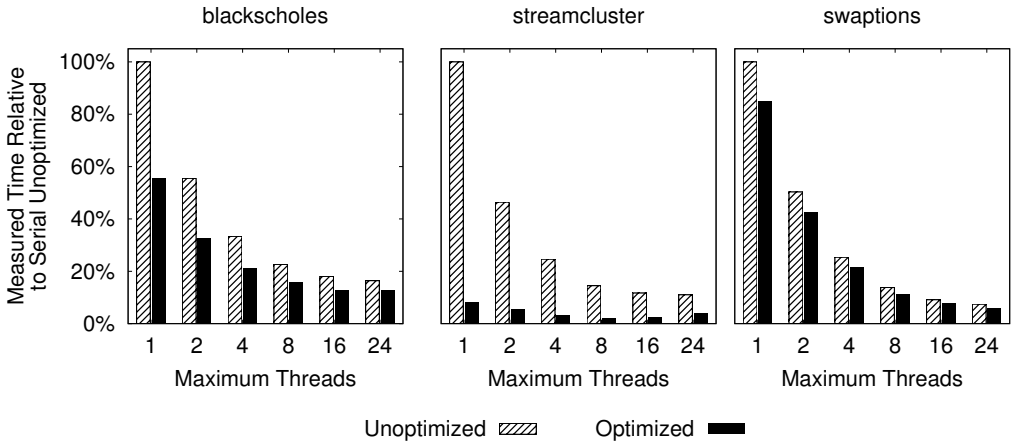


Figure 3.16: **ParaShares pinpoint inefficiencies that lead to significant opportunities for optimization.** With the extremely targeted profiling provided by ParaShares, we were able to improve benchmark performance by up to 92% through source code changes less than 10 lines long.

increase, but they remain significant (up to 82%) even at large thread counts.

In blackscholes, the top two blocks consume nearly 60% of the overall runtime given 24 threads and native input set sizes. These blocks are found in the kernel function that calculates financial option values. By collapsing the original 20 temporary variables in the function to 3, we alleviated register pressure resulting in a 44.6% performance improvement at one thread and 22% at 24 threads. For streamcluster, the top blocks are found in the `dist()` function, which computes the squared Euclidean distance between two `Point`s, each of which is a struct with pointers to arrays of `float` coordinates. Inspecting the line of code in question (the body of a nested loop), we guessed that the compiler missed an opportunity for common subexpression elimination, then modified the code to force it to do so. This change halved the loop body's original four array lookups and two subtractions and reduced register pressure, saving 92% of the serial runtime and 64% of the 24 count runtime. Finally, the top blocks in swaptions correspond to a few nested loops in the `HJM_SimPath_Forward_Blocking.cpp` file. We experimentally unrolled these loops one to four times to find the optimum unrolling level for each. In addition to the inability of the compiler to dynamically test a variety of unroll levels, these opportunities may have been missed because the loops involve nested accesses to custom data structures. In total, our

loop optimizations resulted in a 15% savings for a single threaded swaptions execution, and a 19.7% savings for 24-threaded execution.

Given the simplicity of our optimizations, these changes resulted in disproportionately large performance savings. Across a datacenter or many nodes in a distributed system, the savings could be even more important, and potentially financially significant as well. Best of all, we were able to make the optimizations quickly for developers unfamiliar to these particular applications, because ParaShares allowed us to focus our efforts on just a few lines of code rather than thousands.

3.6 Related Work

To the best of our knowledge, Harmony is the first tool that dynamically records parallelism and maps it back to basic blocks in the application. However, a number of other tools dynamically measure program parallelism and profile thread activity, use the LLVM compiler, or collect basic block-granularity performance data.

Parallelism Analysis Tools. Kremlin by Garcia et al. [66] reboots the classic gprof [71] for the multicore era using hierarchical critical path analysis to help users identify application hotspots that would benefit from parallelization. Quartz [3] is an older tool with similar goals; it computes normalized processing times on SMPs for functions using statistical sampling. TAU [215] is a flexible but complex parallel performance evaluation environment for multi-node HPC systems. Intel's Parallel Amplifier [95] and VTune Amplifier XE [96] allow software engineers to examine performance and scalability of programs and to visualize program hotspots and thread activity. McLaren's QProf [161] unites fine-grained timing measurements with estimates to provide detailed timings of multi-threaded program events. Tallent and Mellor-Crummey use sampling to identify program overhead and identify serialization in Cilk programs [234]. The Sun Studio performance tools identify lock contention, load imbalance, and memory contention in multi-threaded programs [101]. PGPROF from the Portland Group allows users to profile OpenMP and MPI programs and to analyze application scalability [231]. Additional OpenMP parallel performance tools include a runtime API for parallel profiling described by Hernandez et al. [84], and ompP [65]: a tool

modeled off of mpiP [251] that identifies inefficient regions in OpenMP through source code instrumentation that counts OpenMP construct executions. The Pin binary instrumentation tool [202] can monitor a variety of performance metrics in parallel programs [10]. It is best suited to applications where perturbation will not affect measurements, because it can cause significant timing overheads. Several parallelism analysis tools have been built on top of the Pin framework. PinPlay [188], for example, uses Pin to dynamically replay multi-threaded programs with the goal of fixing concurrency bugs. The CilkView Scalability Analyzer by He et al. [81] examines the dependencies in a program to estimate its parallelism using Pin to collect performance metrics serially. Finally, Moseley et al. [168] build a Pin tool that looks for loop behaviors that might indicate easy opportunities for parallelization.

LLVM Performance Tools. We build Harmony on top of the LLVM Compiler Framework. LLVM comes with several instrumentation features including block, edge and path profiling [137]. Like us, other teams have built custom instrumentation passes. For example, VMAD by Jimborean et al. [109] extends LLVM with a pass to support an instrumentation framework that can gather memory–access traces. Rane and Browne analyze memory traces via LLVM instrumentation [198], and Serebryany et al. use LLVM instrumentation for dynamic race detection [212].

Basic Block Profiling. Others have also profiled at the fine granularity of basic blocks. For example, Sherwood et al. [217] use Basic Block Vectors to identify similar intervals of execution in a program, and Smith's Pixie [223] tracer identifies basic block boundaries in MIPS code to count block executions and to monitor the number of branch instructions taken.

3.7 Limitations and Future Work

The primary limitation of this work is that at present Harmony is usable only on pthreads applications that LLVM can compile. With some implementation effort, the tool could be extended to support other parallelization libraries (e.g., OpenMP), and the general architec-

ture could be readily ported to other compilers. In addition to extending the applicability of the Harmony tool, there are a number of other potential additional applications for utilizing PBV profiles spanning software engineering, operating systems, compilers, and machine learning.

Applications in Software Engineering. Writing parallel software is a challenging task. One particular challenge lies in verifying that applications consistently run as the developer expects. Harmony could assist this verification process by checking that particular parts of the program run at the degree of parallelism intended by the developer. For example, a language could introduce assertions to declare that a specific code region should never run when the thread count is greater than one. This might be a critical section, or it might be any other code region that a developer expects to execute serially. Then, Harmony could be modified to insert runtime checks and flag them for programmer inspection.

Another concurrency check that Harmony could assist with is the identification of code regions with *anomalous parallelism*. If a certain code region, say a function, is found to run serially 99% of the time and in parallel 1% of the time, this anomaly might signify a concurrency bug, or at least a potential mismatch in programmer intent and runtime behavior and could also be flagged for programmer review.

Applications in Operating Systems Research. As previously observed, many applications have a significant fraction of mixed parallelism blocks. These blocks might be indicative of poor operating system scheduling. Further examination of such mixed blocks could lead to improvements in scheduling policy.

Applications in Compilers Research. If compilers are knowledgeable about the degree of parallelism at which a basic block might run at, optimization selection could factor in this information. Multi-threaded programs might initially be optimized as if they were to be executed in serial, then run with Harmony profiling. The parallel block vectors produced could be used by a compiler to apply different optimization strategies to parallel and serial code. For example, if a heterogeneous CMP has in-order parallel cores, the compiler might expend more effort on instruction scheduling.

Profile driven re-compilation could also be employed when targeting code to specialized processors in a heterogeneous architecture. An initial run of an application with Harmony profiling followed by instruction analysis could determine the best processing unit on which to run a particular code region. Re-compilation could then prepare the application to run on specialized cores, potentially with different instruction set architectures (ISAs).

Mapping measured parallelism to basic blocks might also help a compiler improve program parallelism. Parallel classifications like always serial, always parallel, and mixed could be mapped to control flow graphs. The attribution of parallelism to CFGs might highlight certain graph patterns where opportunities for further parallelization exist.

Machine Learning. We chose the always serial, always parallel, mixed classifications because they are appropriate to the microarchitectural design case studies presented. However, blocks could be classified in a multitude of ways. For example, we identified blocks which always ran in parallel, but did not distinguish blocks which were highly parallel from blocks which were only somewhat parallel. That is, we did not separate blocks which ran concurrently with five other threads active from those that ran with one other thread. Different classifications might be useful depending on the profiling goal and the type of application being measured. Unsupervised learning could determine useful parallel classifications, leading to more interesting analyses and to further insights for a variety multi-threaded applications.

3.8 Discussion

Like puzzles turned sideways, sometimes new perspectives can yield new insights. Unlike existing profiles which examine parallel programs from the perspective of a thread or process, parallel block vectors collect runtime statistics by basic block laterally by parallelism phase. Parallel block vectors show which parts of a program belong to the serial and parallel phases of execution and in what proportion. Collection of parallel block vectors is fast. This chapter demonstrated Harmony, a compile-time instrumentation pass to collect runtime profiles with just 16-21% overhead. No manual code modification is required by the user, and profiles are architecture independent.

Fast collection coupled with detailed dynamic information about program behavior makes parallel block vectors broadly useful. This chapter examined four ways in which parallel block vectors contribute to this dissertation's goal of finding and minimizing system-wide inefficiencies. First, it identified basic blocks which do not fit the mold of Amdahl's pure parallelism and serialism and instead exhibit a mix of the two. Second, it demonstrated how parallel block vectors can uncover differences in program features at different degrees of parallelism. Third, it revealed that parallelism does not necessarily correlate with basic block execution frequencies. Finally, it showed how PBVs can be used to construct ParaShares, fine-grained scores that localize the bulk of parallel software runtime to a few important lines of code.

Chapter 4

Datacenter-Wide Application Interference

Distributed computing is another area in which inefficiencies can easily, and do frequently, occur. One specific type of inefficiency that arises in distributed datacenters that use computers with CMPs or SMTs is *application interference*. Application interference transpires when multiple applications contend for shared resources such as processor time, cache space, or I/O pins. In datacenters, where it is common to find many applications assigned to a server, this is a prevalent phenomenon. It is also a particularly undesirable one, as the increased running times and operating costs that result from application interference are multiplied across many machines.

Unfortunately, understanding interference in live datacenters is more difficult than in controlled environments or on simpler architectures. Most approaches to mitigating interference rely on data that cannot be collected efficiently in a production environment. This chapter[1] exposes eight specific complexities of live datacenters that constrain measurement of interference. It then introduces new, generic measurement techniques for analyzing interference in the face of these challenges and restrictions. We use the measurement techniques to conduct the first large-scale study of application interference in live production datacenter workloads. Data is measured across 1000 12-core Google servers observed to be running

[1]This work was previously introduced in a conference publication [118].

1102 unique applications. Finally, we identify several opportunities to improve performance that use only the available data; these opportunities are applicable to any datacenter.

4.1 Introduction

The complex characteristics of datacenter workloads and architectures make application interference difficult to reason about. High heterogeneity of applications and high core utilization targets mean that datacenters' CMPs are filled with a wide variety of multi-threaded applications. Because these applications are diverse in their performance objectives, resource requirements, and inputs, and because datacenters put severe limitations on performance monitoring, it is a challenge to even measure application interference, let alone manage it. Yet, as more applications migrate to datacenters, it has become critically important to keep negative application interference under control.

Many current approaches to monitor and combat interference work well on solitary machines, but fall short in a datacenter environment. Some techniques involve predicting application performance at a high level of detail, which is feasible in controlled settings with simple benchmarks and architectures, but becomes much more complex in datacenters. While it is possible to guess application performance at a high level and reduce interference to some degree, it is impossible to accurately predict performance to the level of precision required to eliminate it entirely. Other approaches use gladiator–style match-ups between applications to measure interference and find optimal scheduling solutions. This is not practical in a datacenter, mainly because of financial restrictions on how data can be measured. A third approach observes benchmark application performance (sometimes via simulation), then attempts to apply the observations to live applications. Some of these techniques rely on statistics that are not measurable in datacenters, while others are generous in their assumptions that noiseless and controlled offline measurements are later applicable in live, chaotic settings.

To measure live datacenter application interference, a new methodology is needed. Such a methodology should ideally be able to capture the interference effects of thousands of applications, running with real user inputs, on production servers with diverse architectural

platforms. Furthermore, the methodology should be financially reasonable, not requiring hundreds or thousands of machines for simulations and not disturbing the performance of production services.

In this chapter, we use our experience and exclusive access to live datacenter applications to expose the realities of measuring and analyzing interference in a datacenter. Then, we develop a methodology to measure live datacenter interference, and test the methodology on production servers at Google. Specifically:

- We identify eight sources of complexity in interference measurement and analysis that are either unique to datacenters or frequently not handled by previous works (Section 4.2).
- We introduce a generally applicable methodology for measuring application interference in the restrictive environment of a datacenter (Section 4.3).
- As a proof-of-concept, the methodology is implemented and used in the first large–scale study of measured application interference in a live datacenter. We collect data from 1102 unique applications across 1000 Google servers, each running on 12 core, 24 hyper–thread Intel Westmeres. These measurements capture the performance of production workloads, live schedules, and real user interaction (Section 4.4).
- Given the information that can be measured in live datacenters, we outline two opportunities to control negative application interference in datacenters (Section 4.5).

4.2 Complexities of Interference in a Datacenter

Application interference in a datacenter is much more challenging to reason about, measure, or predict than in a controlled environment or on a solitary machine. It is important for scheduling experts and datacenter systems specialists to understand what performance analysts are up against. This section describes eight specific complexities that are unique to datacenters or largely unaccounted for in past work, in some cases preventing the use of established methodologies for combating application interference. For example, many past works run an application on an isolated machine to determine its baseline performance,

and then run the application with a single application co-runner to measure interference effects([27, 38, 83, 107, 128, 156, 157, 172, 235, 236, 264, 265, 267]). The pairwise impacts are then incorporated into scheduling policies or used to fairly allocate resources between applications. Such techniques rely on well-defined, discrete applications and isolated measurements, neither of which is available in a datacenter. There are thousands of applications to test, user inputs vary in non-obvious ways (such that they cannot be simulated off-line), and applications are frequently re-written and updated.

Other approaches estimate the resource usage of applications and attempt to schedule applications with complementary needs together ([5, 19, 24, 34, 36, 58, 106, 108, 129, 166, 170, 195, 261]). While some general predictions can be made about application performance, it is challenging to make such predictions precise in the complex environment of a datacenter.

The eight complexities below are common to most datacenters; to show that they are realistic, we use experiences and data from our measurement study of production servers at Google described in Section 4.4.

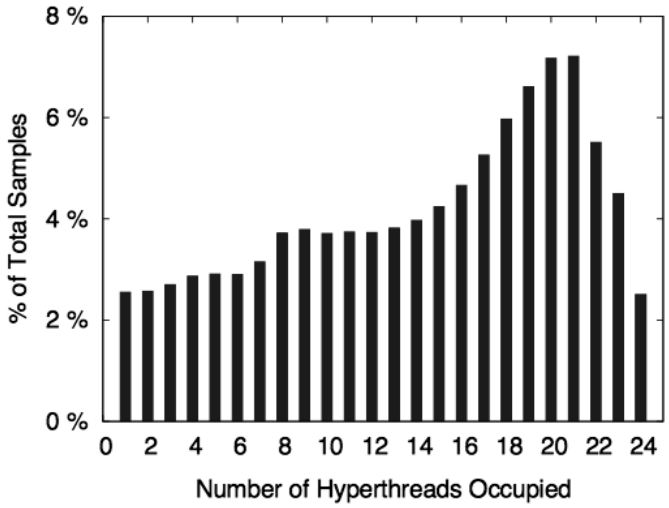


Figure 4.1: **Datacenter machines are filled with applications.** Profiling 1000 12-core, 24 hyperthread Google servers running production workloads, we found the average machine had more than 14 of the 24 hyperthreads in use. These results reveal the extent of multi-way interference, which is largely un-handled by existing interference management techniques.

4.2.1 Large Chips with High Core Utilizations

When slow page loads translate into lost revenue, the pressure to deliver web content quickly is high. Datacenters are driving the demand for increasingly high-core-count chips. CMPs with as many as 100 cores already exist [238], with datacenters today using CMPs with tens of cores. The 1000 Google machines profiled in Section 4.4 are 12-core machines supporting up to 24 hyperthreads. These core-crowded chips mean more applications are sharing resources, such as cache, that they otherwise would not share. Despite this, a survey of recent work in application interference shows that many researchers validate their solutions on chips with only two or four cores ([5, 9, 34, 38, 50, 83, 102, 106, 128, 156, 235, 236, 261, 265, 267]).

In the early days of CMPs, resource contention was not the issue it is today: core counts per chip were low, and datacenters once struggled to use all cores on a chip (see the "bin-packing" problem discussed in [86]). Because it leads to power savings and better parallel performance, high core utilization is desirable, and it has been increasing along with per-chip core counts [125]. Today, core utilization is already high: in profiling the 24-

hyperthread machines, we found that an average of about 14 hyperthreads were occupied. Figure 4.1 shows the full distribution of observed hyperthread occupancies.

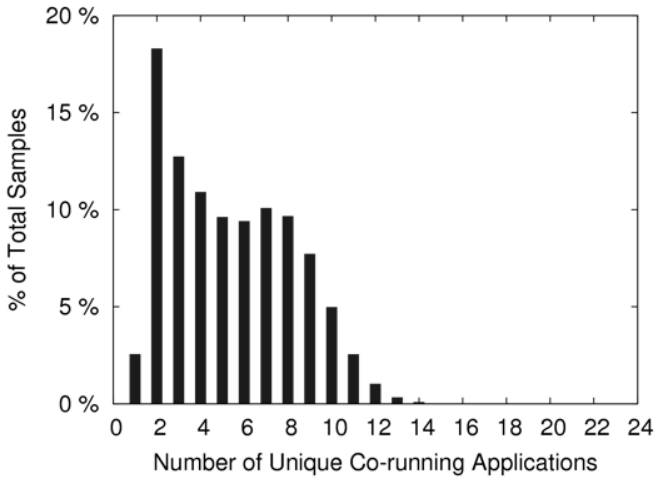


Figure 4.2: **Datacenter servers have diverse application mixes.** Google server profiling reveals that most machines run five or more unique applications at once, and sometimes as many as 20. Many past works consider only two applications running together at a time, a scenario present only 20% of the time in to this data.

4.2.2 Heterogeneous Application Mixes

Datacenter servers not only support many application threads at once, but frequently also execute a diverse mix of applications on each machine. This is not surprising considering the massive number of different applications that run in datacenters today. For example, our profiling of the Google servers revealed 1102 unique applications. While a couple of these were system support applications and thus constantly or periodically running on all machines, the vast majority could be flexibly scheduled among servers in the fleet. Our measurements also showed that a machine runs at least five applications half of the time, and sometimes runs as many as 20 (see Figure 4.2). Characterizing interference is much simpler if only a couple of unique applications are scheduled together, so much prior work assumes only two applications running on a machine at a time. According to Figure 4.2, such methodologies would apply only about 20% of the time.

4.2.3 Fuzzy Application Delineations

Sometimes, even trivial issues become complex in datacenter settings. To measure application interference, one needs to define an application. Applications might be defined as narrowly as on a per process basis, or they can be delineated by user, input, or code segment. The division of applications is tricky though; define them too narrowly, and there will be insufficient data to get useful interference information. Define them too coarsely, and performance variations unrelated to application interference may inadvertently be captured. There is no clear right choice for how applications should be delineated. In the Section 4.4 study and in Figure 4.2, each unique binary is considered to be an application, which is a fairly coarse-grained classification.

4.2.4 Varying and Sometimes Unpredictable Inputs

Unlike in controlled environments, applications in a datacenter are added or updated frequently. Many applications accept user inputs and can experience significant performance swings based on usage, sometimes with predictable periodicity, and sometimes without. It is intuitive that input could affect how an application interferes or is interfered with (Jiang and Shen [106] show this formally), but most prior studies use just single–input benchmarks.

4.2.5 Varying Micro-architectural Platforms

Performance changes depend on the micro-architectural platform as well as inputs. In a large datacenter, it is uncommon for all servers to use the same micro-architecture. As new chips become available, datacenters will incrementally update their servers, resulting in an evolving, heterogeneous mix of platforms. Most past work does not consider this, but interference measurement and mitigation techniques should ideally be micro-architecture independent.

4.2.6 Unknown Optimal Performance

Many existing interference solutions rely on knowing an application's optimal performance without interference. For static input benchmarks, this is as simple as running the appli-

cation on a dedicated machine. At a datacenter, isolating a production application on a dedicated machine is a prohibitively expensive way to find baseline performance, especially given the number of applications to evaluate and the need for frequent re-evaluation as inputs, architectures, or even the applications themselves change. When we conducted our measurement study, Google would not allow us to measure the baseline performance of applications on isolated machines due to the cost.

4.2.7 Limited Measurement Capabilities

Performance analysts at datacenters are restricted in other ways as well. For example, an extremely limiting restriction that we had to work around in developing our methodology for the Google study was that we had to keep our profiling overhead as low as possible, and preferably well under one percent. Google's rationale, which is likely to be echoed by other datacenter companies, is that excessive overhead in measuring is not always a worthwhile investment. The financial losses caused by too much measurement perturbation in the present may outweigh future performance gains that are discoverable with the additional measurements.

4.2.8 Corporate Policy and Structure

Other difficulties relate to corporate policy and the often large size of datacenter companies. For example, performance analysts and scheduling policy makers might work in completely separate teams. That means performance analysis results must be sufficiently flexible to be fed into completely independent scheduling tools. A large company might also delay the deployment of new performance monitoring tools for strategic or accounting reasons. As a result, new solutions might not be testable or applicable for months. Performance objectives of an individual application may also compete with system-wide goals. Even if it were easy to identify and quantify every instance of negative interference, it is not always clear how each instance should be resolved. For example, in most cases a latency-sensitive application's performance is prioritized over less important applications, but performance must also be balanced with cost-efficiency. Thus, even latency-sensitive applications are likely to be co-scheduled with other applications to keep utilization up.

4.3 A Methodology for Measuring Interference in Live Datacenters

Put together, all of the complications outlined in the previous section make for intricate interference scenarios with restricted means to collect data about interference. Here we outline a series of techniques that form the first complete methodology for measuring application interference in the restrictive environment of a live production datacenter. Figure 4.3 shows an overview of this methodology. First, performance data is measured in small samples on live production servers using a small number of remote collection machines. Next, the data is examined to find per-application baseline performance comparators and to identify interference relationships between applications. These relationships are then made to be architecture independent so that performance data can be aggregated across all of the machines monitored. Afterwards, the aggregated performance data and the baseline performance indicators can be used together to analyze system-wide application interference.

4.3.1 Collecting Low-Overhead Performance Metrics

The most accurate way of capturing interference relationships in a datacenter is to measure them live. Since it is critical not to degrade performance, all measurements taken must have as little overhead as possible. Past work shows that sampling-based performance monitoring minimally perturbs applications. For example, the Google-Wide Profiling (GWP) tool [203], from which we borrow some measurement ideas, profiles live applications with less than 0.01% overhead using sampling-based monitoring. GWP samples performance data using `perf` [1], a Linux performance monitoring tool. Perf not only has low overhead, but it also provides abstractions over hardware capabilities, meaning the same monitoring commands can be issued on many different hardware platforms in a datacenter. The tool samples a number of measurable *events* including software events that interface with the kernel (such as page faults) and hardware events reported from the processor (such as CPU-cycles and various types of cache misses).

To further limit overheads, performance information can be reported to a small number of remote, non-production machines for later analysis. Also, sampling periods and frequen-

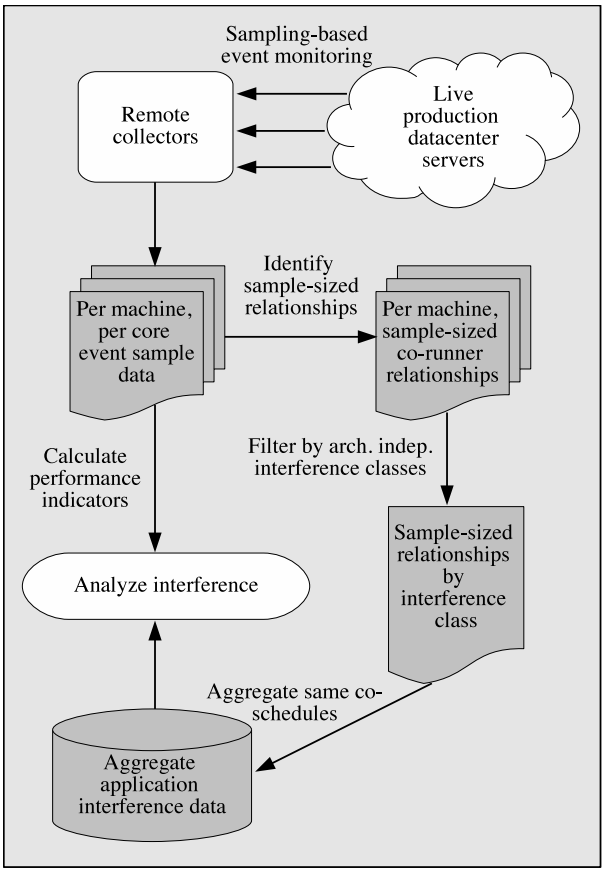


Figure 4.3: **A methodology for measuring application interference on live production servers is described in Section 4.3.**

cies — the number of occurrences of an event per sample, and the average rate of samples per second, respectively — and collection duration per machine can be tuned so that they are high enough to record useful information, but not so high that performance monitoring is overly intrusive.

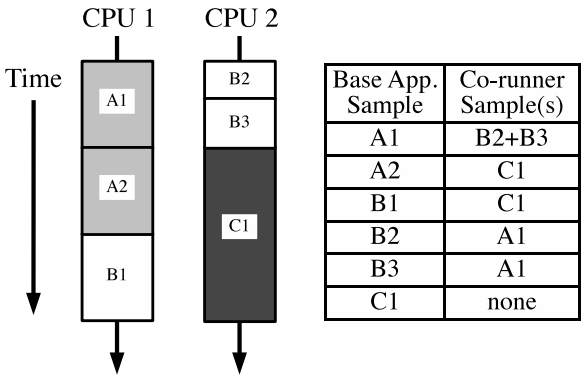
4.3.2 Statistical Performance Indicators

One challenge of assessing interference relationships in datacenters is that the optimal performance of applications is usually unknown. Since user inputs have a big effect on measured performance, and because the cost of isolating an application on a machine is high, it is rarely possible to find out how an application would perform with no application interference. Performance measurements of an application in the wild are usually clouded by several co-running applications. So, instead of using optimal performance as a baseline, we use a statistical performance indicator.

The performance collection technique described above results in sampled performance metrics. After collection, a statistical estimator that aggregates these fine grained measurements—e.g., the mean cycles per instruction (CPI) of a large number of samples—can be used as a comparator for future observed samples. Although some dimensionality is lost in aggregation, a statistical performance indicator works well for a couple of reasons. First, only one hardware counter needs to be monitored, so the necessary information can be safely collected without perturbing live applications. Second, the indicator can be compactly stored and updated for large numbers of samples and applications.

4.3.3 Identifying Sample-Sized Interference Relationships

In a controlled experiment, two applications can be run simultaneously on a machine, with applications' performance interactions monitored for the duration of their execution. As Section 4.2 explained, such co-scheduling cannot be forced in a datacenter. Another complicating factor is that applications run for extremely varying amounts of time. One application may run for a week, for example, during which time many different sets of other applications may alternately share the same machine. Thus, it is difficult to attribute the original application's performance to any one (or even any one set of) co-running applications. To



Base App. Sample	Co-runner Sample(s)
A1	B2+B3
A2	C1
B1	C1
B2	A1
B3	A1
C1	none

Figure 4.4: **Sample sized co-runners.** Timelines of two CPUs on the same machine are shown to the left. Each segment represents a performance sample (e.g., 2 million instructions) from an application. For example, *A1* is the first sample of application *A*. The table to the right shows the *co-runner* samples for each *base application* sample. Application *A1* has two co-runners because two consecutive samples of application *B* run for its duration. In this contrived example, sample *C1* is especially long to illustrate the uncommon case of a sample having no co-runners.

learn specific interference relationships, live data must be carefully filtered.

Each performance sample includes a time-stamp, which can be used to identify which samples overlap in runtime, and eventually reveal interference relationships. Specifically, for a given *base sample*, we compile a list of the given sample's *co-runners*. A co-runner is a sample that ran for the entire duration of the base sample. We use an algorithm similar to liveness analysis in compilers to identify co-runners. The input is the starting time of each base sample, from which we work backwards to find other samples that were "live" for the duration of the base sample.

Figure 4.4 shows an example of samples from two CPUs and the corresponding co-runner relationships between those samples. Each segment in the figure is a different sample, and letter labels are application names so that *A1* is the first sample of an application *A*. Since by definition, co-running samples must run for the same amount of time or longer than the base sample, it may not be possible to identify co-runners for long samples. This can be mitigated by combining successive samples when we are looking for co-runners of a base sample. In Figure 4.4, sample *A1* has two co-runners because two successive samples of application *B* run for its duration. Some samples still may not have co-runners (as illustrated by the long sample *C1*). When applying this methodology (Section 4.4), we

found that this is the case for just 0.6% of the samples. This number can be kept low if the number of samples per context-switch is relatively high; if many samples in a row are of the same application, it is more likely that co-runner relationships can be identified.

Extrapolating application-level interference relationships from a collection of sample-sized relationships is straightforward. First, all of the base samples for the base application are identified. Those samples are then sorted by their identified co-runners. Any base samples with the same sets of co-runners can be aggregated to determine the interference relationship between the base application and a set of co-running applications. With enough samples, this technique becomes schedule-independent. Depending on the schedule, more samples may be collected that represent a certain interference relationship, but with prolonged sampling, all interference relationships that occur can eventually be identified. Thus, *interference relationships can be determined without any prior knowledge of the scheduling policy*. This is extremely useful in a datacenter, because scheduling policies may be very complex, and may even be unknown to those trying to understand interference.

4.3.4 Interference Classes

Interference depends on the resources that two applications are contending for. Depending on the topology of the architectural platform, all applications sharing a chip may not have equal influence on one another. Consider, for example, two applications which share all of their cache versus two applications that share only interfaces to peripheral devices (like an I/O hub). Our analysis distinguishes between such types of interference using architecture independent *interference classes*. An interference class defines the closest relationship (in terms of resource sharing) that two applications running on the same chip might have. The closest interference relationship is between two applications running on different hyperthreads of a single core. Such applications contend for everything from execution slots to cache to memory control and I/O resources. A more distant relationship would be between applications which share the same last level cache and resources beyond. The loosest interference class is between two applications which are on the same chip, but which do not share any resources except their interface to peripheral devices.

Others have used interference classes to estimate the potential amount of interference in

various assignments of applications to a machine (see contention groups in [90] for example). We see a few additional reasons that defining interference classes can be beneficial. First, it allows for data to be aggregated simply across samples on many-core machines — all shared core co-runners, for example, can be considered equivalent. Next, it allows for the aggregation of data across machines with different (but similarly symmetric) architectural platforms. Finally, interference classes help reduce the complexity when considering the range of possible co-schedules of multiple applications at a time.

4.4 Applying the Measurement Methodology

We now apply the general application interference measurement techniques established in the previous section to conduct the first large-scale study of interference on production Google servers running workloads with live user interaction. Unlike past work, this study does not rely on benchmarks or simulation. The study illustrates the noisiness of production interference that any datacenter interference analyst must negotiate. It also reveals that some interference patterns are visible above the noise, leading to exploitable performance opportunities, which are discussed in Section 4.5.

4.4.1 Collecting Performance Metrics

We used the perf tool and remote collection methodology described in Section 4.3 to collect samples across 1000, 12-core production servers at Google. As described, the basic methodology allows for a choice between a number of different performance events to monitor. Unfortunately, there is no single perfect hardware-counter that accurately indicates performance across a variety of applications. With such a large number of applications to compare, it is nearly impossible to use application-specific metrics (like time per transaction) for this study. Application run time is out because it is not necessarily related to performance in datacenters (think an ads server that runs continually until stopped for an update). Some have suggested that last level cache (LLC) miss rates are the best indicators for interference studies [27], while others note that LLC will not accurately monitor all workloads, especially those that are memory bound [235]. Other work suggests that

contention for memory bandwidth and buses might be a good indicator [129, 170, 173]. To capture the effects of cache and memory contention, we use instructions per cycle (IPC) to indicate performance in this study. Although it has been widely used in past interference studies (e.g., [24, 61, 156, 166, 167, 195]), there is debate about IPC too. In particular, Alameldeen and Wood found that architectural enhancements can cause IPC to improve even as application performance worsens, or vice versa — especially for multi-threaded applications [2]. To avoid such unexpected discrepancies, we ensured that the profiled servers were identical in all respects, including chip type, clock speed, RAM, and operating system. If future studies are conducted across multiple architectural platforms, it may be necessary to consider metrics other than IPC.

Application IPC was sampled every 2.5 million instructions. After 2.5 million instructions executed on a production server's core, the remote profiler recorded the time-stamp, the location of the core on its machine, and the application executing. In post-processing, we connect the elapsed time per sample with the machines' clock speed to get the IPC of each sample. Over the course of the study, the remote profiler encountered 1102 unique binaries and collected nearly 350 million samples. See Table 4.1 for a summary of the collection statistics.

Table 4.1: **Profiling and Collection Statistics**

Performance Sample Size	2.5×10^6 instructions
Monitored Indicator	Instructions per cycle (IPC)
Number of Machines*	1000
**Machines identical in all respects (e.g., clock speed, RAM, O/S)*	
Threads / Core	2
Cores / Socket	6
Sockets / Machine	2
Threads / Machine	24
Unique Binaries Encountered	1102
Samples Collected (all 1102 applications)	3.45×10^8

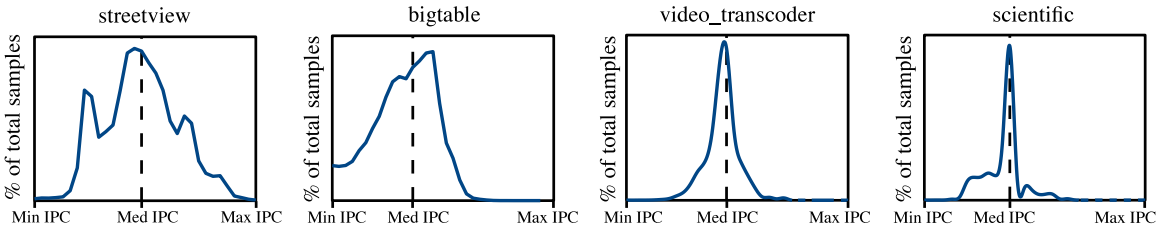


Figure 4.5: **Median IPC is a good performance indicator for the Google data collected.** Each graph shows the performance variations of the specified application when scheduled with eight of their most common co-runners. The overall median IPCs for each base application correspond well to their performance curves.

4.4.2 Statistical Performance Indicators

From the raw samples we calculated a statistical performance indicator to estimate a baseline performance for each application. Because the collected IPCs did not form a normal distribution, we use medians rather than mean as an indicator. For each application and for each sample, we calculated and recorded the median IPC. Note that this aggregated metric is scheduling *dependent*, and we did not examine the schedule in our calculations. There are two reasons for this. First, provided our samples are representative of the system as a whole, a scheduling dependent performance indicator tells us what the normal performance of an application is in the datacenter overall. We believe the samples were representative, as our collections spanned 1000 international machines and a period of twelve hours. Second, it did not make sense for us to try to account for the scheduling system, because the policies in place at Google are not only highly complex, but also highly secretive. If scheduling policies change in the future, the methodology does not need to be revised. To evaluate the choice of medians, medians were compared to the performance curves of the data collected. Figure 4.5 shows the distributions of performance samples for four common Google applications (`streetview`, `bigtable`, `video_transcoder`, and `scientific`). The y-axes on the graph show the percentage of samples that range from the minimum to maximum IPC of each application on the x-axes. The graphs reveal that medians are a representative aggregate indicator. All absolute and relative IPC values have been anonymized at Google's request.

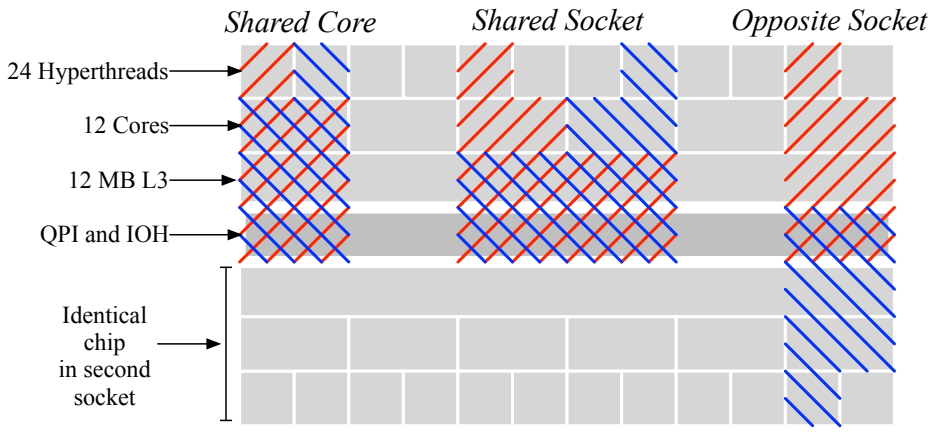


Figure 4.6: **Westmere Interference Classes.** The profiled Intel Westmeres are dual-socket machines, supporting 12 hyperthreads per socket. Interference relationships are partitioned into three classes as depicted here: *shared core*, *shared socket*, and *opposite socket*.

4.4.3 Identifying Sample-Sized Interference Relationships

Returning to the raw, unaggregrated performance samples, the next step was to find co-runners among application samples. As explained in Section 4.3.3, by definition co-running samples must be longer running than or equal length to the base application sample. Because of this, we were concerned that the samples dropped due to lack of co-runner might be biased towards the slower samples. However, the effects were not significant in the data collected. Across the most frequently occurring eight applications only 0.6% of the samples were dropped, with the peak being 3.47% for `search`. The impact on median IPC was negligible; dropping samples reduced it by just 0.23% on average.

4.4.4 Defining Interference Classes

The machines used for collection in this study all have the same chip, so only one set of interference classes needs to be identified. The chips are Intel Westmeres, which have two hyperthreads per core and six cores sharing an L3 cache for a total of 12 hyperthreads per socket as pictured in Figure 4.6. With two sockets connected by an Intel Quick Path Interconnect (QPI) and to an I/O hub (IOH), each Westmere supports a total of 24 hyperthreads. Given this topology, there are three discernible interference classes, also depicted in Figure 4.6. The closest is between two applications on hyperthreads which share a core

(*shared core*); then between two application threads on different cores but sharing a socket and thus an L3 cache (*shared socket*); and finally between two threads on the same machine but on different sockets (*opposite sockets*) which share only the QPI and IOH.

For each of the sample co-runners previously identified, we looked at the relative core locations of the applications. Using these core locations, we assigned each pair of co-runners the appropriate interference class label. Between eight of the most commonly running applications we encountered, the average number of shared core samples ranged from 2000 to 45 million, with about 1 million samples on average. Between the same applications, the number of shared socket samples ranged from 12000 to 400 million per application and 9.5 million on average. The opposite socket relationships ranged from to 14000 to 500 million samples with 11 million on average.

4.4.5 Analyzing Interference

A primary question in past work is *how does a base application's performance change with a particular co-runner?* This is a very challenging question to answer in a datacenter. One approach is to examine the performance effects of one application on another by aggregating all of the performance metrics from the sample-sized relationships of a particular base application and a particular co-running application. However, up to 22 other hyperthreads may be occupied with various unrelated applications during each of the samples, so this must be taken into account. It was rare to find only two applications running together on a machine, which is not surprising considering our earlier observation that Google maintains a high thread occupation rate (Figure 4.1) and runs diverse applications together on a single machine (Figure 4.2). The shared core interference relationship is especially important to understand as it is likely the strongest. Finding two applications running in isolation on the same core with the remaining threads empty was an extremely rare occurrence; probably due to intentional scheduling decisions to distribute resources.

Regardless of the reasons, it is clear that noiseless data is hard to come by in a datacenter. Thus, pairwise comparisons can never fully capture all the causes of interference. Still, we wanted to attempt to see if shared core influences were strong enough to be apparent over the noise of applications scheduled on the rest of the machine. Though necessarily incomplete,

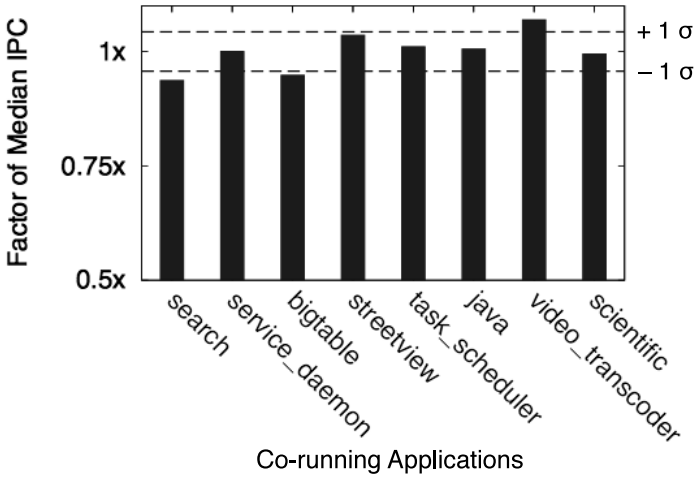


Figure 4.7: **Streetview's performance variations across co–runners.** Bars represent `streetview`'s normalized median performance when co–located with eight common co-runner applications. Dashed horizontal lines show overall variance of all measured `streetview` samples.

if pairwise comparisons can yield any information, they are attractive for two reasons. First, reducing the comparison space makes the resulting information easier to collect, understand, and analyze. Also, some schedulers — including Google's — are already prepared to accept pairwise scheduling information but not more complex inputs.

To find shared-core influences, we aggregated the previously identified pairwise relationships of eight commonly running applications, filtering the samples to use only those that were labelled as shared core. To reduce random performance variations, we required that a minimum of 1000 samples be present for each aggregated metric to be significant; all 64 cross-pairings satisfied this minimum.

Figure 4.7 shows `streetview` as it shares a core with eight other applications. Other applications exhibit similar performance effects in their shared core co-runner graphs. In Figure 4.7, bars along the x-axis show the shared core co-runner of `streetview`, and the y-axis gives the normalized median IPC across each of the aggregated `streetview` and shared core co-runner samples. The dotted horizontal lines show the average variance across all of the measured (co-runner independent) `streetview` samples. We note that while *it is difficult to tell an exact ordering of* `streetview`*'s best to worst co-runners* given the large variance of the samples, it is clear that a few shared core co-runners interfere beyond the

noise.

We collected data on shared socket and opposite socket pairwise interference using the same technique as before. The additional data is not included here because it does not add much insight. In part, this is because the pairwise influence of sharing a socket or machine can be weaker than when sharing a core. Consider, for example, a co-runner sharing a socket with a base application. The base application has one shared core co-runner and ten shared socket co-runners on a Westmere (recall Figure 4.6). So, if we try to examine the effects of a single shared socket co-runner on the base application, we are also capturing the effects of at least ten other co-runners sharing as many or more resources with the base application. To fully understand shared socket and shared machine influences, it would be useful to examine more than just pairwise interference, and to consider larger groups of co-running applications.

4.5 Performance Opportunities

Given a total ordering of interference relationships, some past works are able to find optimal schedules and sometimes nearly eliminate negative interference. An important goal of this work was to show that such solutions cannot be immediately successful when applied to datacenters, primarily because the precision required to determine a total ordering of relationships is not available. The measurement techniques in Section 4.3 outline a path towards better understanding application interference in datacenters, where the measurable information is necessarily more limited. Although it is disappointing that many insightful techniques cannot be immediately applied in datacenters, the good news is that in a datacenter even small reductions in application interference can be valuable. In this section, we outline two techniques that are immediately applicable in a datacenter once the data outlined earlier in this chapter has been collected.

4.5.1 Restricting Beyond Noisy Interferers

With many applications running on live machines, it is difficult to observe isolated (noise-free) interactions. Moreover, measurement restrictions make the discovery of a full ordering

of co-runner preferences difficult. Despite the noise, the data still allow us to recognize that some applications interfere. We define *beyond noisy interferers* (BNIs) as applications that can be clearly seen to hamper another application's performance despite the noisy data. To identify BNIs, we find the average variance from the mean performance of a base application that incorporates all possible co-schedules. This metric indicates the average expected performance fluctuation of an application across diverse scheduling scenarios. Next, the measured samples of a particular co-scheduling relationship can be compared to the overall variance. If a co-schedule affects an application beyond its normal variance, it is classified as a BNI.

We applied this procedure to the Google data to see if any shared-core co-runners could be classified as BNIs. Figure 4.8 shows the performance of eight common Google applications when they were observed to be sharing a core with one of the other eight applications. Boxes in the matrix show the difference from the average variance (across all 1102 applications encountered in the study) of each base application (on the y-axis) for each co-runner (on the x-axis). A white box indicates that the shared-core co-runner positively interferes with the base application beyond the average variance, while a black box indicates negative interference beyond the average variance. Several negative BNIs (6 of 64 possible, or nearly 10%) emerge despite the fact that most of the observed data includes noise from other applications interfering outside of the shared core.

Such observed BNIs do not yield a complete ordering of application co-schedule preferences, and thus do not allow allow the compilation of an optimal schedule. Negative BNIs can, however, indicate specific applications that should not run together. A simple scheduling policy change to restrict negative BNIs from running alongside the base application could result in significant performance gains. Similarly, positive BNIs might be purposely scheduled with a base application to improve its average performance.

In some cases, even eliminating one or two bad co-runners could result in significant performance improvements for an application. In this data for example, the `bigtable` application is a negative BNI for `streetview`. If we eliminate all instances of `bigtable` running with `streetview` and assume that `streetview` will then perform at its median, then `streetview`'s overall performance will have improved by about 1.3%. If we also ex-

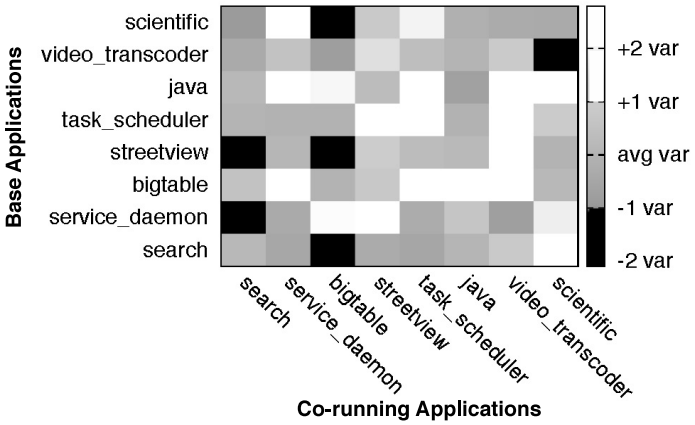


Figure 4.8: **Beyond noisy interferers in the Google data.** Shared core co-runner applications along the x-axis affect the performance of base applications along the y-axis. White boxes show co-runners that positively interfere beyond the average variance with base applications, while black boxes show co-runners that negatively interfere beyond the average variance.

clude `search` from running with `streetview` and make the same assumption, `streetview`'s performance could jump as much as 2.2%. Though these effects may seem small, when multiplied across weeks or months of application execution on thousands of servers, such improvements could result in sizable monetary savings.

4.5.2 Isolating Sensitive Applications and Exiling Antagonists

It is interesting to know how *sensitive* an application is to performance changes. Several previous studies have looked at application sensitivities in the context of resource contention ([108, 129, 154, 155, 235]), some of them using datacenter workload benchmarks. In these studies, sensitivity is defined in terms of an application's optimal performance. As explained in Section 4.2, it is difficult to ascertain a datacenter application's optimal performance, but we can extend the earlier work to comply with the available data. Specifically, the variance data used to determine BNI application relationships in Figure 4.8 can also be used to determine an application's overall sensitivity. Base applications with large performance variations across co-runners can be identified as sensitive to performance changes. For example, in Figure 4.8 the `scientific` and `streetview` applications have shared core co-runners that cause their performance to swing both above and below one average variance.

If the performance of these two applications (or any sensitive application) is important to the datacenter, systems managers can decide to isolate the applications on their own core, or even their own machine.

Antagonistic applications can be identified in a similar manner. A co-running application is antagonistic if it frequently causes base applications to exhibit negative performance swings beyond their average variances. In the figure, `bigtable` is a negative BNI for three applications, so it can be classified as antagonistic. Again, depending on the performance goals of the datacenter, it might make sense to exile such antagonistic applications to their own core or machine so that they do not negatively interfere with other applications' performance.

4.6 Related Work

Several papers and textbook chapters highlight challenges associated with CMPs in datacenters. Ranganathan and Jouppi discuss challenges related to general trends in changing infrastructures at large datacenters [201]. Kas writes about problems that must be solved as datacenters adopt CMPs, but does not specifically address the difficulties involved in measuring application interference [125]. One relevant description of the challenges of resource interference between applications can be found in Illikkal et al.'s work which discusses potential performance problems due to shared resource interference but does not detail the challenges of measuring interference [93].

While this work is the first to conduct a datacenter scale application interference study on live production workloads, a number of other researchers have conducted application interference studies geared towards datacenters. Rather than measuring live applications with user interaction, the following studies use benchmarks, simulations, and offline analysis of server workloads. While a benchmark runs, Mars et al. use performance counters to detect cache miss changes and identify contention so that schedules can be adaptively updated [157]. Another paper by Mars et al. measures changes in instruction rate to detect cross-core interference and adapt schedules accordingly [156]. Tang et al. try different thread-to-core mappings of benchmarks to methodically find the best co-schedules [236].

Another large scale study models resource interference of server consolidation workloads, finding core and cache contention [5]. This methodology requires estimates of cache usage and considers only two jobs co-scheduled at a time. Bilgir et al. simulate Facebook workloads to look for energy and performance benefits in assigning the correct number of cores and mapping applications effectively across CMPs [23]. The works by Carter et al. [33] and Levesque et al. [143] evaluate whether increasing core counts on Cray macines will improve scientific applications' performance by esimating their memory bandwidth contention. Finally, Hood et al. [90] and Jin et al. [110] break down expected contention by class for different arhcitectural platforms using microbenchmarks. They then estimate how real applications will perform on different architectural configurations.

A number of other works have measured the use of shared resources on single machines. Moseley measured resource sharing between threads in simultaneous multithreading (SMT) processors using hardware performance monitoring [167]. Snavely and Tullsen conduct an impressively thorough study of application co-scheduling on SMT architectures [224]. Like us, they use sample-based performance monitoring, but their work uses simulation and benchmarks rather than live workloads and relies on testing a significant number of permutations of all jobs co-scheduled together. Azimi et al. also use hardware sampling of benchmarks to study how threads share resources so that they can optimize cache locality and determine how caches should be partitioned on SMT machines [9]. Zhang et al. perform an extensive examination of cache contention between applications on varying CMP platforms [264], while Zhao et al. took a more detailed approach, monitoring not just cache sharing but occupancy and interference as well [267].

There is no dearth of related previous research proposing operating systems or hardware solutions to mitigate application interference. Unfortunately, many of the proposed ideas cannot accommodate the complexities outlined in Section 4.2. It is difficult to give credit to everyone who has contributed to such a well studied area. We have already discussed a number of works in this area that use measured performance monitoring as input; another relevant body of work estimates applications' resource usage to improve scheduling ([19, 34, 36, 58, 107, 108, 129, 133, 170, 195]). There is also a series of work that adjusts access to computing resources like CPU processing speed and cache partitioning size to make resource

sharing more fair ([54, 61, 83, 93, 102, 128, 159, 166, 172, 261, 265]).

4.7 Limitations and Future Work

Using the data collected in the Google study, it is possible to identify BNIs and to find sensitive and antagonistic applications that can be isolated or exiled, respectively. With extensions to the methodology outlined here, there are further opportunities to minimize interference and improve performance, that can help reduce some of the possible limitations of this work.

Performance Indicators. It is possible that the performance indicators used in finding beyond noisy interferers may not be correctly pinpointing poor application co-location, and instead may simply be application phase changes. A workaround is to collect data on multiple events across separate trials to compare for a fuller picture of application performance and interference. Correlating IPC with metrics such as LLC misses and I/O contention, could lead to more insight than examining than any one metric on its own. The challenge of correlating multiple performance events is that application co-schedules have to be matched across trials. When we analyzed the Google data, we were able to greatly reduce the aggregation complexity by combining sample data across same shared-core co-runners without filtering based on the rest of the applications co-scheduled on the machine. This method is a starting point for correlating multiple events, but it would be more precise to match the full machine co-schedules instead of just matching shared-core co-runners. Additionally, it is possible that our use of medians was not a perfect summary of the application data collected. In the future, it might be helpful to experiment with other statistical summaries, such as means.

Multi-dimensional Scheduling Constraints. This initial study focuses on pairwise interference effects, for simplicity and because Google's scheduler was already ready to accept pairwise scheduling inputs. There may also be significant trios or even larger sets of application co-schedules with relevant interference patterns. For example, some application A might not perform poorly with either B *or* C as a co-runner, but may perform poorly

when B *and* C are both co-runners. One could identify triplet (or larger) BNIs using the same techniques as for pairwise BNIs. Once identified, larger groups of BNIs could be employed in all the same ways as pairwise BNIs. As discussed in Section 4.4.5, this would be particularly useful when examining the effects of interference beyond shared core.

More Fine-grained Application Definitions. It is well known that some applications exhibit distinct phases with different performance characteristics. Such phases might obfuscate the process of identifying performance effects. In our Google study, we were able to observe fairly stable performance (Figure 4.5) by limiting our measurement study to twelve hours because most of the applications had diurnal phases based on the peak and off-peak usage of users. For important applications, it may be worth the additional complexity to identify distinct phases more precisely. Then, each phase of the applications could be considered as separate "applications" when analyzing co-runner relationships. Similarly, if a given application's performance is known to vary widely based on input, the application could be broken apart according to its usage pattern.

4.8 Discussion

Researchers need to develop scalable application interference solutions, and this work made a few contributions towards that goal. First, it identified the challenges of measuring and analyzing application interference at datacenter scale, exposing eight specific challenges that are unique to datacenters or that remain largely un-addressed in past research. These factors combine to make interference effects in a datacenter exceedingly difficult to predict, measure, and correct. To assist in the efforts of understanding interference between datacenter applications, we suggested a collection of measurement techniques to work around the complexities. The new techniques are generically applicable for any datacenter, but as a proof-of-concept, we implemented them to conduct an application interference study on production Google servers. The study, which is the first large-scale measurement study of application interference, revealed application interference "in the wild" on 1000 12-core machines running live commercial datacenter workloads. Using just data that is feasible to collect in the restrictive environment of a datacenter, we outlined several opportunities to

improve performance and improve overall system efficiency by reducing negative application interference.

Chapter 5

Fast Computational GPGPU Design

In this chapter, we take a proactive approach to improving efficiency by addressing it at the time of hardware design.[1] We target a type of hardware for which it was previously very difficult to measure the behavior of software on potential future designs. Specifically, this work facilitates the process of designing graphics processing units that can efficiently run computational applications, i.e., *general purpose graphics processing units (GPGPUs)*. Today, graphics processing units are increasingly used to execute computational workloads, a task for which they were not originally designed. To design new GPUs that can efficiently meet the unique needs of these workloads, architects need the help of performance simulation. Unfortunately, computational GPU programs are so large that simulating them in detail in their entirety is prohibitively slow.

This chapter addresses the need to understand very large computational GPU programs in three ways. First, it introduces a fast tracing tool that uses binary instrumentation for in-depth analyses of native executions on existing architectures. Second, it characterizes 25 commercial and benchmark OpenCL applications, which average 308 billion GPU instructions apiece and are by far the largest benchmarks that have been natively profiled at this level of detail. Third, it accelerates simulation of future-hardware by pinpointing small

[1] This work was previously introduced in a conference publication [113].

subsets of OpenCL applications that can be simulated as representative surrogates in lieu of full-length programs. The fast selection method presented here requires no simulation itself and allows the user to navigate the accuracy/simulation speed tradeoff space, from extremely accurate with reasonable speedups (35X increase in simulation speed for 0.3% error) to reasonably accurate with extreme speedups (223X simulation speedup for 3.0% error).

5.1 Introduction

"Graphics processing unit" is now an inadequate term to describe a piece of hardware with a domain extending well beyond graphics applications. As programmers realize the unique advantages of GPUs (e.g., wide availability on commodity machines, extremely high throughput on parallel tasks, fast memory accesses), many non-graphics applications are being ported from their original CPU implementations to GPU versions. Such *computational GPU* applications are now commonplace in a range of fields including scientific computing [194], computer vision [64], finance [225], and data mining [150].

GPU architects must deliver improved hardware designs to meet the computational needs of these varied applications. A major barrier in achieving this is the massive overheads associated with detailed micro-architectural performance simulations. Simulators execute a program up to 2 million times slower than native execution [42, 141], depending on the simulator and the level of detail in the information recorded. These slowdowns are further compounded when hardware designers need to repeatedly re-run applications to test thousands of design space choices.

These prohibitively large simulation times force architects to focus their evaluation on graphics kernels (potentially neglecting important computational workloads) or to evaluate computational workloads using only kernels rather than full applications. Thus, there is a great need in the computer architecture community for detailed analyses of commercially-sized computational GPU applications without the overheads of full-program simulation.

This work addresses that need in three ways.

- First, it provides **a fast profiling tool that measures performance statistics**

as applications run natively on existing hardware (Section 5.3). This new, industrial-grade tool, called *GT-Pin*, can collect a variety of instruction-level data to inform hardware design. Profiling with GT-Pin typically takes 2-10 times as long as normal execution, does not perturb program execution, and requires no source code modifications or recompilation.

- Second, we use GT-Pin to conduct **a characterization study of very large OpenCL programs, averaging 308 billion dynamic GPU instructions apiece (Section 5.4)**. The commercial and benchmark applications studied are substantially larger than any OpenCL programs that have been characterized publicly. The statistics reported include dynamic instruction counts, breakdowns of memory, control, computation, and logic instructions, kernel and basic block execution counts, SIMD lengths, and memory access information. This characterization reveals a breadth of computational GPU workloads that indicates an even greater need for comprehensive simulation when evaluating future GPU designs.

- Finally, we **demonstrate how to select small, representative subsets of OpenCL programs to accelerate the simulation of future GPU architectures (Section 5.5)**. These small subsets can be simulated in lieu of full programs in a fraction of the time, while still providing an accurate evaluation of the applications' performance on future hardware. The selection process uses GT-Pin profiling and a little post-processing, but itself requires no simulation. This is a key contrast to prior work in CPU subset selection [32, 216] that allows us to make selections *even for applications that are prohibitively expensive to simulate in full a single time.* Developing this methodology required several innovations including how best to break GPU execution into intervals, how best to characterize those intervals, and how to rapidly find the best combination of interval and characterization for any given application. The resulting methodology offers an exploitable tradeoff between simulation accuracy and speed, for example speeding simulation by 35X for 0.3% error or speeding simulation by 223X for 3% error.

These new means of exploring large computational applications enable computer archi-

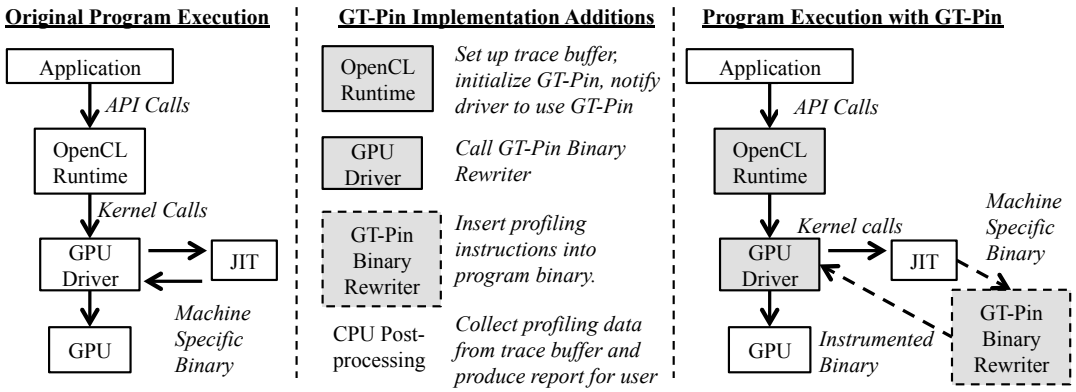


Figure 5.1: **The GT-Pin Implementation** makes multiple changes to the OpenCL runtime and the GPU driver, and adds a new GT-Pin binary re-writer and a CPU post-processor. From a user perspective, however, the tool is easy to use and non-intrusive, with low overheads, no perturbation, and no source code modifications or recompilation required.

tects to rethink and optimize GPU designs for the burgeoning diversity of workloads now being targeted to GPUs.

5.2 Background

This section provides readers with a brief history of simulation acceleration, and discusses the relevant terminology and concepts of OpenCL that are needed to understand this work.

5.2.1 Simulation Acceleration

Microarchitectural performance simulation is an important tool for the early (pre-fabrication) design of computer architectures. Unfortunately, microarchitectural simulation can be prohibitively slow, particularly when one wishes to study the performance of a range of applications on a large design–space's worth of architectures. As a result, one of the biggest research challenges related to simulation in recent years has been how to make it faster. This section provides a brief history of the work in this area of simulation acceleration; a more thorough discussion can be found in Eeckhout's Synthesis Lecture on computer architecture performance evaluation [55], and a comparison of the background work to this present work will come later in Section 5.6.

A variety of methods have been explored to accelerate simulation. One method is to collect only a carefully selected set of program and architecture performance metrics rather than all kinds of metrics and then *synthetically* generate a complete simulation trace (e.g., [18, 31]). Another method is to reduce the set of applications to simulate, choosing ahead of time those that will exercise the architecture in the most diverse ways (e.g., [112, 179]). Parallelizing the simulation process in different ways is another effective means of acceleration (e.g., [139, 175]). A potentially complementary method of accelerating simulation is called *sampled simulation*. Sampled simulation involves simulating subsets of applications to estimate performance in lieu of simulating full programs. Subsets may be selected *randomly*, as Laha et al. did for evaluating cache performance [135], or as Conte et al. did for full processor performance [43]. Subsets may also be selected *periodically*, that is, at fixed intervals within the full program's execution. Two works that use periodic sampling are SMARTS [259] and Flexus [256]. An advantage of periodic sampling over random sampling is that a confidence interval can be constructed via the central limit theorem to account for how accurate a selection may be; a disadvantage is that periodic sampling may inadvertently coincide with periodic behavior in the program (for example, with the subsets always occurring at low performance periods), leading to a skewed estimate of whole program execution. To prevent skewed subset selection, a new methodology for sampled simulation was introduced, called *representative* sampling. Representative sampling chooses subsets based on statistical performance indicators, aiming to select a those that combine to replicate full program performance. The first representative sampling work by Skadron et al. [221] chooses only one subset to represent the whole program, but later works selected multiple program subsets (e.g, [134, 186, 216]).

Perhaps the most famous of the representative sampling methodologies is SimPoint, by Sherwood et al. [216]. The work introduces a procedure that has now become an industry standard to select simulation subsets. The procedure works as follows: **(1)** Profile the program. **(2)** Divide the program trace into *intervals* that serve the dual purpose of encapsulating periodic program behavior and marking the starting and stopping points of the simulation subsets (that will be selected in future steps). **(3)** For each interval, construct a unique *feature vector* that reflects the interval's architectural features. The *feature vector's*

entries count the dynamic occurrences of select runtime events such as the execution of a particular basic block or procedure. **(4)** Group similar feature vectors into a small number of *clusters* (e.g., 10) using machine learning. **(5)** Choose a representative feature vector per cluster, typically the centroid. Additionally, compute a *representation ratio* per cluster, by dividing the number of total dynamic instructions across intervals in the cluster by the number of total dynamic instructions in the whole program. This metric gauges the impact a given cluster has on overall program performance. **(6)** The small number of intervals to which the chosen feature vectors belong make up the selected simulation subset. Simulate this subset of program intervals in detail, while ignoring the remainder of the program by fast-forwarding or check pointing. **(7)** Extrapolate the full-program performance from the results of simulating the representative subset. To do this, simply take the average of each interval's simulated performance, weighted by the representation ratio. Later, in Section 5.5, we adapt this standard procedure for selecting representative subsets originally designed to accelerate CPU simulation to something more suitable for GPUs.

5.2.2 OpenCL

The GT-Pin tool, the benchmark analyses, and the simulation speedup methodology are all based on OpenCL programs and programming concepts. Unlike other GPU languages, such as CUDA which is specific to NVIDIA, OpenCL programs can run on any heterogeneous architecture from any vendor. This chapter uses a number of OpenCL keywords which we briefly introduce here; a more comprehensive discussion of OpenCL can be found elsewhere [169].

OpenCL programs consist of two parts. A *host*, which uses *API calls* to manage the program's execution, and *kernels*, which are procedures that define computational work for OpenCL *devices*. OpenCL devices can be any mix of processing units, for example multiple GPUs and CPUs, but in this chapter the device is always a GPU. To manage OpenCL kernels, the host must determine the available devices, set up device-specific memory, create kernels on the host, pass arguments to and run kernels on target devices, and organize any results returned by the kernels. Each of these tasks is completed via built-in OpenCL API calls. For example, one named `clSetKernelArg`, as its name implies, sets an argument to

an upcoming kernel.

The API call `clEnqueueNDKernelRange` is particularly important in this work. This call dispatches a kernel to a device, signaling that GPU computation is commencing. Also relevant to this project are a set of API calls that manage *synchronization*. Synchronization calls constrain the order of other API calls, enforcing the desired sequences of events. Until a synchronization call forces coordination, for example to make a memory transfer, kernels execute on the devices asynchronously to the host program. OpenCL has seven synchronization calls: `clFinish`, `clEnqueueCopyImageToBuffer`, `clWaitForEvents`, `clFlush`, `clEnqueueReadImage`, `clEnqueueCopyBuffer`, and `clEnqueueReadBuffer`. Because these calls are the only points where host and device work are guaranteed to align, they constitute a natural and necessary point to start and stop device (in our case, GPU) simulation. Thus, in Section 5.5.2, we will use synchronization calls as one potential means to divide a program's execution into intervals.

Another OpenCL concept relevant to this work is the notion of *global work size*. Supplied as an argument to `clEnqueueNDKernelRange` calls, the global work size defines the total amount of work to be done on a given device, so that larger global work sizes take more execution time.

5.3 Tracing GPU Programs with GT-Pin

GT-Pin serves a community need for a fast, accurate, flexible, and detailed tool to profile commercial-scale native OpenCL GPU applications. This section describes how GT-Pin collects profiles within OpenCL execution environment and discusses the kinds of profiling data it can collect.

5.3.1 Instrumenting within the OpenCL Runtime

GT-Pin, which was inspired by the CPU tool, Pin [148], collects profiling data via dynamic binary instrumentation. Instrumentation, which involves injecting profiling instructions into program code, allows for much faster profiling than simulation. GT-Pin uses binary instrumentation, which while harder to implement than static compiler instrumentation

(because it necessitates GPU driver modifications), has the additional benefit of not requiring program recompilation. At a high level, GT-Pin's instrumentation injects instructions into binaries' assembly code as the they are just-in-time (JIT) compiled. These insertions later output profiling results as the program executes natively on the GPU.

To describe how GT-Pin inserts profiling code into OpenCL programs, we first describe how a normal OpenCL execution works. The left side of Figure 5.1 illustrates an uninstrumented OpenCL application's execution. First, the application communicates with the OpenCL Runtime by making API calls. Then, when `clEnqueueNDKernelRange` calls are made, the OpenCL Runtime passes the associated kernel source and arguments to the appropriate device driver, in our case a GPU driver. The GPU driver JIT-compiles the kernel source, typically when a `clBuildProgram()` API call is issued. Finally, the compiled, machine-specific binary code is passed along to the GPU for execution.

GT-Pin modifies this process at two points, as shown in the middle and right sides of Figure 5.1. First, when the OpenCL runtime is initially called upon by the application, GT-Pin intercepts the call and inserts a GT-Pin initialization routine, which notifies the GPU driver that GT-Pin has been invoked. At this time, a memory space called a *trace buffer* is allocated using malloc. The trace buffer is accessible by both the CPU and GPU and will be used to hold profiling data.

The GPU driver is the second point where GT-Pin must make modifications. After the driver compiles the kernel source code into machine-specific assembly, rather than allowing the driver to send the binary directly to the GPU for execution, the binary is diverted to a GT-Pin *binary re-writer*. The binary re-writer inserts profiling instructions into the program's assembly code. The injected instrumentation differs depending on the profiling data GT-Pin's users wish to collect. For example, to track dynamic basic block counts, GT-Pin adds instructions to initialize a basic block counter at the program's start, to update a counter at each block, and to write the final counter value to the trace buffer at the program's end. Once the re-writer finishes inserting profiling instructions, the GPU driver passes the instrumented binary to the GPU. Then, as the program executes, profiling data is sent to the trace buffer.

Finally, when GPU execution concludes, GT-Pin has the CPU read the profiling results

from the trace buffer to post-process the data and generate a user report.

5.3.2 Types of information that GT-Pin can collect

GT-Pin can observe everything that is happening at the level of both the kernel source and machine-specific binary, so it is able to capture many kinds of profiling data, including:

- static and dynamic instruction execution counts for the source and assembly;
- static and dynamic distributions of opcodes;
- static and dynamic SIMD width counts;
- static and dynamic basic block counts;
- thread cycles in kernel and non-inlined functions;
- latency for memory instructions per thread;
- cache simulation through the use of memory traces;
- memory bytes read and written per instruction; and
- utilization rates of per execution unit SIMD channels.

To reduce overheads, users may collect only the desired subset of these statistics by writing custom profiling tools. For example, for the simulation subset selection in Section 5.5, we wrote a custom GT-Pin tool that collected only instruction counts and opcodes, basic block counts, and memory bytes read and written per instruction.

5.3.3 Overheads

Like Pin, GT-Pin guarantees that the side-effects of inserting instructions do not perturb program execution. During instrumentation, GT-Pin minimizes the number of inserted instructions. For example, when counting dynamic instructions, GT-Pin inserts counter increments only once per basic block rather than per instruction. To profile timing events (e.g., thread cycles spent in kernels), GT-Pin inserts a simple timer call, which reads the event timer register. For this type of tracking, we observed a less than 10 cycle per timer

read overhead. From a user perspective, GT-Pin profiling runs take only a little longer than uninstrumented executions. While collecting data for the benchmark characterization study of Section 5.4, we observed 2-10X overheads. These overheads are very small when compared to the up to 2,000,000X greater slowdowns required to collect the same information through simulation.

5.4 A Study of Large OpenCL Applications

This section presents performance data relevant to GPU design for 25 commercial and benchmark applications shown in Table 5.1. All of the programs are written in OpenCL, and come from three sources. First, there are 15 applications from the CompuBench CL 1.2 desktop and mobile suites [132]. These applications include domains such as computer vision, physics, image processing, throughput, and graphics. Next, there are three applications from the SiSoftware Sandra 2014 suite [220], including two cryptography benchmarks and a GPU performance benchmark. Finally, there are seven video rendering benchmarks from the Sony Vegas Pro Test Project [228]. Sony Vegas Pro 2013 is a video editing tool [227], and the seven benchmarks are pieces of a press release project, each demonstrating different kinds of video attributes such as crossfades and Gaussian blurs.

5.4.1 Experimental system

All applications and benchmarks were run on a machine with an Intel Core i7-3770 CPU and an Intel HD 4000 GPU, both of the "Ivy Bridge" generation. As depicted in Figure 5.2, the HD 4000 has 16 execution units (EUs) organized into two subslices. The EUs are

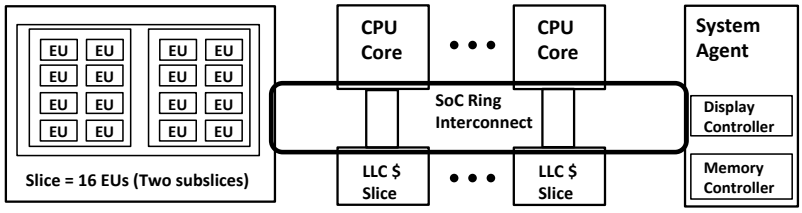


Figure 5.2: **The Processor Architecture** of our test system, which has an Intel Core i7-3770 CPU and HD 4000 GPU.

Source	Applications
CompuBench CL 1.2 Desktop [132]	Graphics T-Rex, Physics Ocean Surf, Physics Part Sim 64K, Throughput Bitcoin, Vision Facedetect, Vision Tv-l1-of
CompuBench CL 1.2 Mobile [132]	Graphics Provence, Gaussian Buffer, Gaussian Image, Histogram Buffer, Histogram Image, Physics Part Sim 32K, Throughput Ao, Throughput Juliaset, Vision Face Detect
SiSoftware Sandra 2014 [220]	Crypto Aes128, Crypto Aes256, Processor GPU
Sony Vegas Pro 2013 [227]	Press Project Region 1, Region 2, Region 3, Region 4, Region 5, Region 6, Region 7

Table 5.1: **Benchmarks used in this study.**

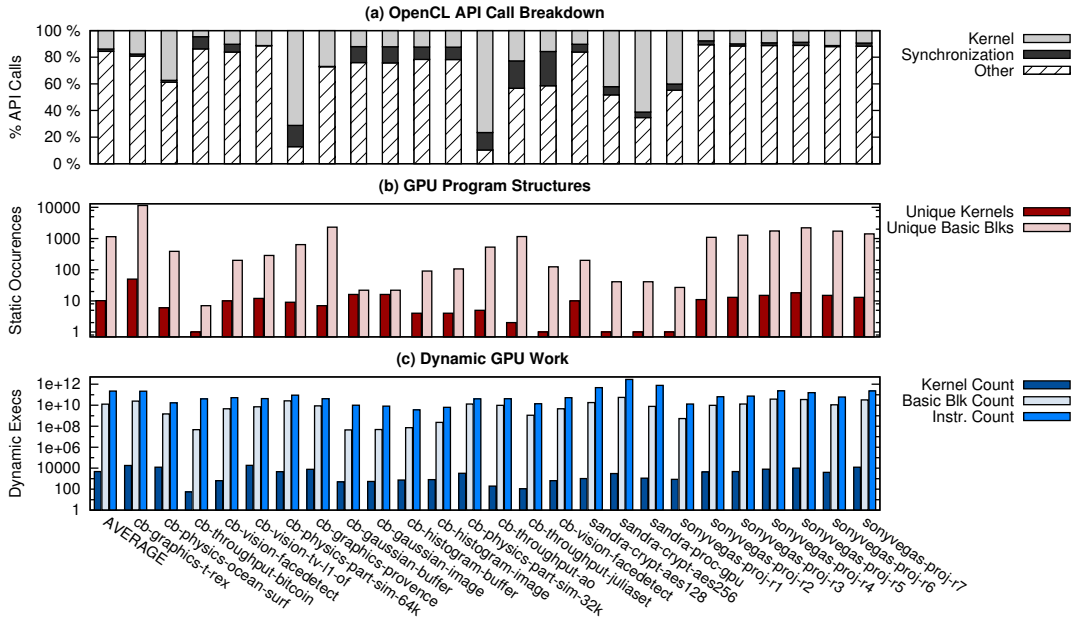


Figure 5.3: **Benchmark Characterization.** OpenCL call breakdowns (% synchronization, kernel, and other API calls) were measured on the CPU *host* using CoFluent; program structure counts (unique kernels and static basic blocks) and dynamic work counts (executions of kernels, basic blocks, and instructions) were measured on the GPU *device* using GT-Pin.

simultaneous multi-threaded (SMT) processor units, which are highly optimized for floating point and integer computations. Each EU has 8 hardware threads per core for a total of 128 simultaneously executing hardware threads. The GPU can perform at a peak rate of 332.8 GFLOPS, and has a maximum frequency of 1150 MHz. The system has 16 GB RAM and runs the Windows 7 64-bit operating system. OpenCL Version 1.2 is used for the runtime, and the GPU driver version is 15.33.30.64.3958.

5.4.2 Profiling results

At program execution time, the CPU was specified as the OpenCL *host*, and the GPU was specified as the *device*. As a convention, data reported at granularities smaller than a kernel invocation (i.e., one execution of a `clEnqueueNDRangeKernel`) are aggregate counts across hardware threads.

Calls between the CPU and GPU (Figure 5.3a.) First, we examine how the CPU and GPU communicate through the OpenCL API. GT-Pin tracks only GPU instructions, so we used the Intel CoFluent CPR API tracing tool [41] to count and categorize OpenCL API calls made by the CPU. To collect the name and arguments of every runtime API call, CoFluent intercepts the calls at execution time just before they application passes them to the OpenCL driver. Application performance is unaffected by this capture. Figure 5.3a divides the API calls made by our 25 applications into three types: *kernel invocations* (i.e., `clEnqueueNDKernelRange` calls), *synchronization* calls (those previously listed in Section 5.2), and *other API* calls, which include program setup, post-processing, and cleanup and supply arguments to kernels. Since the results are reported as percentages, the figure does not show that the 25 applications vary significantly in terms of the total number of OpenCL API calls, from just over 700 calls to over 160,000 calls. The applications are somewhat more consistent in terms of their usage of synchronization and kernel calls. Most applications initiate GPU work through kernel calls with about 15% of the total API calls, though in the case of `throughput bitcoin` and `physics part-sim 32K`, use as few or as many as 4.5% and 76.5%, respectively. Synchronization calls unsurprisingly tend to comprise only a small percentage of the total calls, on average 6.8%, and for the majority of applications less than 3%. The application that uses the highest proportion of synchroniza-

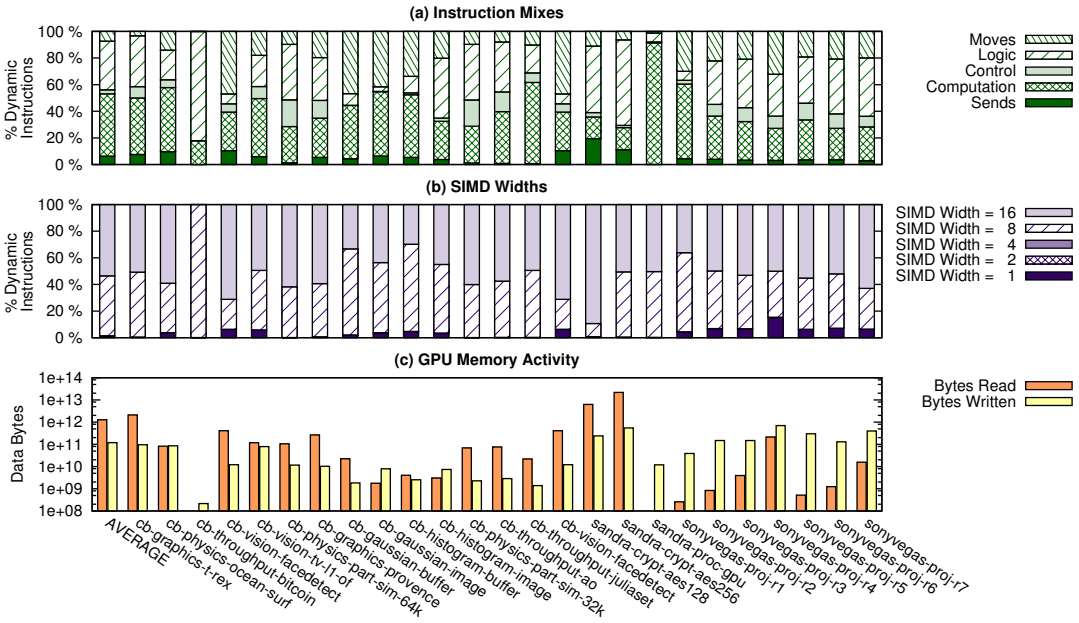


Figure 5.4: **GPU Work.** GT-Pin can also measure GPU instruction mixes, the SIMD widths of instructions (i.e., how data-parallel an application is), and the cumulative number of bytes read and written to memory across hardware threads.

tion calls (`throughput juliaset` at 25.7%) has the fewest total API calls of any program at 703.

GPU program structures (Figure 5.3b.) Using GT-Pin, we next profiled the static program structures created within the kernels. The *unique kernels* counted in the first set of bars in Figure 5.3b are the GPU's analogue of CPU procedures. Applications vary widely in the number of unique kernel programs they contain, ranging from 1 to 50 kernels, with a mean of 10.2. Looking at a smaller granularity, we found that each program has at least 7 and at most 11,500 unique basic blocks within these kernels, with a mean of 1139.

Dynamic GPU work (Figure 5.3c.) The number of unique kernels has little correlation with the number of kernel invocations (initiated by `clEnqueueNDKernelRange` calls), which range from 55 to over 18,000, with a mean of 4764. Inside the kernels, 3.7 billion to 2.9 trillion GPU instructions were executed depending on the application (with a mean of 227 billion), within 44 million to 180 billion total basic block executions (on average, 13 billion).

Dynamic instruction mixes (Figure 5.4a.) Figure 5.4a shows the percentage of op-

codes in five categories including *logic*, *control*, *computation*, *send*, and *move* instructions. The logic instructions, which include `and`, `or`, `xor`, `shift`, and `compare` instructions among others, are heavily used, as are the `mov` instructions. This is to support vector operations, such as loading vectors and arithmetic operations within vectors. The *control* instructions account for a smaller overall proportion, at an average of 7.3% of total instructions, and computation instructions account for 36.2% of the total instructions. The `proc gpu` application stands out with a relatively large proportion of computation instructions (91%), because it is designed to stress-test GPU performance. In GEN ISA, Intel GPU's instruction set architecture [94], *send* instructions make up all of the memory communications between hardware threads and execution units. In our applications they account for 5.1% of the overall instructions across applications.

SIMD vector lengths (Figure 5.4b.) In general, the applications take reasonable advantage of data-parallelism. All use a large proportion of 16- and 8-wide SIMD vectors: they comprise 52% and 45% of the instructions, respectively, across applications. Single-width instructions are just 4% of the instructions on average, 4-wide instructions are much less common (<0.1% across all applications, and 0.3% of the 6 applications that do use them), and 2-wide instructions are never used.

Memory operations (Figure 5.4c.) Finally, we tracked the cumulative bytes read and written to memory across all GPU hardware threads. The two cryptography applications read the most, at 624 and 2174 GB apiece. The seven Sony video rendering applications were on the high end of writes, and tended to write many more bytes (up to 525X more for `proj-r5`) than they read. On average across all applications, however, the opposite was true: an average of 105 GB were written and 1110 GB were read.

5.5 Selecting GPU Simulation Subsets

As we just saw, computational GPU benchmarks can be extremely large. Simulating such large computational benchmarks to determine their performance on future architectures is a problem that until now has been unaddressed. GT-Pin profiling can be used to speed simulation by providing the information necessary to choose small, representative program

subsets to simulate from within the large benchmarks. Unlike prior work in simulation subset selection [32, 216], the selection process itself does not require simulation, allowing us to target extremely large applications that may be prohibitively long to simulate in full even a single time.

This section describes our GT-Pin-enabled GPU simulation subset selection methodology. GPUs pose a number of unique challenges to address versus existing CPU selection methodologies (Section 5.5.1). In the experiments that follow, we explore how computational GPU programs can be represented as temporal intervals and architectural features (Section 5.5.2), how to rapidly identify the best interval and feature set for a given application (Section 5.5.3), and how to trade simulation time for accuracy (Section 5.5.4). Finally, we validate that the selections made based on one profiled execution are accurate across multiple execution trials on different processor architecture generations (Section 5.5.5).

5.5.1 Adapting CPU Simulation Acceleration to GPUs

To adapt the procedure of CPU simulation acceleartion via representative sampling to GPUs, we had to answer several open-ended questions. First, *how to build a GPU selection methodology that is architecturally independent* and not tied to a specific GPU platform or ISA. To achieve architectural independence, we based our methodology around OpenCL programming units and concepts. The next challenging decision was *how to divide the program into intervals*. Interval division 1) must not pose synchronization problems, 2) must strike a balance between being large enough to capture periodic behaviors but not so large as to capture multiple types of behaviors, and 3) must have appropriate boundaries for later simulation, since intervals mark the start and stop points of the selected subsets. According to GPU hardware designers we spoke with, it is a strict limitation that any GPU simulation subset selections be at least a full kernel call in length and that they do not span multiple OpenCL synchronization calls. Another open question was *what feature vectors will accurately summarize the behavior of a GPU execution interval.*

		Intervals per Program		
Interval Bound	**Relative Size**	Min	Avg	Max
Synchronization calls	*large*	56	545	2115
~100M instructions	*medium*	55	916	3121
Single kernel boundaries	*small*	55	4749	18157

Table 5.2: **The Program Interval Space** explores three different ways of dividing GPU program traces into intervals.

Feature Key	**Identifier**
Kernel	KN
Kernel, Argument Values	KN-ARGS
Kernel, Global Work Size	KN-GWS
Kernel, Argument Values, Global Work Size	KN-ARGS-GWS
Kernel, # Bytes Read, # Bytes Written	KN-RW
Basic Block	BB
Basic Block, # Bytes Read	BB-R
Basic Block, # Bytes Written	BB-W
Basic Block, # Bytes Read, # Bytes Written	BB-R-W
Basic Block, # Bytes Read + # Bytes Written	BB-(R+W)

Table 5.3: **The Program Feature Space** explores ten feature vectors, with the above keys and values that count the dynamic execution count of the respective key.

5.5.2 GPU interval and feature exploration

To answer these questions, we ran experiments using three types of interval divisions and ten types of feature vectors. In each of these experiments we used the profiling information from GT-Pin and CoFluent to divide the execution into intervals and populate the feature vectors.

Interval space. Previous CPU work divides program traces into uniform intervals of a given number of dynamic instructions, for example 100M instructions [216]. However, such rigid divisions will not work on a GPU as they violate the constraint that GPU intervals should not span kernel boundaries or synchronization calls. Instead, we experiment with three variable length interval sizes summarized in Table 5.2. Synchronization intervals are the largest division, splitting traces at each OpenCL synchronization call. The next smallest intervals further subdivide these into roughly 100M dynamic instruction segments. In order not to split an interval across kernels or a kernel across intervals, this results in some intervals that are slightly larger or smaller than exactly 100M instructions, so we call the division "Approximately 100M instructions". Finally, we consider each kernel invocation its own interval. While some kernels are larger than 100M instructions, most are not, resulting in the smallest average interval size.

Feature space. Having broken a program into intervals, the second question is which program features to use to characterize that interval for clustering. We experiment with the ten types of feature vectors summarized in Table 5.3. Each feature vector is essentially a set of (key,value) pairs, where the key is a distinct program event such as "calls to kernel foo" or "calls to kernel foo with argument 256", and the values are counts of the number of times this event occurred in a given interval. As Table 5.3 shows, our experiments explore whether there is value in increasing the specificity of events to include not only computational information such as kernel or basic block ID, but also data interaction such as the kernel arguments or the number of bytes read or written.

To ensure that these vectors place appropriate value on differently sized kernels and basic blocks, we weight each vector entry by instruction count. For example, if an interval executes block `A` 10 times and block `B` 5 times, these counts alone would suggest that `A` is a more important feature of this interval. However, if `A` were 3 instructions long and `B` were

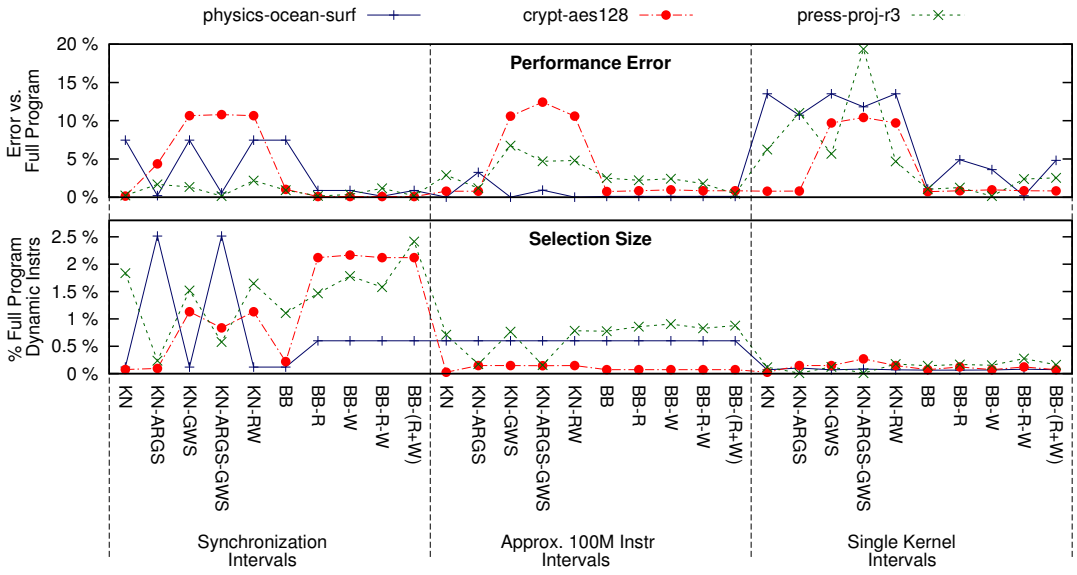


Figure 5.5: **Feature and Division Space Exploration.** Applications vary in terms of which of 10 different feature vector choices and 3 interval division sizes are best able to select subsets that match full program performance. Also, the most accurate selection configurations are not always the best at reducing the number of instructions to simulate.

20, then the weighted score of $5 \times 20 = 100$ for `B` versus $10 \times 3 = 30$ for `A` will better reflect their actual importance. Note that this weighting will also impact the cluster representation ratios that are computed in the next section.

Quantifying simulation error. Once intervals have been divided and feature vectors constructed, any tool can be used to cluster and score them. We used the standard tool from prior CPU work, SimPoint. Specifically, we used SimPoint version 3.0 which can handle variable-sized intervals [77]. SimPoint takes program feature vectors as input, and uses the *k-means* clustering algorithm to group similar feature vectors. It then computes the centroid of each cluster, based on the total element count of each vector and returns these centroids. We trace these reported feature vector centroids back to their associated intervals to get our simulation subset selections. Along with the cluster centroids, SimPoint also returns representation ratios for each of the selected feature vectors. SimPoint allows users to specify the maximum number of clusters and thus selections, but may return fewer than this maximum if its machine learning algorithm judges it appropriate to do so. The

maximum clustering and therefore selection subset count is set to 10 in all the experiments that follow.

Traditionally, detailed simulation of a full program is used to evaluate the representativeness of the selected subsets. However, since we needed to evaluate 30 interval size/feature vector configurations for our 25 large applications, detailed full-program simulation was out of the question. Instead, we developed a heuristic for validating individual selections based on per-kernel timing data, which we collected with the CoFluent CPR tool. The validation heuristic is an error percentage of the measured whole program *seconds per instruction* (SPI) versus the projected whole program SPI, extrapolated from the selections' timings and weights:

$$\text{Error} = \frac{abs(\text{Measured SPI} - \text{Projected SPI})}{\text{Measured SPI}} * 100\% \tag{5.1}$$

To get the measured seconds per instruction of the whole program, we divide the combined time in seconds taken by all of the kernel invocations by the total number of dynamic instructions executed by all of the kernel invocations. To get projected SPI, we first find the SPI per selected interval, dividing the sum of CoFluent reported time in seconds of the kernels in the selected interval by the sum of dynamic instruction execution counts reported by GT-Pin for the kernels in the interval. Then, we multiply each selected interval's SPI by its SimPoint ratio, and add these products together to get the projected whole program SPI.

Interval and feature exploration results. Figure 5.5 shows how the 30 types of interval/feature vector combinations fared in terms of selecting representative program subsets. The figure presents error and selection size results of just 3 sample applications, but we tested all 30 combinations on all 25 applications. The results of the remaining 22 applications lead to the same conclusion as the 3 shown: no single combination of interval size and feature vector is "best" in terms of error or selection size across all applications. There are, however, several trends across applications. For example, basic block based features tend to outperform kernel based features, and features with memory access counts improve basic block based features for most applications. The applications with the fewest unique kernels tend to have high error rates when kernel-only features are used.

As for interval size, synchronization-bounded intervals tend to produce the smallest

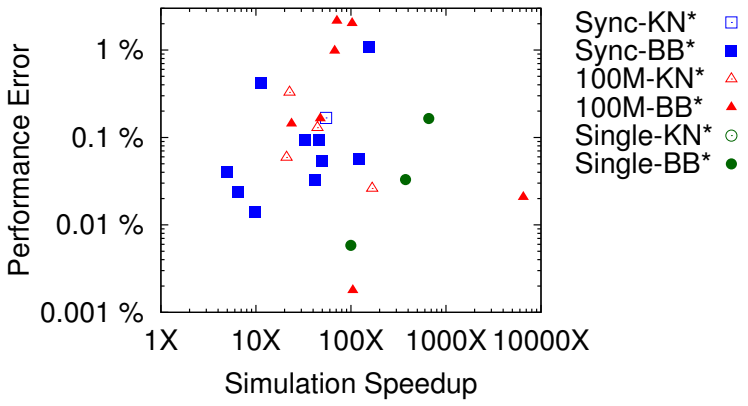


Figure 5.6: **Optimizing Selection to Minimize Error** results in individual applications choosing different interval/feature vector configurations. Across applications, errors average 0.3% and simulation speedups average 35X, ranging from 6X to 6509X.

errors, but since they are also the largest division, they produce the largest selection sizes. For basic block based features, interval size tended to have less of an effect on error rate than the effect of interval size on kernel based features. If we were to choose the best average interval size/feature vector combination of the 30 tested, the combination with the smallest error rate would be basic block intervals with no memory features (BB), and synchronization bounded intervals. This combination averages 1.5% error across all 25 applications, and selects subsets that are 1.9% of the total program instructions (corresponding to a 53X simulation speedup). In the worst case, one individual application has an error of 8.8% and another application has a selection containing 24.0% of the total program instructions.

5.5.3 Identifying application specific intervals and features

To improve these error and selection size numbers, we can leverage the fact that the combination that works best for one application is not always what works best for another. Rather than choosing one universal interval and set of features, we can choose the best interval and features *for each individual application*. Somewhat counter-intuitively, there is almost no additional overhead for doing so, as we need to profile (natively) each application just once to characterize the error and selection sizes for all 30 interval and feature vector combinations.

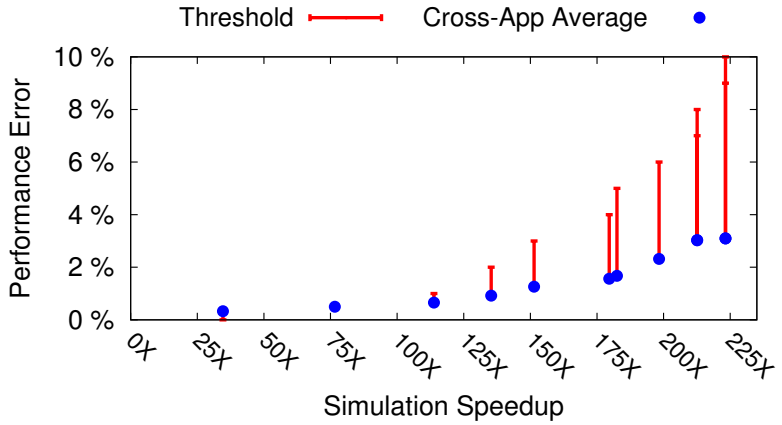


Figure 5.7: **Optimizing for Both Error and Selection Size** means choosing the per application configuration that has the smallest selection size with an error below a given *threshold*. For example, with an error threshold of 3%, simulation speedups average 223X.

Picking the error-minimizing interval and feature combination for each individual application achieves an average error rate of just 0.3%, with the worst case error being 2.1% for the `histogram buffer` application. Figure 5.6 shows the error-minimal configuration for each of the 25 applications. Of the 25 applications, only 5 chose kernel-based features while the remainder chose basic block features.

As for interval sizes, 3 applications chose single kernel long slices, 11 chose synchronization bounded slices, and 11 chose 100M instruction slices. Memory-based features were chosen by 20 of the 25 applications. These diverse choices in best configuration support our previous observation that no single configuration is suitable for all applications.

5.5.4 Co-optimization of simulation time and error

Minimizing the error without regard to simulation speedup may still result in subsets that are too large for certain simulation needs. Across applications, this policy resulted in an average simulation speedup of 35X, but just 6X in the worst case. To improve these numbers, we tried jointly-optimizing for error and selection size. By setting an acceptable error *threshold* rather than aiming to minimize error, we can greatly accelerate simulation. Specifically, we choose the per-application configuration with the smallest selection size that

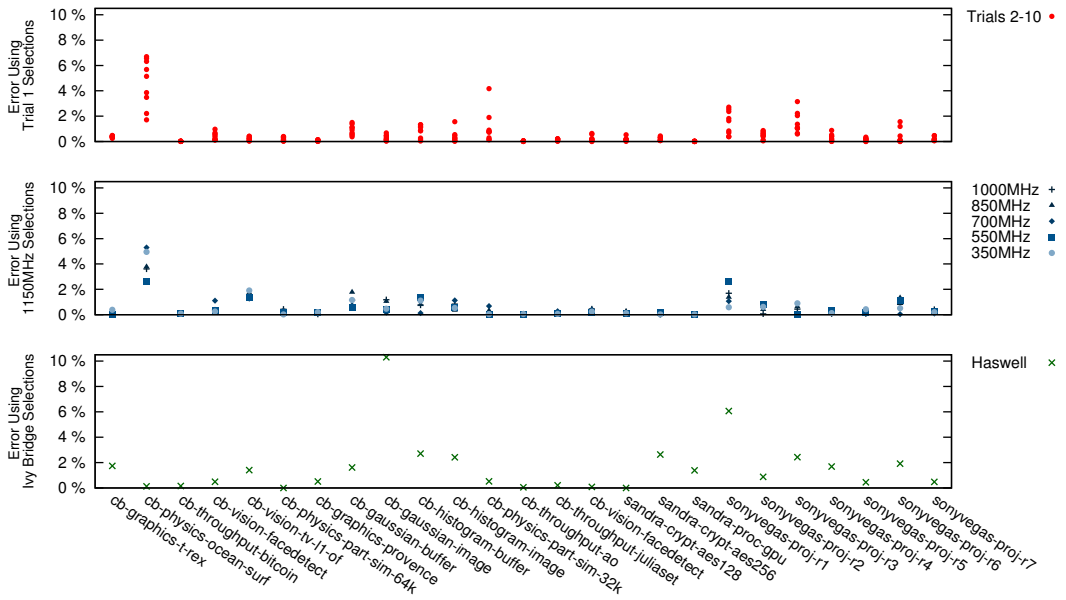


Figure 5.8: **Timed Validation.** One trial's selection are still accurate across trials, frequencies, and architecture generations.

also has an error below a series of given thresholds. If no configuration had an error below the specified threshold, we choose the configuration with the smallest error, regardless of selection size. Figure 5.7 shows the results of this experiment. The furthest left point on the plot shows the cross-application average error and simulation speedup when selection configurations are chosen to minimize error. The remaining points show error thresholds of 0.5% and 1% to 10% at steps of 1%. As error thresholds are relaxed to higher values, the speedups monotonically increase. At the far right end of the graph, when we set the error threshold to 10%, we get an average error across applications of 3.0% and an average simulation speedup of 223X.

5.5.5 Validating the selections for future architectures

We have already seen that the selected subsets can accurately predict the performance of full-programs executed on the same hardware. Here, we test whether the selections built from one set of profiling data can predict full program execution across multiple trials run on different architectures.

Quantifying cross-trial and cross-architecture accuracy. To test a single set of

selections across trials and architectures, we first need to guarantee that the kernel calls contained in the selected intervals will be present and findable in future executions. Thanks to non-determinism, this is not automatically the case, but we can force it to be so with a *record* and *replay* feature of the previously discussed CoFluent tool. CoFluent's record mechanism captures API call data as it passes between the application and the OpenCL runtime. In addition to call names, the recorder captures configuration parameters, memory buffers and images, and OpenCL kernel code and binaries. This recorded information can later be replayed and runs just as a normal executable on native hardware would, with the only difference being a consistent and repeatable ordering of API calls.

We generate just one *original* set of selections and representation ratios per application using a CoFluent recording. We next verify this selection against measured SPIs computed from *new* replayed trials' timing and instruction data. Then, we compute the error of the original selection on the new trial.

Cross trial accuracy. Our first experiments tested the selection of one trial against multiple future trials on the same machine. The top plot of Figure 5.8 shows the resulting error rates for the new Trials 2-9 versus the original Trial 1, for each of the 25 benchmark applications. Most of the error rates are below 3% (with many below 1%), indicating that a single trial's selections can be successfully used to predict the whole program performance of other trials.

Cross frequency accuracy. To see how the selections hold up for future architectures with different processing rates, we next validated the original set of selections against timing data for new trials executed at varying GPU frequencies. All of the data previously reported in this chapter use the GPU's maximum frequency of 1150MHz, so the new frequency tests use lower frequencies, specifically at 1000, 850, 700, 550, and 350MHz. The middle plot of Figure 5.8 shows the resulting error rates. Again, most are less than 3%, indicating that the selections of a single frequency can be used to predict the whole program performance of executions at other frequencies.

Cross architecture generation accuracy. As a final experiment, we tested whether our selections could predict whole program performance across different GPU architecture generations. Specifically we used selections collected on our Ivy Bridge HD4000 GPU to

predict program performance on a newer Intel GPU, the HD4600 Haswell processor. The primary difference between the two processors is the number of execution units (EUs) within each GPU: the HD4000 has 16 EUs whereas the HD4600 has 20 EUs. To compare the two processors' raw performance, we ran LuxMark [149] on both machines. LuxMark is a popular cross-platform benchmarking tool, which scores GPUs on their ability to render different test scenes of varying complexity. The results (higher scores are better) were 269 for the HD4000 and 351 for HD4600, demonstrating the performance increases due to parallelism on the HD4600.

The bottom plot of Figure 5.8 shows the error rates of using HD4000 selections to predict HD4600 performance. Once again, most of the error rates are less than 3%, and the worst case application (`gaussian-image`, one of the shortest benchmarks in terms of kernel invocations) has 11% error. These results show that a single set of selections can predict the performance even on architectures with very different performance characteristics.

5.6 Related Work

This is the first work to characterize large OpenCL programs, and one of the first works to explore accelerating GPU simulation through the selection of representative subsets.

GPU application analysis. There are two related profiling tools from the Georgia Institute of Technology: Ocelot [51, 126] and Lynx [60]. Ocelot is a GPU compiler that instruments programs at compile time to measure various performance statistics. Unlike our work, Ocelot emulates programs rather than running them on native hardware, it also does not yet fully support OpenCL compilation. Lynx, a binary instrumentation tool that stems from Ocelot does support OpenCL execution on native hardware, but unlike GT-Pin, Lynx has only been demonstrated to work on small programs (their tested applications averaged 2 million times fewer dynamic GPU instructions than ours). Lynx also instruments the NVIDIA PTX instruction set rather than the GEN ISA, and does not offer any solutions for selecting simulation subsets as we do.

Several additional works also characterize GPU programs, although most study much smaller applications and focus on CUDA workloads, which are NVIDIA specific, as opposed

to architecture-independent OpenCL workloads. Zhang et al. [266] model the instruction pipeline, shared memory access, and global memory accesses of GPUs to accurately predict — and eventually improve — the performance of overlying applications. Their tested application has 6500 times fewer instructions than the average application studied in this work. Mistry et al. use built-in OpenCL API calls rather than an external profiler to analyze a computer vision algorithm [165] for kernel call durations, average and variations in time spent processing video frames, and GPU command queue activity. Their API-based profiling is much more limited in terms of the types of data it can collect versus GT-Pin's instrumentation-based profiling. Goswani et al. use an instrumented version of the GPGPU-Sim simulator [12] to collect a variety of data including instruction mixes, memory and branching statistics, and parallel execution activity for a large collection of benchmarks, but unlike GT-Pin their tool has hefty overheads, on the order of a million times the original program execution time.

Finally, there are a commercial tools that monitor program performance (e.g., [178]), but they do not measuring instruction-level metrics as we do.

CPU simulation acceleration. Since the simulation acceleration works discussed in this chapter's background section, dozens of papers have been published that extend the area. Here we address only those most relevant to this chapter, such as the PinPoints paper by Patil et al. [186]. Like our work, PinPoints uses dynamic instrumentation to find representative simulation subsets, but it does so only for CPU programs. Follow-up works by the same authors address a repeatability problem that arose between profiling and tracing runs [187] (we avoid this through the use of CoFluent recordings), and a toolkit for finding representative subsets deterministically and check-pointing them for Pin-based simulation of x86 programs [189]. As we do for GPUs, Lau et al. explore a variety of appropriate feature vectors for CPU simulation, finding that basic blocks, loop frequency counts, and register reuse counts work best to encapsulate interval behavior [138]. Finally, the recent BarrierPoint work by Carlson et al. [32] finds representative subsets in parallel OpenMP programs by aligning their interval divisions with synchronization points, much as we do by restricting our GPU programs to kernel invocation boundaries or greater.

GPU simulation acceleration. There are just two other works in the area of GPU

simulation subset selection. The first, by Huang et al. [91], finds representative GPU simulation subsets using a similar overall methodology to our work, with single kernel invocation intervals and compound feature vectors that include a metric analogous to our global work size, a memory request count, and measures of intra-kernel parallelism. Besides the differences in feature vector construction, the work differs from ours in two significant ways. First, they only demonstrate that their feature vector construction works for 12 very small applications, with an average of just 34 kernels invoked per application (versus our applications that average 4749 kernel invocations a piece). Second, while our simulation time savings come entirely from skipping *whole* kernel invocations, their savings come primarily from skipping *parts* of kernel invocations. The second GPU subset selection paper is a work by Yu et al. [262]. This work also reduces simulation sizes by choosing partial kernel invocations, but rather than having the simulator execute intra-kernel samples, they reconstruct reduced-loop count micro-kernels that can be simulated in full. It is possible that such an partial selection method could be combined with our method of skipping whole invocations for improved simulation speedups.

5.7 Limitations and Future Work

There are a couple of potential limitations to this work that lead to opportunities for future work.

Applicability of GT-Pin. The current version of GT-Pin works only on Intel architectures and for OpenCL programs, although the design concepts could be applied to GPU architectures from other vendors. This would require a new driver implementation and a new ISA specific binary re-writer per architecture type.

Cross-Generation Simulation Regions In our evaluation, we only compared the Ivy Bridge generated selections to the next generation of GPU architecture, Haswell. In the future, it would be prudent to compare the selections to further generations of GPU architectures, such as Broadwell, or to GPU architectures outside of the Intel family of processors.

5.8 Discussion

This chapter took three steps towards speeding up the design of GPUs for computational workloads and avoiding inefficiencies at hardware design time. First, it introduced a new, fast GPU profiling tool called GT-Pin, which measures a variety of instruction-level performance factors of applications as they run natively on existing GPUs. Next, it used GT-Pin to characterize 25 very large OpenCL benchmarks, exploring several features relevant to GPU design. Finally, it demonstrated that representative subset selection can successfully accelerate GPU design, by finding small but representative program subsets for GPU developers to simulate in lieu of full programs. These advances enable designers to optimize for the diverse set of computational workloads that are currently being developed for use on GPUs.

Chapter 6

Energy Efficiency Across the Stack

Modern demand for energy-efficient computation has spurred research at all levels of the stack, from devices to microarchitecture, operating systems, compilers, and languages. Unfortunately, this breadth has resulted in a disjointed space, with technologies at different levels of the system stack rarely compared, let alone coordinated.

This chapter presents[1] a remedy for this problem, conducting an experimental survey of the present state of energy management across the stack, and finding those that are most promising for reducing energy–inefficiencies. Focusing on settings that are exposed to software, we measure the total energy, average power, and execution time of 41 benchmark applications in 220 configurations, across a total of 200,000 program executions.

Some of the more important findings of the survey include that effective parallelization and compiler optimizations have the potential to save far more energy than Linux's frequency tuning algorithms; that certain non-complementary energy strategies can undercut each other's savings by half when combined; and that while the power impacts of most strategies remain constant across applications, the runtime impacts vary, resulting in inconsistent energy impacts.

[1] This work was previously introduced in a conference publication [115].

6.1 Introduction

Modern computational needs and resource constraints have promoted energy efficiency to a first order design goal, precipitating a wide array of energy conservation techniques from the circuit to the user and everywhere in between. Despite marked advances in energy efficiency, the anticipated constraints of future domains such as wearable or implanted computers necessitate continued advances.

The fragmentation of work between different communities is one obstacle to progress. Individual research papers tend to compare a new technique against the next closest, which rarely extends into other layers of the system stack. When the energy savings of a new technique are not compared to existing techniques at multiple levels of the stack, it is hard to evaluate the new idea's broader impact to energy research. Since experimental methods vary widely, using different hardware, versions of the OS, compilers and flags, languages, and benchmarks, comparing results across research papers is rarely a viable option. For example, some studies report power while others report energy, some measure power while others model it, and some report usage for the entire package while others report usage only for the cores. Accurately comparing a new technique to old techniques requires normalized experimental evaluation methodologies on similar software and architectural platforms.

Understanding how a research project fits into the quantitative landscape of existing work enables researchers to evaluate the new work's energy savings and tradeoffs in the proper context. For example, if one strategy decreases energy consumption by 50% but requires new hardware, it might be less desirable than an alternative that saves only 40% but uses commodity hardware. Or, a language extension that saves 200% of the energy of existing system level strategies may be more readily adopted into the language standard than one that saves only 20%. It is also important to understand how techniques combine, both in deployment and when discerning the most promising future research directions. For example, if a compiler level energy optimization complements an operating system level technique, both techniques merit further investigation regardless of which saves more in isolation. However, if one eclipses or eliminates the impact of the other, the lower saver may be less valuable.

To restore a broad context for software energy research, this work measures the rela-

tive power, performance, and energy effects of a range of energy management strategies. While most of the strategies we study have been previously evaluated in some context, this is the first time that all of the results can be compared, because our experiments have standardized the architecture, OS, measurement tools, and benchmarks. We examine each technique in isolation as well as in combination with other techniques at different parts of the system. Examining 220 experimental configurations of 41 applications totaling more than 200,000 trial runs, we juxtapose the energy impacts of frequency scaling, sleep states, parallelism, compiler optimizations, application-specific power caps, and source-level optimizations. These are some of our key findings.

There is only so much room to save power in software (Section 6.3). We found that the lowest system baseline power (i.e., the operating system running with no user applications) consumed 60% of serial application power, and 35% of the power of a well parallelized application. Moreover, single-threaded power varies relatively little across programs.

Linux does not provide energy-efficient frequency tuning algorithms (Sections 6.4.1, and 6.4.5). Add us to the chorus [136] noticing that Linux's energy-efficient frequency scaling algorithm, *ondemand*, is not great at its purported job. Particularly when applications were parallelized, ondemand often increased energy rather than saving it. The aptly named *powersave* algorithm does save some power but at great cost to performance, so it is also an energy loser.

Overclocking has little to no effect on energy (Sections 6.4.1 and 6.4.5). While overclocking saves runtime, it eats away a commensurate amount of power, resulting in no net effect on energy for most applications. At increased thread counts (e.g., 16 threads), overclocking's power increases begin to outstrip its runtime savings, meaning overclocking reduces energy by a small amount.

Parallelization can save so much energy relative to other strategies that energy-conscious software developers must embrace it (Section 6.4.3). Most desktop,

server, and mobile chips have multiple cores, each of which costs power even when unused. When these cores are utilized, the performance gains more than offset their power costs. For example, increasing parallelization from 1 to 16 threads saved energy for all the applications we tested — even the poorly scaling applications — for an average of 55% energy savings across applications.

Good compilation beats most other energy management techniques (Section 6.4.4). Performance-oriented optimizations (e.g., gcc's `-O3`) offer significant energy savings, with `-O3` optimized software consuming less than 43% of `-O0` optimized. As for power-oriented optimizations, despite research proposals dating back 20 years [239], modern compilers still do not explicitly optimize for, or significantly impact power.

Java programs require special energy attention, but they don't make it easy (Sections 6.4.4 and 6.5.1). Optimizing Java for energy is even more important than optimizing native languages. Not surprisingly, interpreted Java costs nearly 8X the energy of compiled Java. Additionally, prior work has found that Java is particularly prone to source-level inefficiencies, possibly in part from the development tools used to produce it [30]. Despite this, we observed that Java is challenging to manually optimize for energy.

Power-oriented source code optimizations are probably not worth the average programmer's time (Section 6.5.1). Source-level power tuning suggested by previous research [162] may be effective for tiny embedded programs but is challenging in larger programs. Despite hundreds of micro-optimizations across eight selected benchmarks, we were unable to produce significant power savings for any of the applications.

Idle states are very complementary to other techniques (Sections 6.4.5 and 6.5.4). Processor idle or sleep states saved energy — up to 19% — with nearly all of the energy management strategies we combined it with.

Non-complementary conservation strategies can undercut one another by half (Sections 6.5.3, 6.4.5 6.5.4). Not all of the management techniques play well together

and their benefits are absolutely not additive. For example, the 19% idle state savings can be cut in half when frequency is tuned to lower levels. However, none of the management strategies interfere so badly that they completely negate another strategy's effects when combined.

6.2 Background on Energy Management

To set the context for the techniques that this work measures, this section provides a short primer on energy management strategies. For a more complete survey, we refer the reader elsewhere [200, 245, 249, 255]. Although these techniques span many fields of computer science, they all boil down to two broad strategies: *reduce a computation's resource requirements* and *use no more than the required resources.*

Circuit One popular energy conservation technique is to turn off or turn down underutilized components. This is usually accomplished by reducing or stopping the clock and/or supply voltage. An integrated circuit's power consumption is the sum of the active ($P_{active} = \alpha \cdot C \cdot V_{dd}^2 \cdot f$) and leakage ($P_{leak} = V_{dd} \cdot I_{leak}$) power, where α is an activity factor determined by the dynamic switching activity in the circuit, C is the circuit's capacitative load, V_{dd} is the supply voltage, f is the clock frequency, and I_{leak} is the amount of leakage current. *Frequency scaling* reduces the clock for a linear reduction in active power, while *clock gating* stops it entirely. *Power gating* turns off current to idle components, while *dynamic voltage and frequency scaling (DVFS)* reduces supply voltage and frequency together. Targeting supply voltage is particularly effective as it reduces both active and leakage power, the latter of which accounts for up to 50% of total power today [97].

When applied to an idle or near-idle circuit (e.g., a processor executing a memory-bound workload) these techniques save power while minimally impacting application runtime, ultimately saving energy. The control policies to manage these settings is an active area of research, particularly with respect to emerging integrated voltage regulators [232], which are improving the spatial and temporal resolution of DVFS. These controls are increasingly being exposed to software, however; it remains to be seen what type of control policy is best.

Architecture Above the circuit, there is a huge volume of work in energy-oriented microarchitecture including cache tuning [130], on-chip networks [131], memory compression [17], and instruction speculation control [124]. The research community is also embracing heterogeneity in the form of specialized accelerators [70, 89] and asymmetric designs [29] such as ARM's big.LITTLE. Even the now mainstream chip multiprocessors originated out of a need to scale performance without increasing power density, so the software parallelization it forced could be considered part of the power-conservation landscape.

Platform Off-chip, there are numerous other strategies. DC to AC conversion, which consumes 0.9 Watts for every compute Watt [240], is unsurprisingly a focus of datacenter energy efficiency. Cooling, which incurs similar overheads, has also received significant attention (e.g., [191]). On laptops and mobile devices, reducing screen brightness and duty cycling for services such as GPS are other proven energy savers [14, 39].

Operating System Operating systems get involved by explicitly treating energy as another hardware resource to be managed [174, 243]. To save energy, they control software's interactions with lower level resources, for example adjusting DVFS on the fly [183], mapping processes to cores to keep total power below a cap [15, 208], or strategically offloading computation to achieve battery lifetime goals [246].

Compiler and Runtime Via static analysis, feedback directed compilation, or JIT compilation, compilers can analyze applications to insert hints about when to change frequencies [226], rearrange computation to create longer idle periods [8], and place instructions and data into memory in a more energy efficient manner – either by reorganizing instructions in the register file [213] or by creating a compiler-managed scratchpad [100]. There is also research on offloading compilation to a remote machine [145] to save energy and on power-saving hybrid garbage collection schemes [73].

Source and Language At the source level, energy optimization strategies range from micro-optimizations such as manual loop unrolling [48] to macro solutions like updating software development environments to encourage programmers to be more energy friendly [30].

Additionally, language extensions (such as EnerJ, which recruits programmer assistance in finding opportunities for power-accuracy tradeoffs [207]) and new languages (such as Eon, which has programmers identify high and low power energy regions at the source level [229]) have been proposed to improve energy efficiency.

Suite	Applications Used
Parsec 3.0	`blackscholes`*, `bodytrack`, `canneal`, `dedup`, `ferret`, `fluidanimate`*, `raytrace`, `swaptions`, `streamcluster`, `x264`
SPLASH-2X	`barnes`, `fft`, `fmm`, `ocean_cp`*, `radix`* `water_spatial`
Spec CPU 2006	`bzip2`, `gcc`, `mcf`, `hmmer`, `sjeng`, `milc`, `gromacs`, `cactusADM`, `astar`*, `lbm`*, `wrf`, `sphinx3`, `tonto`, `povray`, `GemsFDTD`, `gamess`, `omnetpp`
DaCapo 9.12	`avrora`, `h2`, `jython`, `luindex`, `lusearch`*, `pmd`*, `sunflow`
Spec JBB 2013	`pjbb2005` with 8 warehouses and 100,000 transactions.

* benchmark chosen for application-specific experiments

Table 6.1: **Experimental benchmarks**, chosen to represent a range of languages, programming styles, and application domains.

6.3 Experimental Design and Methodology

Good experimental design and methodology were crucial for this survey. This section describes and justifies the design choices we made.

Experimental System All the experiments in this chapter use a single, dedicated Dell PowerEdge R420 server. The server is dual socket with Intel Sandybridge E5-2430 chips, each with six cores and two-way hyper-threading for a total of 24 hardware contexts. The system has 24GB of DRAM and runs Ubuntu 12.04.2 with the 3.9.11 version of the Linux kernel, the latest release at the time of our first data collections. To allow the operating system and userspace to adjust certain controls such as frequency tuning, we switched the Dell BIOS settings to 'operating system control'. The machine runs gcc Version 4.6.3

compiler and Java HotSpot 64-bit server VM with JRE 2, build number 1.5.0.

Power Measurements For all of the power and energy measurements, we use Intel's Running Average Power Limit, or RAPL, interface [45]. RAPL uses non-architectural, model-specific registers (MSRs) that indicate the amount of energy consumed by different parts of the system (e.g., package, cores, DRAM). We sample all the energy counters every 50ms over the course of each program's run and then combine the values to compute total energy. Dividing this value by the total runtime produces the average power during a program's execution.

	Benchmark Suite				
	Parsec	*SpecCPU*	*Splash2X*	*DaCapo*	*SpecJBB*
System (Sec. 6.4)					
Processor Frequency Tuning	✓	✓	✓	✓	✓
Overclocking (Turbo Boost)	✓	✓	✓	✓	✓
Processor Sleep States	✓	✓	✓	✓	✓
Parallelism	✓		✓		✓
Compiler Opt. Sets	✓	✓	✓		
Interpreted v. Compiled				✓	✓
Application Specific (Sec. 6.5)					
Source Code Tuning	*	*	*	*	
Per App. Frequencies	✓	✓	✓	✓	✓
Per App. Power Caps	✓	✓	✓	✓	✓

✓= full set of applications, * = select applications only

Table 6.2: **A summary of the energy efficiency techniques explored in this experimental survey.**

Benchmark Applications and Inputs Our experiments use 41 benchmarks from five different suites, each commonly used in previous energy management research. The applications represent a breadth of languages, design paradigms, and application domains. Table 6.1 lists the applications. The first ten come from the Parsec Benchmark Suite [21] which contains multi-threaded programs written in C and C++. We ran each of these pro-

grams with the 'simlarge' inputs. The next six applications are from the Splash-2 Benchmark Suite [257], which are also multi-threaded and written in C. In contrast to Parsec's benchmarks, many of Splash's benchmarks come from high-performance computing and graphics. As prior characterizations demonstrate, these two suites are also fundamentally different with respect to their memory usage and communication patterns [22]. We use the Splash2x variant of the suite that is distributed with the latest version of Parsec in order to have access to the 'simlarge' input sets. The next benchmark suite is SPEC CPU2006 [82], which includes single-threaded, CPU-intensive workloads in C, C++, and Fortran, of which we use 17 benchmarks and the test input sizes. The fourth suite is DaCapo [25], a multi-threaded Java benchmark collection with applications from a variety of real-world domains. We benchmark seven programs using the 'default' input size. Although DaCapo is multi-threaded, it does not allow the user to set the target thread count, so we leave the 'external' thread count setting at one (see the usage documentation [47]) and exclude DaCapo from the parallel experiments. The final benchmark used is SPECjbb2005 [230], which is a client/server system designed to test the performance of Java servers. As packaged, SPECjbb always tries to complete in a fixed amount of time. This makes it hard to compare energy across trials, so we use the pjbb2005 patch [26], a variant of the benchmark that fixes the workload size instead of the runtime. The workload size in pjbb is set via two inputs, a transaction and a warehouse count (see [230] for details). Exploratory experiments on our machine showed the most scalable configuration to be 100,000 transactions and 8 warehouses, so these are the settings we chose. Without constraints, pjbb uses all the hardware threads. To adjust parallelism to a discrete thread count, we used the `taskset` unix command.

Energy Management Technique Selection The energy management techniques cited in Section 6.2 represent just a fraction of work in the area. To narrow down the large pool, this study focuses on techniques that are software-controllable, as opposed to those that require changes to the underlying architecture, circuitry, or hardware devices. Because hardware energy savings are already well studied (e.g., [56].), it made sense to cut the space this way.

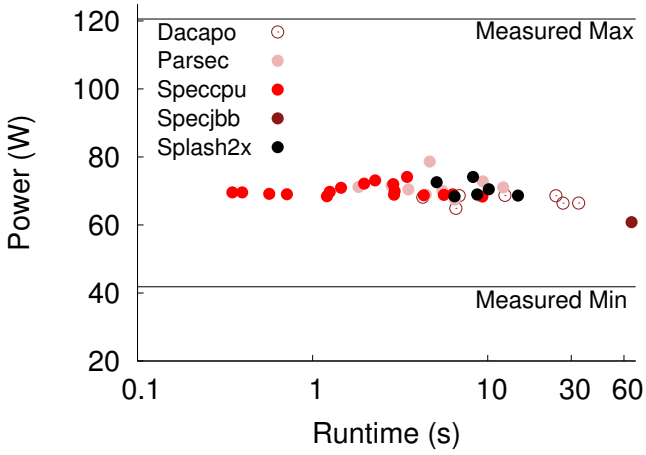


Figure 6.1: **Baseline Performance and Power.** The 41 benchmark applications exhibited more variation in runtime than in power when run at our *baseline* configuration of a single thread utilizing a processor set to maximum frequency, and with compiler/JVM optimizations and processor idle states all enabled.

We culled the remaining space by choosing a representative set of techniques that are broadly applicable to a variety of workloads and systems and that span multiple levels of the software stack. We omitted techniques that were infeasible to replicate on our own machine including those requiring complex toolchains, architectural simulation, specialized hardware, or homegrown compiler or operating systems. Table Table 6.2 summarizes the nine power management techniques we chose to study. More detailed explanations of the techniques, including pointers to relevant prior work, are presented alongside the experimental results. Section 6.4 measures the individual and combined effects of six *generic system techniques*: processor frequency scaling, overclocking, use of idle states (all in the OS), compiler optimization flags, interpretation versus compilation, and the effects of parallel thread counts. Section 6.5 presents the results of three *application-specific* experiments, namely power-oriented source code transformations, per-application processor frequency tuning, and per-application power capping. Section 6.5 also compares and contrasts application-specific strategies with generic system strategies.

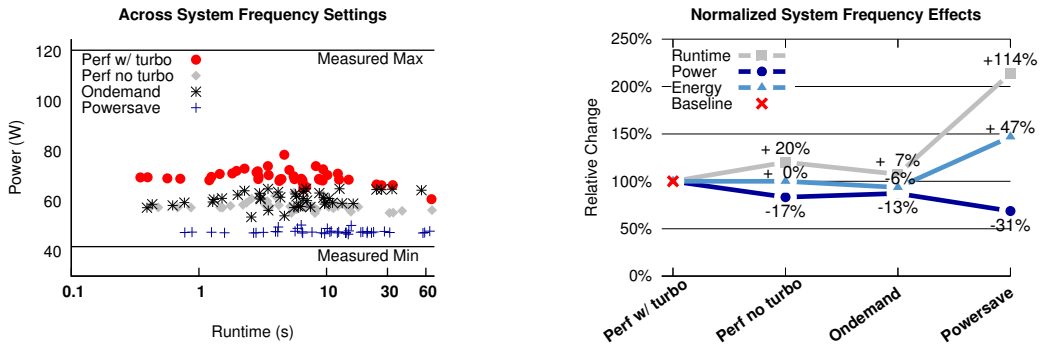


Figure 6.2: **System frequency tuning algorithms**, such as *ondemand* save at most 6% of energy across applications versus the system baseline of maximum frequency with Turbo Boost enabled (*perf w/ turbo*). Other frequency tuning options include disabling Turbo Boost for decreased runtime but no net energy savings (*perf no turbo*) or a powersave option that saves an average of 31% of the power, but with great costs to runtime (*powersave*).

Experimental Rigor Given the breadth of this study, we took particular care to gather accurate, precise, and well organized results. This strengthens our own conclusions and enables other investigators to analyze and build on our data, which we have provided at: `www.arcade.cs.columbia.edu/energy-study`. Automated scripts managed all aspects of the experiment setup, data collection and labelling, thus ensuring repeatability. In addition to the raw energy and runtime data, we gathered supplemental data, such as frequency readings via the i7z tool [103], to confirm that each configuration was successfully applied and implemented as expected. Each benchmark was run a minimum of 20 times at each configuration, and as many times as necessary for the 95% confidence interval to come within 2% of each application's energy, runtime, and power means. In rare cases, this required over 100 program runs of an application for a single configuration. In total, the measurements represent over 200,000 application runs across the 220 individual and combined energy management configurations. Using averaging (with geometric means for any pre-normalized data [62]) and normalization we compress this vast amount of data into easy to understand results.

Baseline Power and Performance For clarity and to aid inter-study comparisons, nearly all experimental data is reported relative to a single *baseline* configuration. This

baseline, which is our system's default, maximizes processor frequency (2200 MHz), enables Turbo Boost and idle states, maximizes compiler optimizations (`-O3` and `-funroll-loops` for `gcc`, compiled for the JVM), and runs each application with one thread. Getting the Java benchmarks to run with one thread required using the `taskset` command to force the virtual machine onto a single thread. When not taskset, we observed that the Virtual Machine might use any number of hardware threads even if the application is offered only a single thread.

Figure 6.1 shows the measured runtime and power of the 41 benchmarks on this baseline configuration. Each point on the plot represents the average across as many runs as required to reach our statistical standards. The runtimes (from 0.4 to 66 seconds) showed a greater range than the power consumption (from 61 to 79 Watts). Primarily a result of the range in runtime, energy also ranged widely from 24 to 4036 Joules.

This initial data corroborates existing work from Esmaeilzadeh et al. [56], showing that power is not necessarily related to the thermal-design point, or TDP, of the CPU. While the TDP of our machine is 190 Watts across both sockets, a multithreaded microbenchmark designed to generate large amounts of busywork consumed only 120 Watts. The fact that we never near TDP even at peak system usage could be a symptom of a good cooling system, though this theory has not been tested. We have marked the busywork micromenchmark as "Measured Max" power on Figure 6.1. We also record a "Measured Min" at 43 Watts, which is is the machine power when nothing other than system utilities and our power profiler were running. Note that this background power is significant, accounting for an average of 60% of the single-threaded benchmark power and for 35% of the multithreaded busywork program.

6.4 System-Level Results

Here, we present the system-level measurements of frequency tuning, overclocking, processor idle states, parallelism, and compiler flags. We first examine the impact of each setting in isolation and then examine how the five techniques combine. Section 6.5 presents the remaining application-specific techniques listed in Table 6.2.

6.4.1 Frequency Tuning and Overclocking

A huge body of prior work uses dynamic frequency scaling to improve energy efficiency. The key insight is that lower processor frequencies consume less power, so energy can be conserved if processor frequencies are reduced during periods of low work. The challenge of frequency tuning is to figure out when and by how much to reduce frequency without causing performance losses significant enough to negate the power savings. Operating systems are often tasked with this, because they can measure application performance and then reactively set the clock frequency via software exposed registers in the CPU [28]. Linux provides several algorithms, called *cpufreq governors*, to manage this process. The available algorithms depend on the machine architecture and version of Linux, so we measure three commonly available ones:

- The *performance* governor sets frequency to its maximum, 2200 MHz on our test machine. We call this setting **perf w/ Turbo** because, as described below, it also includes Turbo Boosting. It is the baseline described in Section 6.3.
- The **powersave** governor also uses a constant frequency, but at the system minimum, which is 1200 MHz on our machine.
- The **ondemand** governor increases or decreases frequency, reportedly per processor, when a (tunable) threshold of dynamically measured CPU utilization is reached [183]. We leave all tunables at their default settings, for example leaving the utilization threshold at 95%.

In addition to frequency, power governors also have limited influence on *overclocking*, which means temporarily raising frequency above the processor manufacturers' recommend level for sustained computation. Both Intel and AMD offer dynamic overclocking called Turbo Boost and Turbo CORE respectively. Overclocking may or may not have significant bearing on energy; while it reduces compute time, it also causes the system to run hotter and dissipate more power. For safety reasons, hardware has ultimate control over when and for how long overclocking can occur, but the operating system does have the option to disable overclocking all together. By default, the ondemand and performance governors

permit overclocking, which kicks in only when the processor frequency has reached the maximum rating. Using an Intel-supported driver, we were able to create a fourth governor that isolates the effects of Turbo Boosting:

- The **performance no Turbo** governor sets the CPU frequency to its maximum, but disables Turbo Boost (i.e., no dynamic over clocking).

On our system, disabling overclocking for the ondemand algorithm is not an option. Disabling overclocking for the powersave algorithm would not make sense because by the algorithm's definition, frequency is always set to minimum.

Experiments show that these four frequency management strategies yield a range of power-performance tradeoffs. The left panel of Figure 6.2 plots the individual application runtimes and power consumption at each of these four settings, while the right panel summarizes the impact of these settings across all applications.

Disabling Turbo Boost and removing the machine's ability to ramp up frequency for short periods of time resulted in a runtime *increase* of 20% across applications. We did not monitor the frequency changes across all of our experiments, but observed using the i7z tool [103] that Turbo Boost almost always increases frequency (up to 2700 MHz, or 500 MHz above the normal maximum frequency) when a single procesor is working at 100% utilization but the remaining processes are idle, as was the case for most of the experiments in Figure 6.2. In many cases, the frequency was allowed to remain at 2600-2700 Mhz for the duration of the application's execution provided the other cores remained idle, which explains the significant performance differential.

Conversely, disabling Turbo Boost *decreased* power by an average of 17% across applications, a direct consequence of the lower average processing frequency. The nearly equivalent increase in runtime and decrease in power meant that across applications, disabling Turbo Boost produced no net change in energy versus Turbo Boost enabled. Individual applications saw some minor energy shifts with Turbo Boost disabled versus enabled: at most a 9% increase and a 6% decrease with 17 applications increasing in energy consumption and 24 decreasing.

Similarly, the out-of-the-box ondemand algorithm affects energy by only a small amount, with 6% average savings across applications. Most of the individual applications (38 out of

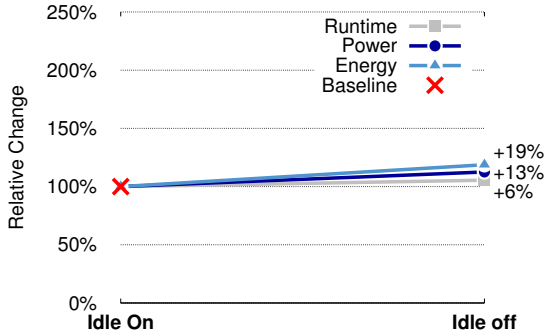


Figure 6.3: **Processor idle states** enable 19% energy savings relative to the mode that prevents cores from entering these power-saving sleep modes.

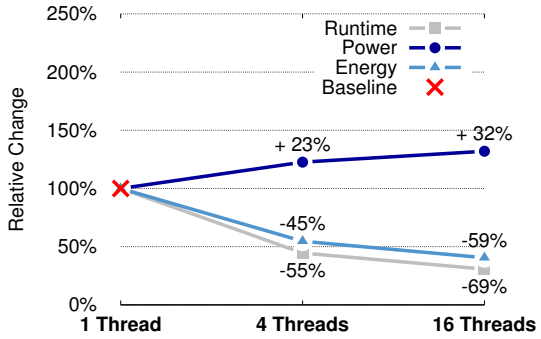


Figure 6.4: **Parallelization** increases energy savings for all applications tested. For our 12 core, 24 hyperthread server, running 16 application threads consumed just 45% of the energy of the serial execution.

41) saved a little energy, but only six saved more than 10% relative to the baseline. This limited savings may be unsurprising to some in the operating systems community, who have questioned the efficacy of the ondemand frequency tuning algorithm [136] as well as frequency tuning's potential to save energy at all on modern processors [140]. Powersave is a big energy loser, with an average increase of 47% versus the baseline and with not a single individual application saving energy. From these results, it is clear that powersave, or any similar strategy that reduces frequencies to a minimum, is not a desirable policy for active processors.

6.4.2 Idle States

Most computers spend a significant amount of time underutilized, for example while serving I/O. Datacenter servers reportedly use only 10-50% of their processors at a time; the remainder are idling [16]. Idleness can be costly in terms of power draw with under-utilized servers still drawing more than 50% of their peak power [16]. In recent years processor vendors have offered a rich menu of processor idle states, that send the processor to increasingly deep levels of 'sleep' for increasing power savings. The specifics vary from vendor to vendor, but as an example, a first level of sleep might be to stop the CPU clocks, a second to turn down CPU voltage, and a third to reduce the voltage further and stop refreshing cache [242]. The reason for multiple levels of idleness, sometimes called *c-states*, is that each deepening state comes at an added transition cost, taking increasingly more time for the processor to switch back to active. If a processor is sent into a deep idle state immediately before an application requests its resources, the application will experience runtime delays. Thus, the main challenge to managing idle states is to figure out when to idle, how deeply to idle, and when to wake up.

As with frequency scaling, the operating system has been tasked with observing application behavior and managing idle states accordingly. Linux provides a *cpuidle* idle state manager [182], which is analogous to its *cpufreq* frequency algorithms. The cpuidle manager monitors the dynamic use of all the system processors and uses this information to determine the appropriate depth of sleep. It is also possible to force a processor to use a specific idle state (see instructions in [80]) rather than allowing the automated manager to control sleep depth. According to the documentation [80], manual settings are helpful for reducing system latency but not likely to save more power than the *cpuidle* manager, so we limit our experiments to the managed algorithm rather than manual settings. This narrows our idle state exploration to just two settings:

- The **idle on** data is measured with the *perf w/ turbo* frequency tuning, per application thread count of one, and `gcc-O3` or the default `javacc` options (i.e., the baseline).
- The **idle off** is the same configuration but with cpuidle disabled (i.e., the cores are not allowed to sleep).

Figure 6.3 plots the comparison, which reveals a 19% energy difference between idle on and off across all 41 applications. For individual applications, the differences range from 11 to 25%, with all applications seeing a net energy decrease when idle states are enabled. Also in line with expectations, power is on average 13% higher when idle states are turned off, at most 24%, and at least 8%. Unexpectedly, all of the applications run faster by an average of 6% when idle states are enabled. We found that this runtime difference reverses when Turbo Boost is disabled, and we suspect that with idle states enabled, the core used by the single-threaded application is able to take advantage of the lower overall system power and turn on Turbo Boost more frequently than when idle states are disabled, resulting in the shorter runtimes.

6.4.3 Parallelism

Although the idea of parallelism has been around since the first computers [252], multicores became mainstream roughly a decade ago, when AMD and Intel started selling dual core processors for desktops. This revolution was driven largely by energy and power concerns. The increasing clock speeds and transistor counts that drove performance higher for fifty years also drove power density to unsustainable levels. Computer architects reacted by simplifying processor cores and offering more of them, which kept heat levels under control while allowing performance to continue to grow. The catch is that to effectively increase performance, software must actually *use multiple cores*.

As core counts grow, software engineers are left to deal with the difficult challenges of writing well parallelized code to improve performance on future generations of chips [153]. One could argue (as Urz Holzle, senior vice president at Google, did [88]) that it is the responsibility of computer architects to keep serial processing efficient so that software is not forced into parallelism. However, for better or worse, chip-multiprocessors now dominate the desktop market, and core counts in the mobile market are also creeping up [222]. In addition to runtime efficiency, prior research has shown that energy efficiency is similarly reliant on effective parallelization [196], so energy conscious programmers must also deal with this reality.

We measured the interplay of performance, energy, and parallelism on our 12 core, 24

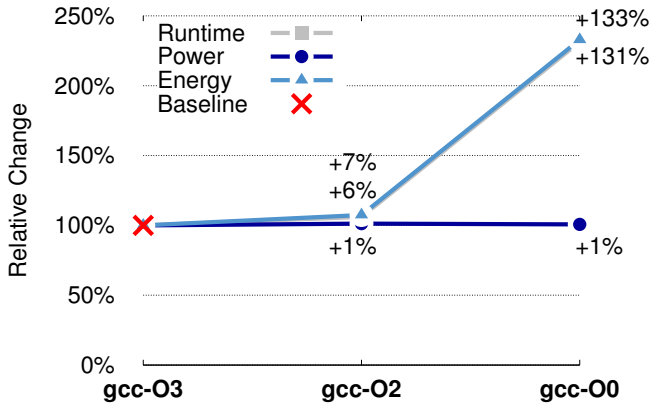


Figure 6.5: **Standard compiler optimization sets** save energy, but largely through runtime reductions not power reductions. Applications without optimization take 133% more energy and 131% more time than fully optimized applications.

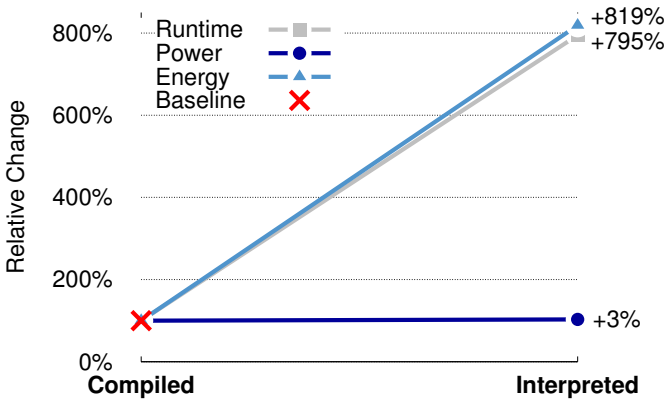


Figure 6.6: **Java compilation** saves substantial energy versus interpreted code, which consumes 8X the energy, but again these savings are due to runtime, not power.

hyperthread machine; Figure 6.4 summarizes the results. The baseline is the same as before, with the data showing average changes in runtime, power, and energy for 4-thread and 16-thread program runs versus single-threaded runs. The results are averaged across multiple runs of the 17 benchmarks from the Parsec, Splash2x, and SPECjbb suites, the three suites that supported discrete thread count settings. From the runtime values, it is evident that some of the applications scale poorly: on average the applications show only a 2X speedup over serial with four threads, and a 3X times speedup with 16 threads. The most scalable application tested, `radix`, saw only an 8X speedup at 16 threads. Jumping from 4 to 16 threads caused `radix`'s power to increase by 50%, thanks to increased core activity reducing the opportunity to exploit idle states. This power increase tempered the runtime savings, so that `radix`'s energy at 16 threads was about 20% of its single-threaded energy. In other applications, a similar phenomenon occurred: speedups provided by added parallelism were offset by the power increases resulting from more concurrently active threads. However the power increases *did not* exceed the runtime savings for any of the applications we tested, meaning all of the applications saved energy. Even the most poorly scaling application, `raytrace`, whose runtime at 16 threads decreased only 19% versus one thread saved a non-negligible amount of energy at 13%. On average, the applications saved 55% of the single-threaded energy when run with 16 threads.

6.4.4 Compiler optimizations

Most existing work on energy efficient compilation focuses on the power and energy impacts of performance optimizations. They typically find that these optimizations reduce runtime much more than they increase power, resulting in a net decrease in energy. In attempts to isolate which optimizations are the most power efficient, a number of studies apply individual optimizations such as function in-lining, loop unrolling, and loop vectorization to benchmarks (e.g., [211]). Other research has constructed new optimization sets for energy rather than performance (e.g., [184]). The conclusion of all prior studies seems to be the same: when it comes to compilation, what is best for performance is best for energy.

This is not a surprising conclusion when the optimizations tested affect performance almost exclusively (and not power). The community has proposed a few power-optimizations,

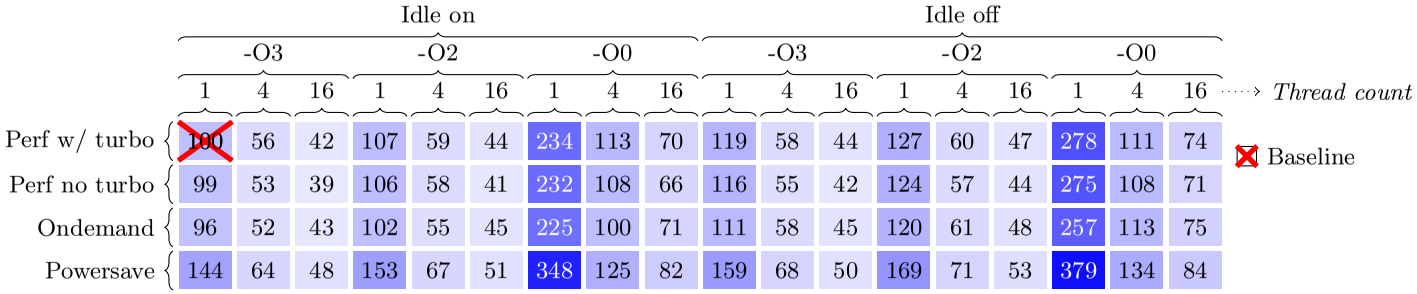


Figure 6.7: **Energy effects of combining multiple configurations.** This table shows cross-level energy interactions of the five energy configurations discussed so far, as a percentage of the baseline, (i.e., baseline = 100%). Note that the matrix includes data from only the parallel, native benchmark suites: Parsec, Splash2x.

such as reordering instructions or memory operands or reassigning registers to reduce control path switching. A good overview of these techniques is presented in a 1994 paper by Tiwari et al. [239]. Twenty years later it seems none of these techniques have made it into mainstream compilers, so quantitative data on their ability to improve power and energy is sparse. Given the lack of power-specific optimizations in commercial compilers, we measure the energy effects of standard sets of compiler optimizations. While we are far from the first to take these measurements, we include them to provide a quantitative comparison point for the other measurements in this chapter.

Figure 6.5 shows the energy effects of the standard `gcc` compiler optimization sets for applications in the three native benchmark suites. On average, the applications ran in 131% less time for `gcc-O3` versus `gcc-O0`, meaning the optimized code took 43% of the time of the unoptimized code to run. The change in power was negligible, about 1% on average, so the energy effects track the runtime, with `gcc-O0` taking 233% of `gcc-O3`'s energy. The per application energy savings of turning on optimizations ranged wildly, from less than 1% to nearly 700%, likely a reflection of how optimized the original source code was. In terms of energy and runtime the `-O2` optimizations were very similar to `-O3` on average, with 8 of the 33 applications actually saving more energy with `-O2` than with `-O3`. These numbers emphasize compilers' important contributions to energy savings, but confirm that all the savings come in the form of reduced runtime.

For the eight Java benchmarks, we measured the energy of interpreting rather than compiling. On average, the cost of interpreting was huge, consuming 818% more energy on average than compilation, which is roughly in line with the runtime impact of 795%. Again, the energy changes varied between applications, from an energy savings of 23% for `pmd` (the only application to save energy, and purely a result of runtime savings), to an increase of over 2600% for `sunflow`. Average power increases were barely significant at just 3%, and varied less between applications (-0.5% to 6%).

6.4.5 Cross-Layer Energy Effects

The preceding sections explore the impact of each optimization in isolation. We wanted to know whether turning on multiple techniques resulted in additive, negative, or synergistic

interference, so we ran experiments that combine all of the techniques presented so far. The heatmap matrix in Figure 6.7 shows all of these combinatorial effects as a percentage of the baseline. The data in the matrix comes from the 16 applications in the two parallelizable, native benchmark suites, Splash2x and Parsec. The rows represent different system frequency algorithms, while the columns cover all of the idle states, compiler options, and parallelism configurations previously discussed.

A number of insights could be drawn from these comparative experiments. Most notably, the energy savings of one strategy can be cut by half depending on what other strategies are in use (e.g., enabling idle states saves 19% at the baseline frequency, but only 10.4% when the powersave algorithm is used.) Similarly, the ondemand frequency algorithm saves less energy at 16 threads than with one thread. In fact, at 16 threads, ondemand actually increases energy regardless of compiler optimization or idle state configuration. Compiler optimizations follow this pattern as well, saving less energy at 16 threads (about 40% across configurations) than at one thread (57%).

Several techniques were a win across the board. For all 18 configurations, disabling Turbo Boost saved energy because the runtime savings from Turbo Boost's increased frequency were more than offset by corresponding power increases. However, unlike the other techniques, disabling Turbo Boost saves *more* energy for 16 threaded trials than serial trials. Idle states also provided nearly universal energy savings, with 34 out of 36 configurations showing energy decreases when they were enabled. Increasing parallelism, even without perfect performance scaling, was also a relatively large energy winner, with energy decreasing from 1 to 4 to 16 threads for all configurations.

Ultimately, the best energy configuration was with idle states on, the `-O3` optimization set, 16 threads, and the performance no turbo frequency tuning. Note that this does not match our baseline and the system default, which enables Turbo Boost. The worst configuration was essentially the opposite: `-O0`, idle off, one thread, and the powersave algorithm. The difference between these two configurations is a whopping 10.3X.

6.5 Application-Level Energy Management

This section presents the measurements of three application-specific energy management techniques: source code tuning, custom frequency scaling, and power capping. It then links these techniques to system-level techniques in a combination study.

6.5.1 Source Code Tuning

A number of recent and older works suggest that optimizing source code for power savings can have significant impact [20, 30, 162, 239]. Surveying these works and others, we found eight kinds of source-level transformations purported to save power or energy, and applied these transformations to the eight benchmarks marked with a * in Table 6.1. The transformations aim to:

1. reduce temporary variables,
2. eliminate common subexpressions (e.g. consolidate duplicate computations or lookups in complex structures),
3. postpone variable declarations until needed,
4. use operator= instead of the operator alone and use prefix instead of postfix operators (but only for complex types),
5. use direct assignments of variables rather than initializations followed by assignment,
6. replace multiply and divide operations with shifts or addition when possible,
7. optimize loops with unrolling and unswitching (moving a conditional from inside to outside of a loop)
8. reduce the number of arguments passed to functions.

To focus our efforts, we looked for opportunities to apply the first six optimizations within loops or within functions called inside loops. In total, we made 688 changes across the eight applications, ranging from 5 to 292 changes per application. Figure 6.8 shows the exact number of changes per application, in square brackets above each triplet of bars. It was simpler to make changes in the less optimized applications of the Splash2x and Parsec

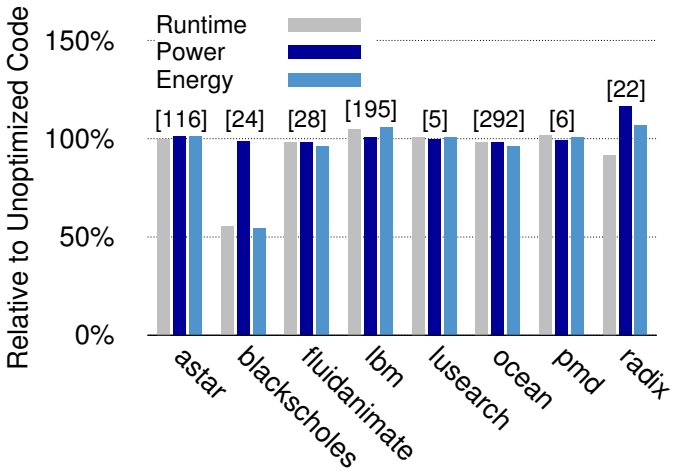


Figure 6.8: **Source code tuning** methods from prior embedded systems research were not very effective energy savers for for our complex and already well-optimized benchmarks running on servers.

benchmarks, and conversely more difficult to improve the already-optimized SPECCPU benchmarks. The DaCapo benchmarks were especially challenging to transform. This is partly because they are already well optimized, and partly because it is difficult to track the scope of objects whose instantiation may be far removed from use, as opposed to variables in native benchmarks, whose scope often lasts only one function. When the scope of an object was unclear, it was necessary to be more conservative about deletion or modification.

Figure 6.8 shows the power, performance, and energy effects of our transformations relative to the unoptimized programs. The data represents multiple trials, all utilizing the same baseline as previous experiments, with `gcc` optimization level `O3` and the `-funroll-loops` options enabled for native programs, and the compiled virtual machine used for the Java programs. Only one application saw a significant reduction in energy — `blackscholes` — while four others' energy was slightly reduced by our transformations. The effective transformations in `blackscholes` were common subexpression elimination, the reduction of temporary variables, and direct assignment, all within a 'hot' function, and missed by the optimizing compiler due to the complex objects involved in the computation. The transformations reduced power for five of the eight applications, but none of these measured reductions were outside of our 2% confidence interval range, so they should be considered statistically insignificant. One application, `radix`, experienced a significant power increase

at 17%, likely due to additional loop unrolls that the compiler would normally not perform.

This study could be considered a failure given that the optimizations did not result in significant power or energy savings, but we still felt it important to include the negative results. They demonstrate that, at least for previously optimized applications run on servers, micro-optimizations for power and energy are challenging. Given that the results were poor and the source-level transformations require disproportionately more effort than other energy saving techniques, we suggest that *power-specific source transformations are not worth the average programmer's time once the code has been optimized for performance.*

6.5.2 Application Tuned Frequencies

As previously shown, the ondemand frequency governor provides only small energy savings. In part this is because it is conservative (optimizing for performance) and reactive (waiting to measure processor utilization before adjusting frequency). We also observed (using the i7z frequency monitoring tool [103]) that even when only one core is utilized by an application, ondemand tends to unnecessarily ramp up the frequency of the entire socket. All of these behaviors limit ondemand's ability to conserve energy.

A number of researchers have noticed that reactive measurements coupled with the high latencies of switching frequencies through the OS may result in less than optimal frequency tuning, and have proposed alternate methods. For example, a recent paper by Rangan et al. [199] proposes setting individual core frequencies to different static values, then migrating application threads to improve both energy and throughput. Hints from compilers [258], static analysis tools [219], and even software developers [229] have also been proposed to tune frequencies more effectively. None of these papers distribute open-source code, so in lieu of reimplementing their work, we contextualize it by running individual applications at discrete, constant frequency levels. This obviously does not replicate techniques that continually switch applications between frequency levels, but it at least gives us an idea of the range of power-performance tradeoffs involved in application-specific frequency tuning.

Linux provides a mechanism for root to change individual processor frequencies through a *userspace* governor. On the machine used for these experiments, frequencies can be set to 11 distinct levels, from 1200 MHz through 2200 MHz at 100 MHz steps. A twelfth option

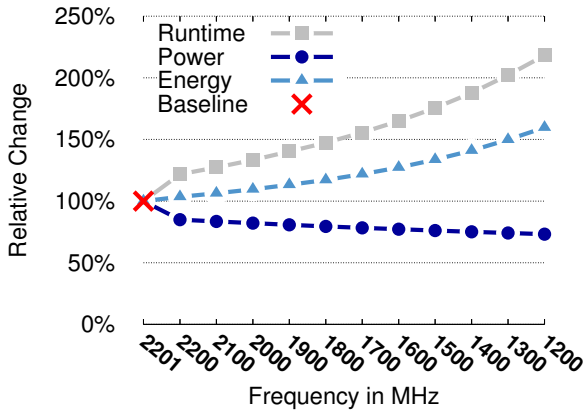


Figure 6.9: **Application-specific frequency tuning**, or running an application at a single discrete frequency, allows power-performance tradeoffs to be flexibly manipulated.

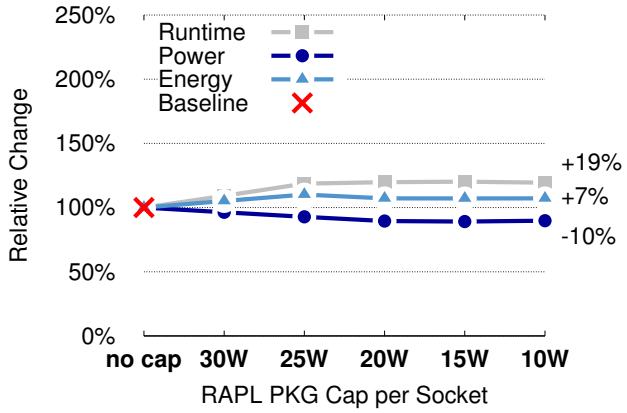


Figure 6.10: **RAPL power caps**, which limit the amount of power a part of the chip is allowed to consume over a given time window, yield a more limited power-performance tradeoff range.

is to set the frequency to maximum (2200 MHz) and enable Turbo Boost. Figure 6.9 shows how all 41 benchmarks perform, on average, at different frequency levels. So that all the applications could be used, these results show single-threaded runs: the unused 11 cores (22 hyperthreads) were set to minimum frequency while the occupied processor's frequency was varied. Idle states are turned on for all of these experiments. In the average case, none of the frequency configurations saves energy relative to 2200 MHz with Turbo Boost; instead, there is a smooth power-performance tradeoff curve with runtime increases always slightly over-shadowing power decreases. Looking at individual applications, three of the 41 save a negligible amount of energy when Turbo Boost is disabled but frequency remains set to 2200 MHz. Moving down the frequency scale to 2100 MHz none of the applications save any significant amounts of energy. Power decreases are relatively uniform across applications, sinking a little more at each frequency. The corresponding runtime increases, however, vary significantly between applications. For example, at 1700 MHz, runtime may increase as little as 38% or as much as 74% relative to the baseline, resulting in relative energy losses of 9 to 35%. Future algorithms should be sure to account for this highly application-specific response to frequency tuning.

6.5.3 Per Application Power Caps

While hardware ensures that on-chip power levels do not exceed the TDP, sometimes there is a need to cap power at a lower level. For example, in datacenters, enforcing a strict power cap somewhere below the TDP could make energy expenses more predictable and affordable. Several research projects have addressed this need via power-attentive thread to core scheduling and DVFS [40, 99, 208]. A couple of industrial tools exist as well, for example, Intel's RAPL Power Caps [45]. Since our machine contains Intel processors, we experiment with this particular implementation.

RAPL allows a user with sufficient privileges to limit power across multiple domains per socket: *power plane 0* which includes cores and private caches, *power plane 1* which includes alternate processing units such as GPUs, the *package* which includes both power planes as well as shared caches, and finally *DRAM*. The user selects a domain to cap, then gives the RAPL interface a specific power value to limit that domain, as well as a time

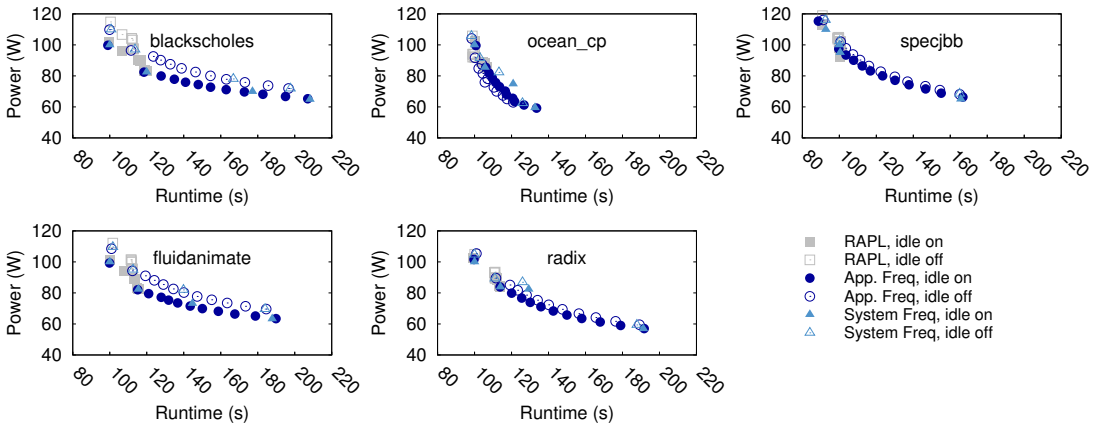


Figure 6.11: **Application-specific strategies versus system level strategies for frequency tuning.** RAPL caps, application-specific frequency tuning, and system frequency governors could not be combined with each other, so we compared their power performance effects instead. All three could be combined with idle states, however, which when enabled saved energy across all of the different frequency configurations.

window. The time window specifies periods during which average power levels must meet the cap. For example, if the given window is 100ms and the cap is 30W, RAPL promises that every 100ms, the average power of the specified domain will not exceed 30W. RAPL documentation is unclear about how these power limits are maintained, but our reverse engineering shows that frequency scaling is at least part of their strategy. RAPL capping overrides system frequency algorithms, but does allow idle states to be enabled.

Figure 6.10 shows the results of capping both sockets' package power at various levels. Initial experiments showed that useful package capping values fell between 30 and 10 Watts. Only the 8 applications marked with a * in Table 6.1 were used for these results. Across applications, capping traded modest (up to 11%) decreases in power for modest but slightly larger (up to 20%) increases in runtime, resulting in a slight net energy *increase*. The graph indicates that the power increases and runtime decreases were not quite monotonic as power caps were lowered, but the slight up and down fluctuations are less than our error range at 1%, and thus should not be considered statistically significant.

6.5.4 Comparing Application-Specific to System-Level Strategies

We wrap up our study by showing how system-level strategies (frequency governors and idle states) compare with application-level techniques (application-specific frequency tuning and RAPL caps). The frequency governors, application-specific frequencies, and RAPL caps all control the same knob of processor frequency, so these three strategies cannot be combined and should be considered mutually exclusive. However, idle states can be combined with all three forms of frequency tuning, and we noticed that toggling Turbo Boost on or off has some effect on RAPL capping performance. For the five parallelizable applications marked with * in Table 6.1, we set the compiler optimization level to `-O3`, and the thread counts to 16 (note this deviation from the baseline of one thread; we return to unnormalized data here) then ran multiple trials of the following configurations:

- 6 levels of RAPL caps $\times$ 2 idle settings $\times$ 2 turbo settings
- 12 application-specific frequency values $\times$ 2 idle settings
- 4 system frequency algorithms $\times$ 2 idle settings

This comes to a total of 56 configurations per application, each of which is plotted in the power-performance graphs of Figure 6.11. A few insights immediately jump out. First, the fastest configurations tend to take the most power, and overall, the majority of configurations seem to make a strict trade of increased runtime for decreases in power. Second, `ocean_cp` trades off much less performance for power savings than the other three applications. Third, regardless of the configuration, turning idle states on has a positive effect on power. Though it may not jump out of the plots immediately, idle states also save energy for the majority of configurations.

Relative to the system default (i.e., the baseline), the number of configurations that save energy varies significantly per application. For example, 50 of `ocean_cp`'s 56 configurations save energy, with the best saving 24%. However, for `fluidanimate`, just 12 configurations save energy with the best saving only 5%. The lowest energy configuration for each application varies as well: for `ocean_cp` the best is to turn off idle states and minimize frequency, while for `fluidanimate` and `blackscholes` it is idle states on and maximized frequency, and for both `radix` and `SPECjbb` it is idle states on with RAPL caps set to 10W and

no Turbo Boost. These results show that while the power savings trends of each strategy may hold across applications, the performance tradeoffs can vary, resulting in unpredictable energy effects.

6.6 Related Work

In the most comprehensive prior study of energy efficiency techniques, Esmaeilzadeh et al. [56] examined the power-performance tradeoffs of different microarchitectural features including clock frequencies, memory hierarchy configuration, and hardware parallelism. While the two studies overlap in some dimensions (parallelism, frequency tuning), ours explores a wider variety of software techniques, for the first time allowing direct quantitative comparisons between energy efficiency solutions at different layers of the stack.

Several other studies also compare multiple hardware-level energy efficiency techniques. Patki et al. take an HPC perspective, examining how overprovisioning techniques (such as overclocking) and power capping can help improve supercomputers' efficiency [190]. Subramaniam and Feng combine RAPL capping with a variety of server load inputs to see how well RAPL can provide energy proportionality (i.e., similarly efficient execution for different levels of server utilization) [233]. Le Sueur and Heiser examine the effects of DVFS and idle states across multiple processor generations [140], finding that newer processors see smaller energy benefits from frequency scaling. None of these works compare the hardware-level techniques to higher level software techniques as we do.

Like ours, Schone et al.'s experiment space includes processor level DVFS, different degrees of parallelism, and overclocking [210], but their study measures the impact of these techniques on memory and last level cache bandwidth only. The relative energy savings of multiple software-level techniques are compared in just a few prior works. Most are either qualitative (e.g., [59]), or focused exclusively on compiler or application-level energy management strategies [104, 171]), without linking those techniques to the system-level as this study has.

6.7 Limitations and Future Work

The experimental survey of software–level energy management solutions presented here is the most thorough published work in the area, but there are still opportunities for plenty of future work.

Compare more energy solutions. Beyond the nine explored energy management techniques, there are many additional energy management solutions discussed in literature or available as open source tools. Future work could continue to compare and combine other energy strategies to those presented in this work.

Investigate cross–language energy differences. We did not examine the energy effects of the same algorithm being implemented in different languages. Such a future investigation would need to be nuanced to pick apart the energy losses or gains involved in not only the implementation of the language itself, but also of the types of choices developers make when writing an algorithm in different languages — for example through the use of iterative versus recursive solutions or procedural versus object-oriented program structures.

Examine effects of different benchmark inputs. While we did study a breadth of benchmarks in this work, we did not examine the effects of different benchmark inputs. Different inputs could have a significant effect on the energy use of an application, and it would be worth investigating whether this effect impacts the efficacy of different kinds of energy saving strategies.

6.8 Discussion

Energy management has become a large field in recent years, with work spanning all levels of the stack. The broad interest in energy-efficiency has caused fragmentation: most management strategies are not compared against each other – especially those at different levels of the stack – and most research papers do not quantitatively or even qualitatively address how their work will combine with existing strategies. As a first step to bridging these discontinuities, this experimental survey directly compared and combined nine existing but

previously uncontrasted energy management strategies.

Chapter 7

NRG-Loops

For the final project of this dissertation, we demonstrate how feedback–directed optimizations within software can allow applications to have a more active role in reducing inefficiencies. Specifically, we introduce a new language extension called *NRG-Loops*,[1] which allows an application to manage its own power and energy consumption through dynamic adjustments to functionality, performance, and accuracy. The adjustments, which come in the form of truncated, adapted, or perforated loops, are conditionally enabled as runtime power and energy constraints dictate. NRG-Loops are portable across different hardware platforms and operating systems and are complementary to existing system-level efficiency techniques, such as DVFS and idle states. Using a prototype C library supported by commodity hardware energy meters (and with no modifications to the compiler or operating system), this chapter demonstrates four NRG-Loop applications that in 2-6 lines of source code changes can save up to 55% power and 90% energy, resulting in up to 12X better energy efficiency than system-level techniques.

7.1 Introduction

Computer scientists were concerned about power consumption when the Eniac was built in 1946 [158], and since then, concerns have only increased as power overconsumption

[1]This work was previously introduced in a conference publication [117] and earlier versions of this work appeared as a tech report [114] and as a workshop paper [116].

threatens to have significant financial and environmental consequences. Although power and energy efficiency have traditionally been considered an issue for only hardware and operating systems to handle, energy efficiency concerns are slowly creeping up the stack and becoming a problem for application software to address as well. In 2011, 70% of returned Motorola devices were due to battery life complaints attributable to applications [69], and new power monitors — such as the OS X Battery Status Menu — allow end-users to see which applications are to blame when power consumption is high.

While power efficiency is important, it does not yet trump runtime efficiency. Unfortunately, runtime performance and power efficiency are often at odds, and balancing the two needs simultaneously is a challenge. Until recently, a choice had to be made at hardware design time between optimizing for higher performance or for lower power. Now, instead of preselecting the tradeoffs, computer architects build in the option to dynamically tune hardware resources at runtime, so that hardware may switch between performance-aggressive and power-saving states, or to various states in the middle of those two goals. The different balances of power and performance can be adjusted through a wide menu of power management controls that include changing processor or memory voltage and frequency (DVFS), temporarily overclocking or putting processors into idle states, and choosing from one of multiple asymmetric multicores.

These *system-level knobs* are controlled by operating systems, runtime software, and compilers. A key downside of system-level knobs is that they must be tuned conservatively to avoid disturbing the performance and accuracy of overlying programs. In typical uses, even mildly power-saving states are entered only when the hardware is completely unused by overlying applications resulting in limited power and energy savings for active systems. Another issue with system-level power techniques is that they must be applied to hardware constructs — for example, voltage must be scaled for an entire socket or at least an entire processor core — which is not suitable when multiple applications share a hardware context through multiprogramming or simultaneous multithreading.

To achieve more aggressive energy savings, programs must introduce their own *application-level knobs*. Tough decisions about when it is appropriate to trade performance, accuracy, or functionality for power and energy savings cannot be made by compilers or operating

systems. They need to be internal to applications, and made on a case-by-case basis by developers, as different applications may want to make syntactically and semantically varied changes. For example, one application might choose to save power by adjusting its caching strategies, while another might reduce thread counts, and another may choose to scale down data structure sizes. However, to make power and energy efficiency trades, programmers need two types of support not presently available. First, they need runtime power and energy usage statistics that are accessible to the program's source code, in order to evaluate when adaptations are needed, or even whether they are needed at all for a particular program execution. Second, they need language support that helps them incorporate adaptations into source code simply but flexibly, without making assumptions about the underlying platform so that applications remain portable.

This chapter provides for both of these needs through a set of language extensions called *NRG-Loops* (pronounced "Energy Loops"). NRG-Loops let applications to set runtime evaluated *NRG-Conditions* that dictate whether and when to make changes. NRG-Conditions are used within annotated `for` loops, and ensure that applications make adjustments only in the case that runtime power or energy use meets a specified budget. This budget can be expressed in absolute Watts or Joules, or alternatively set relative to other parts of the program (e.g., function `foo` is allocated 50% of the energy of function `bar`), or relative to the system (e.g. 80% of the maximum system power). At runtime, the budgets are compared to the accumulated energy or average power across loop iterations using measurements abstracted from system hardware counters. When the budget is met, the program dynamically *truncates*, *adapts*, or *perforates* the loop to begin reducing power or energy use. NRG-Loop adjustments can be made in arbitrary application-specific ways, the resulting code is portable to multiple systems, the savings are complementary to system-level energy management techniques, and no compiler or operating system modifications are required.

The primary contributions of this chapter are:

- A specification of NRG-Loops, a platform-independent C or C++ language extension that lets applications trade performance, accuracy, or functionality only as dynamic power and energy use necessitates (Section 7.2).

- A prototype implementation of NRG-Loops, called *NRG-RAPL*, that utilizes hardware power counters to implement the NRG-Loop syntax and semantics for a Linux/Intel platform (Section 7.3).
- Four case studies that demonstrate 10-90% energy savings and up to 55% power savings by changing just 2-6 lines of source code per program (Section 7.4).

NRG-Conditions

```
NRG_TOT_E <= <float> // in Joules
```

```
NRG_AVG_P <= <float> // in Watts
```

NRG-Loops

```
NRG_TRUNC_for (<loop bounds> &&
               <NRG condition>) {
  // body
}
```

```
NRG_ADAPT_for (<loop bounds> &&
            <NRG condition>) {
  // original body
} NRG_ALTERNATE {
  // alternate body
}
```

```
NRG_PROB_PERF_for (<loop bounds> &&
                   <NRG condition>;
                   PROB_SKIP=<float>) {
  // body
}
```

```
NRG_AUTO_PERF_for (<restricted bound> &&
                   NRG_TOT_E <= <float>) {
  // body
}
```

NRG-Helpers

```
SYS_MAX_POWER
```

```
SYS_MIN_POWER
```

```
struct NRG_USAGE_INFO {
    float energy;
    float average_power;
    float wall_time;
}
```

```
NRG_AUDIT {
  // arbitrary code
} NRG_USAGE(NRG_USAGE_INFO* foo);
```

Figure 7.1: **The NRG-Loops Syntax.**

7.2 NRG-Loops

NRG-Loops provide a simple, flexible interface for applications to modify their own performance, functionality, and accuracy to save power and energy. The syntax of NRG-Loops is purposefully brief to avoid a steep learning curve for users. This first version of NRG-Loops extends C or C++ programs, but a similar paradigm could be developed for other languages. This section describes the syntax and semantics of NRG-Loops, which consists of several abstractions: *NRG-Conditions*, four types of `for`-loop directives, and a few helper data structures and functions. To increase portability, NRG-Loops abstract away the underlying measurements or models that collect power and energy usage statistics. Later, Section 7.3 discusses one possible implementation of NRG-Loops that uses hardware power meters to populate this information into the appropriate syntactic structures.

7.2.1 NRG-Conditions

NRG-Conditions enable programs to monitor accumulated energy or average power across loop iterations, and then to react intelligently to these measurements at runtime. There are two types of NRG-Conditions, as illustrated in Figure 7.1. The first type of condition, `NRG_TOT_E`, checks if the total accumulated energy across loop iterations is less than or equal to the given value of Joules. The right-hand side can be any expression that evaluates to a floating point value. The second type of condition, `NRG_AVG_P`, checks if the average power across loop iterations is less than or equal to the specified value in Watts, again expressed as a floating point expression. An example NRG-Condition that limits power to 50 Watts is: `NRG_AVG_P <= 50.0`.

Instead of setting NRG-Condition values in terms of absolute Watts or Joules, the power or energy limits for one piece of code can alternatively be set relative to the amount consumed by a different part of source code at runtime. For example, a user could write an NRG-Condition that ensures a loop body in function `bar()` consumes at most the energy that function `foo()` consumed: `NRG_TOT_E <= foo_energy`.

The value of `foo_energy` could be predetermined at development time, but more likely, users will want to dynamically import such a value, because energy varies across platforms,

inputs, and even different executions. Dynamic energy measurement is easy with the help of NRG_AUDITs, which enclose an arbitrary region of code in a pair of curly braces, as shown in Figure 7.1. The enclosed code may perform any computation including spawning threads and calling functions — even calling more NRG_AUDITS. The audits record the energy (in Joules), the average power (in Watts) and the wall time (in seconds) of the enclosed region (and any child threads or functions) and deposit the information into a named NRG_USAGE_INFO structure. For example, foo_energy can be obtained as follows:

```
NRG_AUDIT {
  foo();
} NRG_USAGE (NRG_USAGE_INFO* foo_usage);
float foo_energy = foo_usage->energy;
```

7.2.2 Truncate Loops

NRG-Conditions serve as a secondary for loop bound (concatenated to the original loop bound) for different types of NRG-Loops. The first type of NRG-Loop is called an NRG_TRUNCATE_for. Continuing our foo() example, an NRG_TRUNCATE_for can be expressed as follows:

```
NRG_TRUNCATE_for (int i=0; i<N; ++i &&
                  NRG_TOT_E <= foo_energy) {
  // do work until foo_energy exceeded
}
```

This directive tells the loop body to execute while both the original loop bound (int i=0; i<N; ++i) and NRG-Condition (NRG_TOT_E <= foo_energy) hold, and to stop execution otherwise. Like a regular loop bound, the NRG-Condition is checked only at the beginning of a loop iteration, and thus will not stop a loop in the middle of an iteration, even if the power limit or energy budget has already been exceeded. The user rather than the NRG_TRUNCATE_for is responsible for any clean-up (e.g. releasing a lock, freeing memory, closing files) that may be required as a result of exiting the loop early.

7.2.3 Adapt Loops

The next type of loop directive is an NRG_ADAPT_for. Like the truncate directive, it concatenates an NRG-Condition to the original loop bound. It also has the user add a second,

alternate loop body that directly follows the first, is wrapped in brackets, and is preceded by the NRG_ALTERNATE keyword as shown in Figure 7.1 and the snippet below:

```
NRG_ADAPT_for (int i=0; i<N; ++i &&
                   NRG_TOT_E <= foo_energy) {
  // do work until foo_energy exceeded
} NRG_ALTERNATE {
  // do more energy efficient work
}
```

Execution of the original loop body continues while both the original loop bound and the NRG-Condition hold. If the NRG-Condition breaks before the original bound, then execution transfers to the alternate loop body. After the transfer, the alternate body continues to execute until the original bound is met. Note that the original bound state (e.g., loop index value or i in then running example) is *not* reset upon transfer to the alternate body. As with the truncate loop, NRG-Conditions are checked only at the beginning of loop iterations and control is never transferred in the middle of an iteration. Again, any required clean up relating to loop body transfer must be handled by the user.

In the case of an NRG_AVG_P condition, loop execution may transfer back to the original loop body if the average power goes back below the specified limit after the alternate body is executed for a time. This will not happen with an NRG_TOT_E condition, because total energy increases monotonically.

7.2.4 Perforate Loops

Finally, there are two types of NRG-Loops that allow applications to perforate, or skip select loop iterations. The first type, NRG_PROB_PERF_for, executes normally until the NRG-Condition bound is exceeded, then probabilistically skips iterations with a probability of the specified PROB_SKIP, which should be a floating point number between 0 and 1. For example, if the user specifies PROB_SKIP = 0.1 as in the following example, once the NRG-Condition has been exceeded, 1 out of 10 future loop iterations will be skipped.

```
NRG_PROB_PERF_FOR (int i=0; i<N; ++i &&
                   NRG_TOT_E <= foo_energy;
                   PROB_SKIP = 0.1) {
  // once NRG-Condition met, do work 9/10 times
```

```
}
```

The second type of perforation is `NRG_AUTO_PERF_for`. This type of NRG-Loop automatically decides how many loop iterations to skip in order to meet a user-specified energy budget. Unlike the other types of NRG-Loops, it does not support `NRG_AVG_P` and it restricts the original loop bound to be of the form `(int i=0; i<=N; ++i)`. For example:

```
NRG_AUTO_PERF_FOR (int i=0; i<N; ++i &&
                   NRG_TOT_E <= foo_energy) {
  // do work, skipping an estimated number of
  // iterations to exactly match foo_energy
}
```

These two restrictions allow the loop increment to be modified (e.g., change `++i` to `i=i+2` or `i=i+3`, etc.) to keep the loop on target to consume no more energy than specified in the `NRG_TOT_E` condition.

7.2.5 NRG Helpers

NRG-Loops also contains two helpers to assist users in choosing platform-independent NRG-Condition values. The first helper, `SYS_MAX_POWER`, is a global floating point value that holds the maximum power (in Watts) that the platform can achieve with all hardware threads active. Similarly, `SYS_MIN_POWER` is a global floating point value that holds the minimum power (in Watts) of the system when running essential services only (i.e., an idle operating system). These two constants can be used within NRG-Conditions to further abstract NRG-Loop code from a particular platform, for example, `NRG_AVG_P <= 0.8*SYS_MAX_POWER`.

7.3 NRG-RAPL

The first implementation of the NRG-Loops interface is a C library called *NRG-RAPL*. It is named for its use of Intel's *Running Average Power Limit (RAPL)* [45] counters to profile energy. NRG-RAPL is portable and lightweight, utilizing commodity hardware and requiring no operating system extensions or middleware. This section describes the library implementation, including how NRG-Loops syntax is translated at the preprocessor level into pure C (Section 7.3.1), how energy is profiled (Section 7.3.2), and how we attribute

the hardware-level measurements to software constructs (Section 7.3.3). The tool usage (Section 7.3.4) is also discussed.

7.3.1 Translating NRG-Loops

Since the NRG-Loops syntax is highly abstracted to minimize programmer effort, NRG-RAPL's first job is to translate the code into a pure C intermediate representation. Translation involves (1) accumulating energy or recording average power across loop iterations as required, (2) checking this usage against the specified limits, and (3) adjusting the source code as necessary. To collect power and energy data, the intermediate representation uses the previously described `NRG_AUDIT` helper and its `NRG_USAGE_INFO` structure.

Each kind of loop directive and type of NRG-Condition has its own intermediate representation, so there are eight types of translations in total. Space prevents us from sharing all eight, but as an example, here is the intermediate representation of an Adapt NRG-Loop when an energy condition is used:

```
float NRG_BUDGET = 0.0;
for (int i = 0; i<N; ++i) {
  if (NRG_BUDGET <= <float>) {
    NRG_AUDIT {
      // original loop body
    } NRG_USAGE(NRG_USAGE_INFO *use);
    NRG_BUDGET += use->energy;
  } else {
    // alternate loop body
  }
}
```

The translated code starts the loop with its original bounds, inserting a check at the top of each iteration to see if energy use has exceeded or met the user-specified budget. If it has, control transfers to the alternate body through the use of an `else` statement. If the budget has not been met, the original loop body runs — within an `NRG_AUDIT`. The recorded `NRG_USAGE_INFO` is accessed to update the budget based on the energy consumed by the loop body.

7.3.2 Profiling Energy and Power

Any power profiler or model could be used to collect energy and power within the audit calls. There is some debate about the best way to monitor power and energy today, because techniques vary widely in terms of implementation complexity, precision, accuracy, portability/availability, and scope of coverage (i.e., is only processor power being measured, or the whole system's power draw including any peripherals such as monitors?). Unfortunately, no existing power measurement technique fares well in all of these categories. For this implementation, we chose to use a combination of hardware power meters and operating system usage statistics that performs at least reasonably in each of the categories and has been shown to be among the most accurate techniques for measuring power [204], and has been used in several previous works (e.g., [214] and [263]).

Hardware meters supplement power measurements with event-based linear models that are periodically re-calibrated. Meters are conveniently found in widely available server SoCs such as the Intel SandyBridge and the IBM POWER7, and have recently been introduced to mobile devices. The meters can cover a large portion of computation, accounting for processor, interconnect, cache, and DRAM power and energy. However, with update frequencies on the order of 1ms, they are not terribly precise and have been shown to have occasional modeling errors [160]. Other drawbacks are that the meters are small and overflow frequently, and that they currently exist only at the granularity of a whole socket, which can contain multiple CPUs running many concurrent processes, making it a challenge to attribute power measurements to individual processes and threads.

7.3.3 Energy Accounting in NRG-RAPL

NRG-RAPL's energy accounting fixes the overflow and hardware meter granularity problems, and reduces the overheads involved in dynamic profiling by combining meter reads across multiple concurrently running audit functions. When the application has one or more audits open, NRG_RAPL spawns a single monitoring thread (regardless of the number of active audits) to periodically sample:

- E_{sys}: a system-wide energy reading obtained from RAPL counters for all sockets

- U_{sys}: system-wide CPU time from `/proc/stat`
- U_{tid}: CPU time for each application thread *tid* from `/proc/<pid>/tasks/<tid>/stat`

The frequency of these readings is configurable, but RAPL samples must be taken frequently enough to provide good precision and to detect overflow — the RAPL counters overflowed roughly every 10-20 seconds on our experimental machine — yet not so often that profiling results in excessive overhead. Sampling at 100 Hz strikes a good balance between these constraints on our machine, yielding reasonable precision with negligible time or power overhead above the unmonitored application.

As every i^{th} sample is recorded, NRG-RAPL decomposes it into individual measurements for every active thread *tid* according to the following equation:

$$E_{tid,i} = \frac{U_{tid,i} - U_{tid,i-1}}{U_{sys,i} - U_{sys,i-1}} \times (E_{sys,i} - E_{sys,i-1})$$

This divides the measured energy values amongst running threads according to CPU utilization. Thus, should another thread or co-running application run up E_{sys}, that usage will not be charged to *tid*.

In addition to the profiling samples, NRG-RAPL maintains an application thread tree by interposing on calls to pthreads which create or destroy threads. It also maintains information about the nested structure of the audits created per thread by reading the opening and closing brackets surrounding audits in the intermediate translations of NRG-Loops. Combining the active thread and audit information with $E_{tid,_i}$ measurements, NRG-RAPL fills in usage records of any open audits of the thread *tid* and its ancestors.

7.3.4 Usage Logistics

After a user adds NRG-Loop directives to their application, he or she must link against NRG-RAPL (i.e., `-lnrgrapl`), and ensure that the NRG-RAPL shared object file is loaded first (i.e., via `LD_PRELOAD or LD_LIBRARY_PATH`). The only other requirement of the current implementation is that, once compiled, the application be executed by a sudoer. This is because Linux does not currently expose even read-only access of the RAPL registers to

non-sudoers. This work is just one example of why it would be beneficial for Linux to do so in the future.

7.4 Case Studies

This section demonstrates the potential of NRG-Loops and the immediate utility of NRG-RAPL with four case studies that: maintain an energy budget by trading-off video quality (Section 7.4.1), respect a power cap by reducing parallelism (Section 7.4.2), save power by approximating a mathematical algorithm (Section 7.4.3), and keep a third-party advertisement's power use in check (Section 7.4.4). The NRG-RAPL instrumentation adds minimal energy, power, and runtime overhead to these applications (Section 7.4.5).

The experimental platform is a dual socket Dell PowerEdge R420 server with Intel Sandybridge E5-2430 processors, each with 6 cores, 12 hardware threads, and 24GB of DRAM (for a machine total of 12 cores, 24 threads, and 48GB DRAM). The machine runs Linux kernel version 3.9.11 and Ubuntu 12.04.2. Intel sleep states [182], Turbo Boost [35], and the ondemand frequency governor [183] are turned *on* for all the experiments to demonstrate that NRG-Loops complement and extend the savings of existing system-level energy management techniques.

7.4.1 Perforate: Bodytrack

Sometimes the best strategy to meet high computational demands under strict energy budgets is to reduce the accuracy of application services provided. To demonstrate real-world, application-level accuracy tradeoffs, we augmented the `bodytrack` application from the Parsec benchmark suite [21]. `Bodytrack` tracks the poses of a person recorded on multiple video cameras; the majority of this work occurs in a for loop within the main function, which processes frames one at a time. We perforated this loop using both types of NRG perforation loops. For the first type of perforation, we varied the probabilistic percentage of loops a user might choose to skip from 0 to 75% by modifying the `MY_PROB` variable in the code below. Note that the NRG-Condition is somewhat of a no-op in this example, telling the code to start perforating when energy use is greater than or equal to zero; we did this

to make the numbers comparable to our next experiment.

```
NRG_PROB_PERF_for (int i=0; i < frames; ++i &&
  NRG_TOT_E <=0; PROB_SKIP = MY_PROB) {
    // DO FRAME PROCESSING
}
```

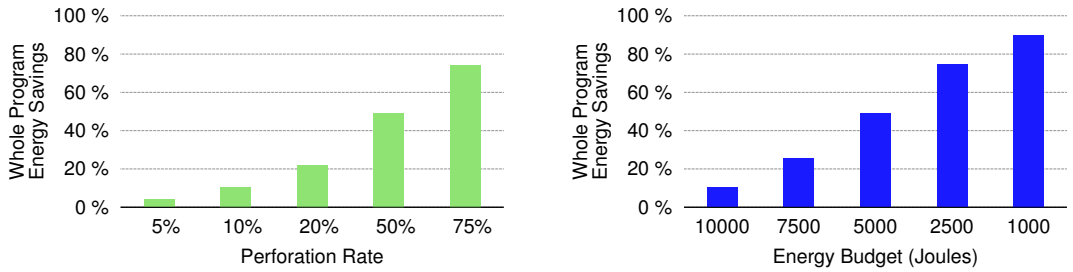


Figure 7.2: **An NRG Perforate Loop** augments `bodytrack` to (left) drop different specified percentages of frames to save energy, or (right) maximize quality without exceeding various allocated energy budgets.

Varying the probability of perforation (to 0.05, 0.1, 0.2, 0.5, and 0.75) produces the whole program energy savings shown to the left of Figure 7.2 when `bodytrack` is run with an input size of native and the `-O3` compiler flag is enabled. Unsurprisingly, as more frames are skipped more energy is saved, ranging from 3.9% savings when 5% of frames are dropped to 74% savings when 75% of frames are unexecuted. Of course, in most applications skipping 75% of the work is not reasonable. Developers will have to decide how many frames it makes sense to trade for energy, but at least with NRG-Loops they can now reason about the energy values of these tradeoffs and easily affect the changes they want to make at runtime, incorporating dynamic conditions such as changing inputs into their decision.

The second way to initiate perforation changes in a program using NRG-Loops is to set a total energy budget. This may be preferable in situations where the developer knows exactly how much energy they need to save, and is flexible about the number of frames skipped. Adding this kind of NRG-Loop to a program is as easy as choosing the `BUDGET` to spend, connecting the library, and modifying a single line of source code:

```
NRG_AUTO_PERF_for (int i=0; i < frames; ++i &&
  NRG_TOT_E <= BUDGET) {
    // DO FRAME PROCESSING
```

```
}
```

To find appropriate possible `BUDGET`s, we first used an `NRG_AUDIT` helper to determine the overall energy consumed by the program, which is approximately 9600 Joules on our machine. The right side of Figure 7.2 shows the percentage of energy saved when the `BUDGET` is set to 10,000, 7500, 5000, 2500, and 1000 Joules, and the framerate is adjusted automatically by the NRG Perforate Loop to meet these budgets. Even though 10,000 is above the un-budgeted, original program's energy, there is still a little savings because the NRG-Loop drops a few early frames before it is sure that the target budget will be met. For lower budgets, even more frames are dropped throughout the loop's iterations, and the energy savings are accordingly higher — up to 89.8% when the budget is set to 1000 Joules.

7.4.2 Adapt: Parallel Substring Search

For safety reasons and to prevent overheating, hardware enforces a *hard power cap*. Called a TDP, or thermal design point, this upper bound for power varies by architecture, and in practice may never be reached thanks to efficient cooling strategies. For example, on our experimental machine, which has a 190W TDP, we never observed a peak power of more than 120W even with all 12 cores fully utilized. Despite this, there are numerous reasons one may wish to cap an application's power below the TDP, for example to make datacenter energy expenses more predictable and affordable, to allow more headroom under the TDP for applications sharing a machine, or to throttle usage when the power supply is intermittent or variable as with RFID harvesting [49], solar [229], or kinetic [105] sources of energy. We call these sub-TDP power caps *soft power caps*.

Several existing tools tune hardware and operating system resources to enforce user-specified soft caps, using techniques such as DVFS, thread mapping, and asymmetric hardware [40, 99, 208]. The soft caps provided by these tools are hardware-centric, and thus applied to specific hardware components such as a single core, a socket, or a whole machine. In contrast, NRG-Loops enable soft caps on software entities, specifically those encased by `for` loops.

Software-centric power caps can be simply implemented using the `NRG_ADAPT_for` directive. To compare system-based soft caps to NRG Adapt Loop-based soft caps, we imple-

mented a C++ benchmark that searches many long base strings for a particular substring match. Substring searching is used in many important real world applications, such as genomic analysis and satellite image processing. For faster throughput, base strings can be searched in parallel by multiple software threads. And, since parallelism is highly correlated with power [115], we opted to use dynamic adjustments to the degree of parallelism to regulate the application's power and enforce the soft cap.

The code that follows shows how NRG-Loop annotations can be added to the benchmark to enforce a user-specified cap at `SOFT_CAP` Watts.

```
NRG_ADAPT_for (int i=0; i<STRINGS_TO_CHECK; ++i &&
               NRG_AVG_P <= SOFT_CAP) {
  if (num_threads < MAX) num_threads += 2;
  // num_threads search concurrently for substring
} NRG_ALTERNATE {
  num_threads -= 2;
  if (num_threads < MIN) num_threads = MIN;
  // num_threads search concurrently for substring
}
```

As the benchmark searches base strings with `num_threads` threads, an `NRG_ADAPT_for` checks that average power stays below the `SOFT_CAP`. If it does not, control shifts to the alternate loop body, where the thread count is decreased by 2, and a check for over decrementing is performed to ensure that `num_threads` is at least a `MIN` number of threads (in our experiments, 2). Afterwards, the smaller `num_threads` search for the substring just as in the original loop body. Finally, in case the average loop power happens to go back above the `SOFT_CAP`, execution will automatically return to the original loop body, so we also added a line of code to the original body that increments the threads back up by 2 to increase search speed.

In the code snippet, `SOFT_CAP` is expressed as an absolute value, but it could also be expressed relative to the system maximum using the helper described in Section 7.2.5, for example: `#define SOFT_CAP 0.5*SYS_MAX_POWER`.

We compare the NRG-Loop solution against Intel's RAPL and DVFS-based power capping tool, Intel Power Governor [46], which is an example of system-only soft caps. The tool lets users limit power across three domains per socket: *power plane 0 (PP0)* which

includes cores and private caches, the *package (PKG)* which includes all PP0 elements as well as alternate processing units such as GPUs and shared caches, and *DRAM*. The user selects one or more of these three domains to cap, then gives the Power Governor interface a specific power value in Watts to limit each domain to, as well as a time window in milliseconds over which the running average power must not exceed the cap. As an example, if the time window is 100ms and the cap is 30W, RAPL promises that for every 100ms, the average power of the target RAPL domain will not exceed 30W. Because RAPL caps are enforced in multiple domains, getting our benchmark to respect an overall soft cap required tuning all the individual settings. For example, to get the program to run within a soft cap of 55 Watts, the RAPL PKG cap had to be set to 45 Watts, and the other two domains had to remain uncapped. To get the program to run within a soft cap of 40 Watts, we had to set the PKG cap to 15 Watts and the DRAM to 15 Watts and disable Turbo Boost. Such device and system-specific tuning is probably more than most software developers will wish to take on, particularly as the settings must be re-tuned for each soft cap, application, device, and, potentially, input.

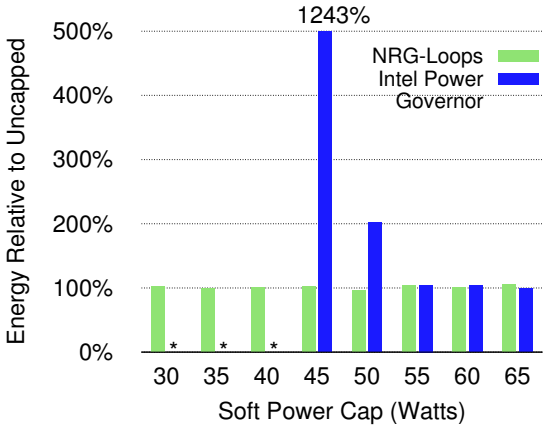


Figure 7.3: **NRG Adapt Loops** can meet a preset power budget by adjusting application-internal thread count, analogously to the Intel Power Governor tuning DVFS. For the string matching application shown, NRG-Loops can set a broader range of caps (Power Governor caps could not be used below 45 Watts), and required up to 12X less energy to enforce them.

Figure 7.3 shows the energy impact on the substring search benchmark when NRG-Loops and the Power Governor enforce a range of soft caps between 30 and 65 Watts. In

each experiment, the maximum thread count is set to 12, the number of base strings to 12 thousand, and the length of each base string to 5 million. Each bar reports the program energy at a particular cap relative to the energy of the uncapped program, which maximizes performance by running continuously with 12 threads. For the NRG-Loop soft caps, the reduced power was always almost exactly offset by the increase in runtime due to decreased parallel processing. Thus, the total energy was roughly equivalent to the peak performance energy, regardless of what cap was set.

The same was not true for the Power Governor caps. First, no Power Governor cap setting could produce a soft cap below 45 Watts. Even at 45 Watts, the Power Governor struggled to maintain the soft cap, significantly increasing runtime so that the energy consumption was more than 12 times both the uncapped program and the NRG Loop-based cap. For the 50 Watt Power Governor soft cap, the energy consumed was still 2X the uncapped energy. Only at the 60 Watt cap, as the program neared its uncapped power of 63 Watts, were Power Governor caps finally able to keep energy within 4% of uncapped.

At least for this application, NRG-Loops worked better than system capping in two ways. First, once we added the 7 lines of code above to the program, setting a new cap was as simple as passing a new value for `SOFT_CAP`); far preferable to the complicated tuning required to set the Power Governor caps. Second, the NRG-Loop based caps offered a broader range of viable power caps than Power Governor, often at a significantly lower energy consumption.

7.4.3 Truncate: Streamcluster

Algorithms sometimes spend valuable energy converging to a perfect solution when an approximate solution is good enough. The `streamcluster` application from the Parsec benchmark suite [21] is one example of this. Streamcluster is a data mining/pattern recognition application that solves the online clustering problem, assigning a stream of input points to their nearest *center* [181]. Misailovic et al. identified a loop within the `pFL` function that can be approximated: if given fewer iterations, the number of centers that the program considers clustering the data around will be decreased, possibly without detriment to a final solution [164].

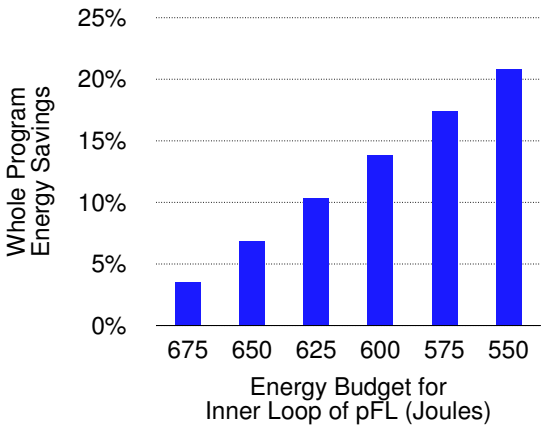


Figure 7.4: **NRG Truncate Loops** estimate a mathematical clustering algorithm within `streamcluster` to save various amounts of whole program energy depending on the degree of approximation.

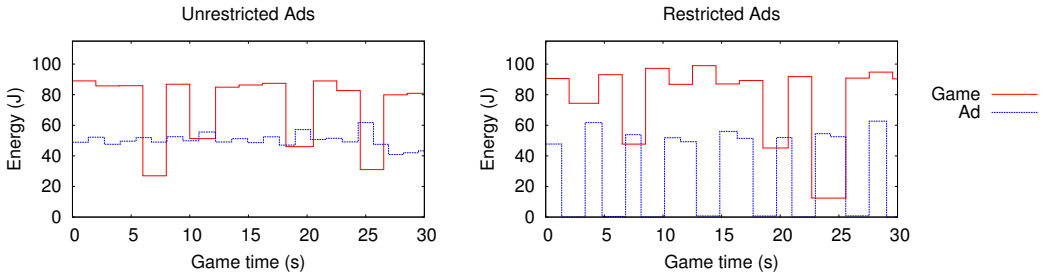


Figure 7.5: **A minesweeper game uses NRG-Adapt Loops to prioritize game power over third-party advertisements.** Run unchecked, the ads sometimes consumes more power than the game, but NRG-Loops can force the ads to occasionally pause, decreasing net game+ad energy.

In the Misailovic et al. work, the authors tried to guarantee minimal disturbance to an ideal solution, so they conservatively perforated the loop. In a scenario where overall energy savings is more important that guaranteeing a near-perfect outcome of the algorithm, a truncating NRG-Loop may be appropriate. Truncating the inner-loop of `pFL` with NRG-Loops is as simple as including the NRG-RAPL header file and modifying the original `for` loop to include an extra NRG-Condition:

```
float pFL (<args>) {
  NRG_TRUNCATE_for (i=0; i<iter; ++i &&
    NRG_TOT_E <= BUDGET) {
    // COMPUTE CLUSTER VALUE
```

```
    }
  }
```

To determine the potential whole program energy savings of truncating the loop, we experimented with different values assigned to the `BUDGET` variable in the NRG-Condition. To find a baseline for our experiments, we first used an `NRG_AUDIT` and discovered that, when run with native inputs, the inner loop consumes anywhere from 600 to 733 Joules to complete all of its iterations. With a `BUDGET` value of 675 Joules (which is in the range of un-truncated consumption and therefore most likely affects program accuracy minimally), NRG-Loops provides a whole program energy savings of 3.5%, as shown in Figure 7.4. Reducing the energy of the inner loop further — down to 550 Joules — results in a whole program energy savings of over 20%.

7.4.4 Adapt: Minesweeper and Advertisement

Free applications comprise 91% of the mobile marketplace [76], and 77% of the top free applications in the Google Play store are advertisement supported [142]. Moreover, third-party ads may consume as much as 65-75% of the total mobile application energy [185]. These numbers indicate that developers and mobile providers alike have an incentive to ensure that mobile ads consume only their fair share of energy.

While operating systems could be tasked with moderating ad energy use, that would take valuable control away from developers. Instead, NRG-Loops make it simple for developers to moderate ads on their own terms using familiar programming techniques even if the advertisements are written by third-parties and have unpredictable demands. To demonstrate this, we introduced NRG-Loops into a text-based minesweeper game [152] that calls a simulated advertisement. The advertisement in this experiment is a separate pthread that performs computationally and I/O intensive busywork. The minesweeper game has potentially long rounds of play with varying durations depending on live user interactions. To keep the advertisement's power in check, we converted the code that calls the advertisement into an Adapt NRG-Loop that pauses the ad for `PAUSE_TIME` microseconds if it consumes more than a given `POWER_LIMIT`:

```
NRG_ADAPT_for (int i=0; i<MAX_ADS; ++i &&
```

```
                NRG_AVG_P <= POWER_LIMIT) {
  // run ad normally
} NRG_ALTERNATE {
  usleep(PAUSE_TIME);
}
```

Figure 7.5 shows resource measurements of two versions of the game being played by a real user. The first version (at left) does not restrict ad energy, while the second version (at right) restricts advertisements with the `PAUSE_TIME` set 2 seconds and the `POWER_LIMIT` to 20 Watts. Both graphs show the energy in Joules consumed by the minesweeper game in two second intervals over 30 total seconds of play, and the energy of the advertisement recorded at each iteration of the above `for` loop. Fluctuations in the game energy (solid red series) are a result of dynamic user interactions. In the unrestricted ads version, the advertisement energy (dashed blue series) regularly consumes more than half of the energy of the game, and sometimes even exceeds the game's energy. In the restricted version of the game, the ads are periodically paused to respect the power limit. Together the unrestricted game and ads consume an average of 72 Watts, while the restricted game and ads consume only an average 54 Watts together.

7.4.5 Overheads

To find the overheads of NRG-RAPL, we compared the applications' runtime, power, and energy with no NRG-Loops annotations versus with annotations added but dynamic modifications disabled (i.e. the frame skip probabilities were set to zero, and the energy budgets and power caps were set to unattainable levels). We then used a standalone RAPL monitor thread to measure 10 trials of the programs running on the opposite chip socket so that the power and energy consumption of the monitor itself would not be counted. Across trials and benchmarks, the maximum energy increase was 0.6%, the maximum power increase was 0.1%, and the maximum runtime increase was 1.4% (on average, -0.4%), indicating that NRG-RAPL has a very limited affect on program performance. These overheads do not include the minesweeper game, because the live user interaction makes the energy usage and runtime highly variable.

7.5 Related Work

Energy management is an established and crowded field. This section contextualizes NRG-Loops within this large body of work, relating it to examples of different categories of efficiency techniques: *system-only management* techniques, those that use *application hints* or both *application hints and exposed knobs*.

System-Only Management. Most energy managers operate entirely at the system level, with the system both offering the energy conserving *knobs* and initiating the action to tune them. Modern systems offer many knobs, including DVFS, overclocking, idle states, adjustable DRAM refresh rates, asymmetric multicores, configurable floating point widths, and dynamic adjustments to LED screens [52]. These knobs let the system avoid using and paying for any more resources than necessary to maintain performance and accuracy, but must be conservatively tuned to avoid performance or accuracy hits to overlying applications. Projects that fall into this category including Linux's `cpufreq` and `cpuidle` governors [28, 182], PowerAdvisor [260], Pack & Cap [40], computational sprinting [196], and a tool for optimizing dynamic backlight scaling [146].

Application Hints. Another category of energy management tools adds application hints to help manage system resources. These tools use annotations or new languages to denote regions of code for which it is safe to optimize power while potentially reducing performance or accuracy. To make the adjustments or approximations, they rely on the same set of system-level knobs as tools in the previous category, though those knobs may now be tuned more aggressively. For example, EnerJ [207] is a language where the type system indicates which program values can tolerate imprecision for subsequent approximation by the runtime system. Some of the other techniques in this category include architecture support for disciplined approximate programming [57], Flikker [147], Eon [229], and the Latency, Accuracy, Battery abstraction [123].

Application Hints & Exposed Knobs. A third category of work lets the application expose internal knobs (most commonly, loop perforation), but ultimately relies on the sys-

tem to decide when and by how much to tune those knobs. Examples include PowerScope used with the Odyssey OS [63], GRACE-2 [247], PowerDial's Dynamic Knobs [87], the Green framework [11].

Application-Only Management. NRG-Loops comprise a new category of management solutions, that allow applications to tune their own knobs from within themselves. There are several important benefits to forgoing system involvement. First, application-only management provides transparency and control to the programmer, which beats hoping for a system's "best effort" of efficiency. Additionally, application-only management does not require modifying the operating system, meaning that updating new application knobs is simpler. Finally, application-only management is portable — the source code is not tied to specialized systems, and thus program annotations do not need to be revised to execute on new platforms.

7.6 Limitations and Future Work

There are a few limitations to the NRG-Loop work that could be addressed in future work; we discuss them in the following paragraphs.

Language and Platform. NRG-Loops are currently only supported within C and C++, and on machines that can dynamically measure power usage. However, with extra implementation effort, the NRG-Loops paradigm could be extended to work with other popular languages, such as Java and Python. Additionally, while today's mobile phones do not typically provide power measurements accessible to user-space, we believe that this will be a common feature in the near future.

Measurement precision. RAPL registers are updated every 1ms and the CPU usage every 10ms (at 100 jiffies/sec), so NRG-RAPL cannot audit sub-10ms windows of execution. However, since the application must adjust loops that represent large chunks of execution to make a difference in energy consumption, this may not be a practical limitation. In the case studies presented in the next section we had no problems with precision.

System coverage. The specifics of the power model remain, by design, orthogonal to the NRG-Loop interface, with the expectation that as power models improve and energy meters become more pervasive, the availability and quality of power information provided through the NRG-Loop interface will only improve. For example, in smartphones, many non-processor resources such as the network and backlight account for a significant portion of smartphone power draw. To the extent that their energy usage is exposed, it can be incorporated into improved implementations of the NRG-Loop interface.

Library Preemption. With a user-level library like NRG-RAPL, it is possible that the monitor thread could be preempted or delayed, thus creating irregular profiling samples. However, Linux never stalled the monitor in any of our experiments, including stress tests with more than 100 busy application threads. On extremely busy systems, it may be necessary for monitor threads to be granted a higher priority so that they are not preempted.

7.7 Discussion

In most previous work, the operating system or specialized hardware took sole responsibility for managing power and energy efficiency. In contrast, NRG-Loops enable *applications* to have a more active role in reducing efficiency, resulting in more control for programmers. Additionally, this chapter showed that application-only management is not only complementary to system-level management, but that is also portable, simple, and effective — saving up to 55% of whole system power and up to 90% of system energy with just a few lines of source code modifications. All of the benefits of NRG-Loops are immediately available through an open source, software-only C library (NRG-RAPL) which runs on commodity hardware and does not require changes to the operating system or a new runtime system.

Chapter 8

Conclusions

As computing systems become more diverse, it is increasingly difficult to match arbitrary software to general purpose hardware in an optimally performant and energy efficient manner, even with the help of sophisticated compilers and operating systems. This dissertation explained why this mismatch is a significant problem, demonstrated some cases where the mismatch caused losses in efficiency, and presented five case studies that use new measurement techniques and methodologies to improve the efficiency of contemporary computer systems.

8.1 Summary of Findings

The five ideas presented in Chapters 3-7 support this dissertation's thesis that measurement can mitigate inefficiencies. The projects used different types of new and existing measurement technologies to understand efficiency behaviors at the intersection of hardware and software, and to direct users towards the construction of more efficient general-purpose systems. Below, we summarize the findings of each project.

Parallel Block Vectors Chapter 3 explored efficiency issues that arise as a result of parallel computing. To do so, it introduced a new way of examining parallel program performance, called PBV profiles. Unlike existing profiles which examine parallel programs from the perspective of a thread or process, PBV profiles count runtime statistics per basic

block and per parallelism phase. The fast collection of PBV profiles (enabled by the new Harmony tool), coupled with detailed dynamic information about program behavior, makes parallel block vectors broadly useful. This chapter demonstrated a few ways in which PBV profiles can help identify inefficiencies in past program executions— pinpointing the very small regions of code where they occur — and also discussed and demonstrated methods for optimizing both hardware and software to reduce inefficiencies in future program executions.

Datacenter-Wide Application Interference Chapter 4 moved beyond single machine parallelism to discuss inefficiencies that arise as a result of CMP and SMT parallelism in the context of a distributed datacenter. The chapter discussed the challenges involved in predicting, measuring, and correcting datacenter–wide efficiency issues, with a particular focus on *application interference*. The chapter went on to suggest a collection of measurement techniques to work around the identified complexities and to work towards understanding interference between datacenter applications. A proof-of-concept implementation and an application interference study on production Google servers revealed application interference "in the wild" on 1000 12-core machines running live commercial datacenter workloads. In addition to demonstrating the feasibility of measurement and the presence of real world application interference, this chapter outlined a couple of procedures to reduce this specific kind of inefficiency.

Speedy GPGPU Design Chapter 5 showed how inefficiencies can be avoided at hardware design time with hardware–software co-design. Specifically, this chapter took three steps towards speeding up the design of GPUs for computational workloads. First, it introduced a new, fast GPU profiling tool called GT-Pin, which measures a variety of instruction-level performance factors of applications as they run natively on existing GPUs, helping to identify a variety of inefficiencies. Next, the chapter showed a characterization by GT-Pin of 25 very large OpenCL benchmarks, exploring several features relevant to GPU design. Finally, it introduced a method to expedite cycle-accurate performance simulations of very large, real–world computational applications on general purpose graphics processing units, thus making it simpler to avoid efficiency issues at the hardware design stage.

Energy Efficiency Across the Stack Chapter 6 shifted focus to power– and energy–inefficiencies. The project in this chapter involved understanding and evaluating a growing body of software-level solutions for reducing energy–inefficiencies, and reporting the results of an experimental survey that compared and combined 220 combinations of such solutions. The work prompted a number of suggestions and directions for future energy research, particularly for software-controlled energy management.

NRG-Loops Chapter 7 introduced a method for correcting energy–inefficiencies "on the fly" within a programming language, without compromising the portability of applications to different platforms. The new NRG-Loop language extensions enable this by allowing applications to conditionally adapt their own performance, accuracy, or functionality, when runtime measurements indicate that their execution has exceeded preset power limits or energy budgets. This chapter demonstrated that, among other adaptations, applications can cancel unnecessary work, estimate mathematical solutions, adjust framerates, or decrease internal parallelism to save power and energy on general purpose systems.

8.2 Looking Forward

The research conducted for this dissertation has uncovered a few takeaway lessons for the community, as well as some research areas in need of future attention.

Power consumption needs to be a focus across the system stack. Currently, most consider power aand energy consumption to be a problem that should be solved exclusively at the level of hardware and the circuitry. However, it is important for all systems researchers from programming languages and compilers researchers to system researchers and computer architects to care about power and energy consumption for several reasons including environmental sustainability, financial expenses of computing, and device battery life.

A couple of ideas for improving power and energy efficiency emerged as we conducted the energy survey in Chapter 6. For example, development aids that help programmers write energy efficient code could overcome multiple issues including the difficulty of manual

power optimization, the special needs of object-oriented programs [20], and the inflation in energy caused by IDE programming [30]. There is also a need for power aware compiler optimizations, since present optimizations have minimal effect on power. Machine-specific power optimizations may be especially timely, with mobile devices such as Android moving to ahead-of-time byte code compilation [144].

Adequate measurement and tuning support must be provided. One of the chief reasons researchers today are not focusing on power and energy consumption is a dearth of good measurement and modelling tools. For example, smartphones typically do not include meters for monitoring memory versus processor versus peripherals such as LCD screens, if they include any power meters at all. Even when in–hardware support is provided, sometimes it is not as simple to use as it should be. For example in the NRG-Loop project presented in Chapter 7, the hardware power counters were extremely limiting. First, they were small, and frequently overflowed; and second, they were available only per socket, and not per core, which would have made our attribution of power to processors much simpler.

Parallelism needs to stop being overlooked. Parallelism is largely related to power consumption, as we discussed in Chapters 3 and 6. It is also extremely ubiquitous — all five of our projects dealt with some form of parallelism. Despite this, many research papers seem to ignore the effects of parallelism at both the hardware level (CMP, SMT, and ILP), and the software level (multithreaded programming at the application level, multi-programming at the operating system level). Pretending parallelism does not exist and will not persist into the future is a dangerous presumption that is likely to lead to incorrect or at least inapplicable research results.

Work should be quantitatively compared to commonly available baselines. As the number of researchers greatly increases the existing corpus of research and the volume per year of research papers will grow as well. Thus the research community needs to take concerted steps to reduce the amount of fragmentation in related work. Ideally, all quantitative research ideas should be quantitatively compared to similar works, but this is becoming increasingly less feasible as the amount of related work increases, and as systems

work becomes increasingly complex to implement. One solution is to have quantitative research papers compare their work to commonly available baselines. For example, OS DVFS strategies might compare their energy savings to the widely accessible Linux frequency scaling governors. An alternative is for more experimental surveys — such as the one in Chapter 6 — to be conducted by independent researchers.

Processing and memory heterogeneity will be a major challenge. Heterogeneity of processing units is finally more of a reality than a theory, and memory heterogeneity is likely to increase in the future with the new NVRAM, stacked, and NDP technologies being studied. A lot of work, particularly the area of measurement and performance analysis, will be needed to figure out how to keep inefficiencies to a minimum even within the highly heterogeneous architectures that are the future of general purpose systems.

Specialization will not negate the importance of general purpose efficiency. Computer hardware that is specialized to minimize inefficiencies via software co-design shows incredible promise in reducing performance and energy costs by multiple orders of magnitude. However, one lesson that is underscored by a few of the projects in this dissertation as well as related work is that general purpose computing and computer system parts (including dated hardware, programming languages, and application source code) are not going away any time soon. As such, they need to remain a major focus of computer systems researchers. In Chapter 3, we discussed Amdahl's law and its implication that the serial code regions impart bounds on program speedups. In the future, there may be an Amdahl's law corollary for the bound that *non-specialized* code regions and hardware impart on specialized systems, and we suspect that this will put significant focus back on general purpose systems research.

As computing technology progresses, complexity and diversity will continue to abound at all levels of the system stack. While coordinated hardware-software co-design will undoubtedly be critical in reaching peak computational capabilities, we must be careful not to

forget about keeping general purpose computing efficient. Just like those presented in this dissertation, measurement studies that focus on understanding behaviors at the intersection of hardware and software will continue to be an important facet of keeping inefficiencies in check.

Bibliography

[1] perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/`, July 2011.

[2] Alaa R. Alameldeen and David A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4), July 2006.

[3] Thomas E. Anderson and Edward D. Lazowska. Quartz: a tool for tuning parallel program performance. *SIGMETRICS*, 18:115–125, 1990.

[4] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[5] Padma Apparao, Ravi Iyer, and Don Newell. Towards modeling & analysis of consolidated CMP servers. *ACM SIGARCH Computer Architecture News*, 36:38–45, May 2008.

[6] Luca Ardito, Giuseppe Procaccianti, Marco Torchiano, and Giuseppe Migliore. Profiling power consumption on mobile devices. In *ENERGY: The International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, 2013.

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 2010.

[8] JosL. Ayala, Alexander Veidenbaum, and Marisa Lpez-Vallejo. Power-aware compilation for register file energy reduction. *International Journal of Parallel Programming*, 31(6), 2003.

[9] Reza Azimi, David K. Tam, Livio Soares, and Michael Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *ACM SIGOPS Operating Systems Review*, 43:56–65, April 2009.

[10] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, Chi-Keung Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with Pin. *Computer*, 43(3):34 –41, 2010.

[11] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 198–209, June 2010.

[12] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, 2009.

[13] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4):36–42, 2014.

[14] Nilanjan Banerjee, Ahmad Rahmati, Mark D. Corner, Sami Rollins, and Lin Zhong. Ubiquitous Computing. *Lecture Notes in Computer Science*, 4717, 2007.

[15] Mohammad Banikazemi, Dan Poff, and Bulent Abali. PAM: A novel performance/power aware meta-scheduler for multi-core systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2008.

[16] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12), 2007.

[17] L. Benini, D. Bruni, A. Macii, and E. Macii. Hardware-assisted data compression for energy minimization in systems with embedded processors. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, 2002.

[18] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *ACM SIGMETRICS*, 2005.

[19] Major Bhadauria and Sally A. McKee. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2010.

[20] Suparna Bhattacharya, Karthick Rajamani, K. Gopinath, and Manish Gupta. Does lean imply green?: A study of the power performance implications of Java runtime bloat. In *ACM SIGMETRICS*, 2012.

[21] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[22] Christian Bienia, Sanjeev Kumar, and Kai Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *2012 IEEE International Symposium on Workload Characterization*, 2008.

[23] Ozlem Bilgir, Margaret Martonosi, and Qiang Wu. Exploring the potential of CMP core count management on data center energy savings. *Workshop on Energy Efficient Design (WEED)*, June 2011.

[24] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2008.

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

[26] Steve Blackburn, Martin Hirzel, Robin Garner, and Darko Stefanovic. pjbb2005, 2006. `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`.

[27] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *Transactions on Computer Systems (TOCS)*, 28, December 2010.

[28] Dominik Brodowski and Nico Golde. CPU frequency and voltage scaling code in the Linux kernel: CPUFreq governors. `http://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`.

[29] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2012.

[30] Eugenio Capra, Chiara Francalanci, and Sandra A. Slaughter. Is software "green"? application development environments and energy efficiency in open source applications. *Information and Software Technology*, 54(1), 2012.

[31] R. Carl and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Workshop on Performance Analysis and its Impact on Design (PAID)*, 1998.

[32] Trevor E. Carlson, Wim Heirman, Kenzo Van Craeynest, and Lieven Eeckhout. BarrierPoint: sampled simulation of multi-threaded applications. In *ISPASS*, 2014.

[33] Johnathan Carter, Yun He, John Shalf, Hongzhang Shan, Erich Strohmaier, and Harvey Wasserman. The performance effect of multi-core on scientific applications. In *Cray User Group Meeting*, Seattle, WA, USA, 2007.

[34] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351, 2005.

[35] James Charles, Preet Jassi, Narayan S Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the Intel® Core i7 Turbo Boost feature. In *2012 IEEE International Symposium on Workload Characterization*, pages 188–197. IEEE, 2009.

[36] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C.

Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 105–115, 2007.

[37] Tony Nowatzki Jaikrishnan Menon Chen-Han and Ho Karthikeyan Sankaralingam. gem5, gpgpusim, mcpat, gpuwattch, "your favorite simulator here" considered harmful.

[38] Ron C. Chiang and H. Howie Huang. TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2011.

[39] Yohan Chon, Elmurod Talipov, Hyojeong Shin, and Hojung Cha. Mobility prediction-based smartphone energy optimization for everyday location monitoring. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, 2011.

[40] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Pack & cap: Adaptive DVFS and thread packing under power caps. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2011.

[41] CoFluent CPR. Intel System Modeling and Simulation. `http://www.intel.com/content/www/us/en/cofluent/intel-cofluent-studio.html`.

[42] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A parallel functional simulator for GPGPU. In *MASCOTS*, 2010.

[43] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, 1996.

[44] Gary Cook, Tom Dowdall, David Pomerantz, and Yifei Wang. Clicking clean: How companies are creating the green internet. greenpeace.org, 2014.

[45] Intel Corporation. Intel 64®and IA-32 architectures software developer's manual. `http://download.intel.com/products/processor/manual/253669.pdf`.

[46] Intel Corporation. Intel®power governor. `https://software.intel.com/en-us/articles/intel-power-governor`.

[47] DaCapo Project. The DaCapo benchmark suite usage documentation, 2009. `http://www.dacapobench.org/`.

[48] V. Dalal and C. P. Ravikumar. Software power optimizations in an embedded system. In *International Conference on VLSI Design*, 2001.

[49] Danilo De Donno, Luca Catarinucci, and Luciano Tarricone. An UHF RFID energy-harvesting system enhanced by a DC-DC charge pump in silicon-on-insulator technology. *IEEE Microwave and Wireless Components Letters*, 23(6), 2013.

[50] Matthew Devuyst, Rakesh Kumar, and Dean M. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[51] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *PACT*, 2010.

[52] Mian Dong, Yung-Seok Kevin Choi, and Lin Zhong. Power modeling of graphical user interfaces on OLED displays. In *Proceedings of the Design Automation Conference (DAC)*, 2009.

[53] Jack Dongarra. China's Tianhe-2 supercomputer maintains top spot on list of world's Top500 supercomputers. Top500.org, 2015.

[54] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[55] Lieven Eeckhout. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture*, 10, 2010.

[56] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[57] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[58] Stijn Eyerman and Lieven Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. *ACM SIGPLAN Notices*, 45:91–102, March 2010.

[59] Faiza Fakhar, Barkha Javed, Raihan ur Rasool, Owais Malik, and Khurram Zulfiqar. Software level green computing for large scale systems. *Journal of Cloud Computing*, 1(1), 2012.

[60] Naila Farooqui, Andrew Kerr, Greg Eisenhauer, Karsten Schwan, and Sudhakar Yalamanchili. Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures. In *ISPASS*, 2012.

[61] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[62] Philip J Fleming and John J Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3), 1986.

[63] Jason Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. *Transactions on Computer Systems (TOCS)*, 22(2), May 2004.

[64] J. Fung and S. Mann. Computer vision signal processing on graphics processing units. In *Acoustics, Speech, and Signal Processing, 2004. ICASSP '04.*, volume 5, 2004.

[65] Karl Frlinger and Michael Gerndt. ompP: A profiling tool for OpenMP. In *Proceedings of the International Workshop on OpenMP*, 2005.

[66] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 458–469, 2011.

[67] Gartner,Inc. Worldwide traditional PC, tablet, ultramobile and mobile phone shipments on pace to grow 7.6 percent in 2014, 2013.

[68] Gartner,Inc. 4.9 billion connected "things" will be in use in 2015, 2014.

[69] Nancy Gohring. Motorola CEO: Open android store leads to quality issues. ComputerWorld, 2011.

[70] V. Govindaraju, Chen-Han Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[71] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Notices*, 17:120–126, 1982.

[72] Torbjorn Granlund. Instruction latencies and throughput for AMD and Intel x86 processors, February 2012. `http://gmplib.org/~tege/x86-timing.pdf`.

[73] Paul Griffin, Witawas Srisa-an, and J. Morris Chang. An energy efficient garbage collector for java embedded devices. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.

[74] Preston Grisham. United states tech industry employs 6.5 million in 2014. CompTIA, 2015.

[75] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the Interna-*

tional Conference on Parallel Architectures and Compilation Techniques (PACT), pages 205–216, 2010.

[76] Matt Hamblen. Mobile app download tally will soar above 102b this year. Computer World, 2013.

[77] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. SimPoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[78] Erica Check Hayden. Technology: The $1,000 genome. *Nature*, 507:294–295, 2014.

[79] Brian Hayes. The semicolon wars. *Computing Science*, 2006.

[80] Stuart Hayes. Controlling processor C-State usage in Linux. Dell Whitepaper, 2013.

[81] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The Cilkview scalability analyzer. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–156, 2010.

[82] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4), 2006.

[83] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Don Newell, Vineet Chadha, and Jaideep Moses. Rate-based QoS techniques for cache/memory in CMP platforms. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2009.

[84] Oscar Hernandez, Ramachandra C. Nanjegowda, Barbara Chapman, Van Bui, and Richard Kufrin. Open source software support for the OpenMP runtime API for profiling. In *International Conference on Parallel Processing (ICPP)*, ICPPW, pages 130–137, 2009.

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

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

[87] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[88] Urs Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 30(4), 2010.

[89] Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010.

[90] R. Hood, H. Jin, P. Mehrotra, J. Chang, J. Djomehri, S. Gavali, D. Jespersen, K. Taylor, and R. Biswas. Performance impact of resource contention in multicore systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[91] Jen-Cheng Huang, Lifeng Nai, Hyesoon Kim, and Hsien-Hsin S. Lee. TBPoint: reducing simulation time for large-scale GPGPU kernels. In *IPDPS*, 2014.

[92] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. In *Proceedings of the Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2004.

[93] Ramesh Illikkal, Vineet Chadha, Andrew Herdrich, Ravi Iyer, and Donald Newell. PIRATE: QoS and performance management in CMP architectures. *ACM SIGMETRICS*, 37, March 2010.

[94] Intel Corp. Intel OpenSource HD Graphics programmers reference man-

ual. https://01.org/linuxgraphics/sites/default/files/documentation/snb_ihd_os_vol1_part1.pdf, 2011.

[95] Intel® Corporation. Intel® Parallel Amplifier 2011. http://software.intel.com/en-us/articles/intel-parallel-amplifier/.

[96] Intel® Corporation. Intel® VTune Amplifier XE. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[97] International Technology Roadmap for Semiconductors . ITRS Report, 2009.

[98] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, ISCA '07, pages 186–197, 2007.

[99] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2006.

[100] Yuriko Ishitobi, Tohru Ishihara, and Hiroto Yasuura. Code and data placement for embedded processors with scratchpad and cache memories. *Journal of Signal Processing Systems*, 60(2), 2010.

[101] Marty Itzkowitz and Yukon Maruyama. HPC profiling with the SunStudio performance tools. In *Parallel Tools Workshop*, 2009.

[102] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. *ACM SIGMETRICS*, 35:25–36, June 2007.

[103] Abhishek Jaiantilal. i7z, 2013. http://code.google.com/p/i7z/.

[104] Ravi Jain, David Molnar, and Zulfikar Ramzan. Towards understanding algorithmic factors affecting energy consumption: Switching complexity, randomness, and

preliminary experiments. In *Joint Workshop on Foundations of Mobile Computing*, 2005.

[105] AJ Jansen and ALN Stevels. Human power, a sustainable option for electronics. In *IEEE International Symposium Proceedings on Electronics and the Environment*, pages 215–218, 1999.

[106] Yunlian Jiang and Xipeng Shen. Exploration of the influence of program inputs on CMP co-scheduling. In *European Conference on Parallel Processing (EUROPAR)*, volume 5168 of *Lecture Notes in Computer Science*, pages 263–273. 2008.

[107] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[108] Yunlian Jiang, Kai Tian, and Xipeng Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, 2010.

[109] Alexandra Jimborean, Matthieu Herrmann, Vincent Loechner, and Philippe Clauss. VMAD: a virtual machine for advanced dynamic analysis of programs. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, 2011.

[110] Haoqiang Jin, Robert Hood, Johnny Chang, Jahed Djomehri, Dennis Jespersen, Kenichi Taylor, Rupak Biswas, and Piyush Mehrotra. Characterizing application performance sensitivity to resource contention in multicore architectures. Technical Report NAS-09-002, NASA Ames Research Center, 2009.

[111] Don Jones, Jr., Simon Marlow, and Satnam Singh. Parallel performance tuning for Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell, pages 81–92, 2009.

[112] A. M. Joshi, L. Eeckhout, L. K. John, and C. Isen. Automated microprocessor stressmark generation. In *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[113] Melanie Kambadur, Sunpyo Hong, Juan Cabral, Harish Patil, Chi-Keung Luk, and Martha A. Kim. Fast computational GPU design with GT-Pin. In *Processings of the International Symposium on Workload Characterization (IISWC)*, 2015.

[114] Melanie Kambadur and Martha A. Kim. Energy exchanges: Internal power oversight for applications. Technical Report CUCS-TR-009-14, Department of Computer Science, Columbia University, 2014.

[115] Melanie Kambadur and Martha A. Kim. An experimental survey of energy management across the stack. In *Proceedings of the Annual Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, 2014.

[116] Melanie Kambadur and Martha A. Kim. Trading functionality for power within applications. In *SIGPLAN Workshop on Probabilistic and Approximate Computing at PLDI (APPROX)*, 2014.

[117] Melanie Kambadur and Martha A. Kim. NRG-loops: Adjusting power from within applications. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2016.

[118] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A Kim. Measuring interference between live datacenter applications. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2012.

[119] Melanie Kambadur, Kui Tang, and Martha A. Kim. Harmony: Collection and analysis of parallel block vectors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2012.

[120] Melanie Kambadur, Kui Tang, and Martha A. Kim. Collection, analysis, and uses of parallel block vectors. *IEEE Micro*, 99(1):1, 2013.

[121] Melanie Kambadur, Kui Tang, and Martha A. Kim. ParaShares: Finding the important basic blocks in multithreaded programs. In *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2014.

[122] Melanie Kambadur, Kui Tang, Joshua Lopez, and Martha A Kim. Parallel scaling properties from a basic block view. In *ACM SIGMETRICS*, volume 41, pages 365–366. ACM, 2013.

[123] Aman Kansal, Scott Saponas, A.J. Bernheim Brush, Kathryn S. McKinley, Todd Mytkowicz, and Ryder Ziola. The latency, accuracy, and battery (LAB) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *Proceedings of the Annual Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, 2013.

[124] Tejas Karkhanis, James E. Smith, and Pradip Bose. Saving energy with just in time instruction delivery. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2002.

[125] Miray Kas. Towards on-chip datacenters: A perspective on general trends and on-chip particulars. *The Journal of SuperComputing (SCI)*, October 2011.

[126] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of PTX kernels. In *IISWC*, 2009.

[127] Changkyu Kim, S. Sethumadhavan, D. Gulati, D. Burger, M.S. Govindan, N. Ranganathan, and S.W. Keckler. Composable lightweight processors. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 381 –394, 2007.

[128] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, 2004.

[129] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2010.

[130] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 1997.

[131] Nevin Kirman and José F. Martínez. A power-efficient all-optical on-chip interconnect using wavelength-based oblivious routing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[132] Kishonti. CompuBench CL for OpenCL and CompuBench RS for RenderScript. `http://compubench.com`, 2014.

[133] Younggyun Koh, R. Knauerhase, P. Brett, M. Bowman, Wen Zhihua, and C. Pu. An analysis of performance interference effects in virtual environments. In *International Symposium on Performance Analysis of Systems Software (ISPASS)*, april 2007.

[134] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workshop on Workload Characterization (WWW)*, 2000.

[135] H. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers (TC)*, 37(11), 1988.

[136] Michael Larabel. Benchmarking the Intel P-State, CPUfreq changes. Phoronix Media, 2013.

[137] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 75–, 2004.

[138] J. Lau, S. Schoemackers, and B. Calder. Structures for phase classification. In *ISPASS*, 2004.

[139] G. Lauterbach. Acceleration architectural simulation by parallel execution of trace samples. Technical Report TR-93-22, Sun Microsystems Laboratories Inc., 1993.

[140] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Conference on Power-Aware Computing and Systems (HotPower)*, 2010.

[141] Sangpil Lee and Won Woo Ro. Parallel GPU architecture simulation framework exploiting work allocation unit parallelism. In *ISPASS*, 2013.

[142] Ilias Leontiadis, Christos Efstratiou, Marco Picone, and Cecilia Mascolo. Don't kill my ads!: balancing privacy in an ad-supported mobile application market. In *Proceedings of the Workshop on Mobile Computing Systems & Applications*, page 2, 2012.

[143] John Levesque, Jeff Larkin, Martyn Foster, Joe Glenski, Garry Geissler, Stephen Whalen, Brian Waldecker, Johnathan Carter, David Skinner, Helen He, Harvey Wasserman, John Shalf, Hongzhang Shan, and Erich Strohmaier. Understanding and mitigating multicore performance issues on the AMD Opteron architecture. Technical Report LBNL-62500, Lawrence Berkeley National Laboratory, 2007.

[144] Joe Levi. Dalvik vs. ART: Android virtual machines and the battle for better performance, 2013. `http://pocketnow.com/2013/11/13/dalvik-vs-art`.

[145] Zhiyuan Li, Cheng Wang, and Rong Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001.

[146] Chun-Han Lin, Pi-Cheng Hsiu, and Cheng-Kang Hsieh. Dynamic backlight scaling optimization: A cloud-based energy-saving service for mobile streaming applications. *IEEE Transactions on Computers*, 63(2), 2014.

[147] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. In *Proceedings*

of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 213–224, March 2011.

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

[149] LuxMark. OpenCL Benchmarking Tool for GPUs. `http://www.luxrender.net/wiki/LuxMark`.

[150] Wenjing Ma and Gagan Agrawal. A translation system for enabling data mining applications on GPUs. In *SC*, 2009.

[151] Allen D. Malony. Event-based performance perturbation: a case study. In *Proceedings of the ACM SIGNPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 201–212, 1991.

[152] Marek Marczykowski and Krzysztof Sachanowicz. The saper project (a minesweeper game). Version X.0.14, 2013. `http://marmarek.w.staszic.waw.pl/saper/`.

[153] Ami Marowka. Back to thin-core massively parallel processors. *Computer*, 44(12), 2011.

[154] Jason Mars, Lingjia Tang, and Robert Hundt. Heterogeneity in homogeneous warehouse-scale computers: A performance opportunity. *IEEE Computer Architecture Letters*, 10(2), July 2011.

[155] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2011.

[156] Jason Mars, Lingjia Tang, and Mary Lou Soffa. Directly characterizing cross core interference through contention synthesis. In *International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, 2011.

[157] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention aware execution: online contention detection and response. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2010.

[158] C. Dianne Martin. Eniac: The press conference that shook the world. *IEEE Technology and Society Magazine*, 14(4), 1996.

[159] Michael R. Marty and Mark D. Hill. Virtual hierarchies to support server consolidation. *ACM SIGARCH Computer Architecture News*, 35:46–56, June 2007.

[160] John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta. Evaluating the effectiveness of model-based power characterization. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2011.

[161] Greg McLaren. QProf: a scalable profiler for the Q back end. MIT PhD Thesis, 1995.

[162] H. Mehta, R.M. Owens, M.J. Irwin, R. Chen, and D. Ghosh. Techniques for low energy software. In *Low Power Electronics and Design*, 1997.

[163] Miniwatts Marketing Group. Internet world stats, 2015. `http://www.internetworldstats.com/stats.htm`.

[164] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2010.

[165] Perhaad Mistry, Chris Gregg, Norman Rubin, David Kaeli, and Kim Hazelwood. Analyzing program flow within a many-kernel OpenCL application. In *GPGPU Workshop*, 2011.

[166] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Rizos Sakellariou, and Mateo Valero. FlexDCP: a QoS framework for CMP architectures. *ACM SIGOPS Operating Systems Review*, 43:86–96, April 2009.

[167] Tipp Moseley. Adaptive thread scheduling for simultaneous multithreading processors. Master's thesis, University of Colorado, 2006.

[168] Tipp Moseley, Daniel A. Connors, Dirk Grunwald, and Ramesh Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the International Conference on Computing Frontiers*, CF, pages 143–152, 2007.

[169] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.

[170] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2007.

[171] Kshirasagar Naik and David S. L. Wei. Software implementation strategies for power-conscious systems. *Mobile Networks and Applications*, 6(3), 2001.

[172] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for QoS-aware clouds. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2010.

[173] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2006.

[174] Rolf Neugebauer and Derek McAuley. Energy is just another resource: Energy accounting and energy pricing in the nemesis os. In *Workshop on Hot Topics in Operating Systems (HOTOS)*, 2001.

[175] A. T. Nguyen, P. Bose, K. Ekanadham, A. Nanda, and M Michael. Accuracy and speed–up of parallel trace–driven architectural simulation. In *Proceedings of the International Parallel Processing Symposium (IPPS)*, 1997.

[176] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 221–234, 2009.

[177] NPR Staff. The future of nanotechnology and computers so small you can swallow them. National Public Radio Archives, 2015.

[178] NVIDIA Corporation. NVIDIA visual profiler. `http://developer.nvidia.com/nvidia-visual-profiler`, 2014.

[179] M. Oskin, F. T. Chong, and M. Farrens. HLS: combining statistical and symbolic simulation to guide microprocessor design. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2000.

[180] Alexandra Ossola. Your full genome can be sequenced and analyzed for just $1,000. Popular Science, 2015.

[181] L OCallaghan, N Mishra, A Meyerson, S Guha, and R Motwani. High-performance clustering of streams and large data sets. In *Proceedings of the International Conference on Data Engineering*, 2002.

[182] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. cpuidle: Do nothing, efficiently. In *Linux Symposium*, volume 2, 2007.

[183] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor. In *Linux Symposium*, volume 2, 2006.

[184] James Pallister, Simon J. Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 2013.

[185] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 29–42, 2012.

[186] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.

[187] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. CGO, pages 2–11.

[188] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 2–11, 2010.

[189] Harish Patil and Mack Stallcup. PinPoints: Simulation Region Selection with PinPlay and Sniper. In *ISCA tutorial*, 2014.

[190] Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. Exploring hardware overprovisioning in power-constrained, high performance computing. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2013.

[191] M.K. Patterson. The effect of data center temperature on energy efficiency. In *Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITHERM)*, 2008.

[192] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *SIGMETRICS*, volume 31. ACM, 2003.

[193] Taciano Perez and César A. F. De Rose. Non-volatile memory: Emerging technologies and their impacts on memory systems. Technical Report TR-060, Pontifícia Universidade Católica do Rio, 2010.

[194] Tobias Preis, Peter Virnau, Wolfgang Paul, and Johannes J Schneider. GPU accelerated monte carlo simulation of the 2D and 3D Ising model. *Journal of Computational Physics*, 228(12), 2009.

[195] Kishore Kumar Pusukuri, David Vengerov, Alexandra Fedorova, and Vana Kalogeraki. Fact: a framework for adaptive contention-aware thread migrations. In *International Conference on Computing Frontiers (CF)*, 2011.

[196] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P Pipe,

Thomas F Wenisch, and Milo MK Martin. Computational sprinting. In *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[197] Alexander Randall. Q&A: A lost interview with ENIAC co-inventor J. Presper Eckert. Computerworld, 2006.

[198] Ashay Rane and James Browne. Performance optimization of data structures using memory access characterization. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 570 –574, 2011.

[199] K. Rangan, G. Wei, and D. Brooks. Thread motion: Fine-grained power management for multi-core systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009.

[200] Parthasarathy Ranganathan. Recipe for efficiency: principles of power-aware computing. *Communications of the ACM*, 53(4):60–67, 2010.

[201] Parthasarathy Ranganathan and Norman Jouppi. Enterprise IT trends and implications for architecture research. In *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, 2005.

[202] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. PIN: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the Workshop on Computer Architecture Education*, WCAE, 2004.

[203] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Sivius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010.

[204] Suzanne Rivoire, Parthasarathy Ranganathan, and Christos Kozyrakis. A comparison of high-level full-system power models. In *Proceedings of the Conference on Power Aware Computing and Systems*, 2008.

[205] G. Robertson. How powerful was the Apollo 11 computer? endgadget, 2009.

[206] Bratin Saha, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Mohan Rajagopalan, Jesse Fang, Peinan Zhang, Ronny Ronen, and Avi Mendelson. Programming model for a heterogeneous x86 platform. *SIGPLAN Notices*, 44:431–440, 2009.

[207] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: approximate data types for safe and general low-power computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[208] H. Sasaki, S. Imamura, and K. Inoue. Coordinated power-performance optimization in manycores. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[209] Daniele P Scarpazza, Douglas J Ierardi, Adam K Lerer, Kenneth M Mackenzie, Albert C Pan, Joseph Bank, Edmond Chow, Ron O Dror, JP Grossman, Daniel Killebrew, et al. Extending the generality of molecular dynamics simulations on a special-purpose machine. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 933–945. IEEE, 2013.

[210] Robert Schöne, Daniel Hackenberg, and Daniel Molka. Memory performance at reduced CPU clock speeds: An analysis of current x86 64 processors. In *Conference on Power-Aware Computing and Systems (HotPower)*, 2012.

[211] John S. Seng and Dean M. Tullsen. The effect of compiler optimizations on Pentium 4 power consumption. In *Workshop on Interaction Between Compilers and Computer Architectures*, 2003.

[212] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitry Vyukov. Dynamic race detection with the LLVM compiler, 2011.

[213] Dongrui She, Yifan He, B. Mesman, and H. Corporaal. Scheduling for register file energy minimization in explicit datapath architectures. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, 2012.

[214] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power containers: an OS facility for fine-grained power and energy management on multicore servers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 65–76, March 2013.

[215] Sameer S. Shende and Allen D. Malony. The Tau parallel performance system. *International Journal of High Performance Computing Applications*, 20:287–311, 2006.

[216] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, 2002.

[217] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGOPS Operating Systems Review*, 36:45–57, 2002.

[218] Huihui Shi, Yi Wang, Haibing Guan, and Alei Liang. An intermediate language level optimization framework for dynamic binary translation. *SIGPLAN Notices*, 42(5), May 2007.

[219] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *Proceedings of the Design Automation Conference (DAC)*, 2001.

[220] SiSoftware. SiSoftware: Sandra 2014. `http://www.sisoftware.net`, 2014.

[221] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction–window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Transactions on Computers (TC)*, 48(11), 1999.

[222] E. Slivka. Apple's A8 chip production for iPhone 6 underway at TSMC. MacRumors, 2014.

[223] Michael D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Department of Computer Science, Stanford University, 1991.

[224] Allan Snavely and Dean Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, 2000.

[225] S. Solomon, R.K. Thulasiram, and P. Thulasiraman. Option pricing on the GPU. In *HPCC*, 2010.

[226] Seung Woo Son, Guangyu Chen, O. Ozturk, M. Kandemir, and A. Choudhary. Compiler-directed energy optimization for parallel disk based systems. *Parallel and Distributed Systems*, 18(9), 2007.

[227] Sony Creative Software, Inc. Sony Vegas Pro. `http://www.sonycreativesoftware.com/vegaspro`, 2014.

[228] Sony Creative Software, Inc. Sony Vegas Pro Test Project. `http://download.sonymediasoftware.com/whitepapers/vp11_benchmark.zip`, 2014.

[229] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: A language and runtime system for perpetual systems. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.

[230] Standard Performance Evaluation Corporation. SPECjbb2005, 2013. `http://www.spec.org/jbb2005/`.

[231] STMicroelectronics, Inc. PGProf: parallel profiling for scientists and engineers, 2011. `http://www.pgroup.com/products/pgprof.htm`.

[232] N. Sturcken, M. Petracca, S. Warren, P. Mantovani, L.P. Carloni, A.V. Peterchev, and K.L. Shepard. A 2.5D integrated voltage regulator using coupled magnetic core inductors on silicon interposer delivering $10.8\text{A}/mm^2$. In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, 2012.

[233] Balaji Subramaniam and Wu-chun Feng. Towards energy-proportional computing for enterprise-class server workloads. In *International Conference on Performance Engineering (ICPE)*, 2013.

[234] Nathan R. Tallent and John M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. *SIGPLAN Notices*, 44:229–240, 2009.

[235] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT)*, pages 12–21, 2011.

[236] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011.

[237] Lori Thurgood, Mary J. Golladay, and Susan T. Hill. U.S. doctorates in the 20th century, 2006. National Science Foundation.

[238] Tilera Corporation. Tile-Gx Processor Family. `http://www.tilera.com/products/processors/TILE-Gx_Family/`, 2012.

[239] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Compilation techniques for low energy: An overview. In *Low Power Electronics*, 1994.

[240] My Ton, Brian Fortenbery, and William Tschudi. DC power for improved data center efficiency. 2008.

[241] Nigel Topham and Daniel Jones. High speed CPU simulation using JIT binary translation. In *MOBS*, volume 7, 2007.

[242] Gabriel Torres. Everything you need to know about the CPU c-states power saving modes, 2008. `http://www.hardwaresecrets.com/article/611`.

[243] Amin Vahdat, Alvin Lebeck, and Carla Schlatter Ellis. Every Joule is precious: the case for revisiting operating system design for energy efficiency. In *ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, EW 9, 2000.

[244] Valgrind Developers. Cachegrind: a cache and branch-prediction profiler. `http://valgrind.org/docs/manual/cg-manual.html`.

[245] N. Vallina-Rodriguez and J. Crowcroft. Energy management techniques in modern mobile handsets. *Communications Surveys Tutorials, IEEE*, PP(99), 2012.

[246] Narseo Vallina-Rodriguez and Jon Crowcroft. ErdOS: achieving energy savings in mobile OS. In *International Workshop on MobiArch*, 2011.

[247] Vibhore Vardhan, Wanghong Yuan, Albert F Harris, Sarita V Adve, Robin Kravets, Klara Nahrstedt, Daniel Sachs, and Douglas Jones. GRACE-2: Integrating fine-grained application adaptation with global adaptation for saving energy. *International Journal of Embedded Systems*, 4(2), 2009.

[248] Chris Velazco. 3,997 models: Android fragmentation as seen by the developers of opensignalmaps. TechCrunch, 2014. `http://bit.ly/1mrbUeQ`.

[249] Vasanth Venkatachalam and Michael Franz. Power reduction techniques for microprocessor systems. *ACM Computing Surveys*, 37(3), 2005.

[250] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Taylor. Conservation cores: reducing the energy of mature computations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–218, 2010.

[251] Jeffrey Vetter and Chris Chambreau. mpiP: Lightweight, Scalable MPI Profiling, 2011. `http://mpip.sourceforge.net/`.

[252] J. von Neumann. First draft of a report on EDVAC. Technical report, Univ. of Pennsylvania, 1945.

[253] Bryan Walsh. The surprisingly large energy footprint of the digital economy. Time Magazine Online, 2013.

[254] Peter Wayner. 10 reasons the browser is becoming the universal OS. InfoWorld, 2013.

[255] Gregory F. Welch. A survey of power management techniques in mobile computing operating systems. *SIGOPS Operating Systems Review*, 29(4), 1995.

[256] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4), 2006.

[257] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1995.

[258] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2005.

[259] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2003.

[260] Chao Xu, Felix Xiaozhu Lin, Yuyang Wang, and Lin Zhong. Automated OS-level device runtime power management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[261] Chi Xu, Xi Chen, R.P. Dick, and Z.M. Mao. Cache contention and application performance prediction for multi-core systems. In *International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2010.

[262] Zhiqiang Yu, Lieven Eeckhout, Nilanjan Goswami, Tong Li, Lidiya John, Hye-Jin Jin, Changsheng Xu, and Junyong Wu. GPGPU-MiniBench: Accelerating GPGPU micro-architecture simulation. *IEEE Transactions on Computers*, 2014.

[263] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. ECOSystem: managing energy as a first class operating system resource. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 123–132, 2002.

[264] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Proceedings of the ACM SIGNPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[265] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Hardware execution throttling for multi-core resource management. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2009.

[266] Yao Zhang and J.D. Owens. A quantitative performance analysis model for GPU architectures. In *HPCA*, 2011.

[267] Li Zhao, Ravi Iyer, Ramesh Illikkal, Jaideep Moses, Srihari Makineni, and Don Newell. Cachescouts: Fine-grain monitoring of shared caches in CMP platforms. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[268] Hongtao Zhong, S.A. Lieberman, and S.A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, pages 25 –36, 2007.

[269] Stuart Zweben and Betsy Bizot. 2014 Taulbee survey. *Computing Research News*, 27(5), 2015.

Appendix – Acronyms

This appendix expands all acronyms that are used in this document.

ALU — ***A****rithmetic* ***L****ogic* ***U****nit*

AMP — ***A****symmetric* ***M****ulti****P****rocessing*

AOT — ***A****head* ***O****f* ***T****ime*

API — ***A****pplication* ***P****rogram* ***I****nterface*

ASIC — ***A****pplication-****S****pecific* ***I****ntegrated* ***C****ircuit*

BIOS — ***B****asic* ***I****nput/****O****utput* ***S****ystem*

BBV — ***B****asic* ***B****lock* ***V****ector*

BB — ***B****asic* ***B****lock*

BNI — ***B****eyond* ***N****oisy* ***I****nterferer*

CFG — ***C****ontrol-****F****low* ***G****raph*

CMP — ***C****hip* ***M****ulti****P****rocessor*

CPI — ***C****ycles* ***P****er* ***I****nstruction*

CPU — ***C****entral* ***P****rocessing* ***U****nit*

DRAM — ***D****ynamic* ***R****andom-****A****ccess* ***M****emory*

DVFS — ***D****ynamic* ***V****oltage and* ***F****requency* ***T****uning*

FPGA — ***F****ield* ***P****rogrammable* ***G****ate* ***A****rray*

FPU — ***F****loating* ***P****oint* ***U****nit*

EU — ***E****xecution* ***U****nit*

FLOPS — ***F****loating* ***P****oint* ***O****perations* ***P****er* ***S****econd*

GB — ***G****iga****B****yte*

GPGPU — ***G****eneral* ***P****urpose* ***G****raphics* ***P****rocessing* ***U****nit*

GPS — ***G****lobal* ***P****ositioning* ***S****ystem*

GPU — ***G****raphics* ***P****rocessing* ***U****nit*

HDD — ***H****ard* ***D****isk* ***D****rive*

HPC — ***H****igh* ***P****erformance* ***C****omputing*

I/O — ***I****nput/****O****utput*

ILP — ***I****nstruction* ***L****evel* ***P****arallelism*

IoT — ***I****nternet* ***o****f* ***T****hings*

IPC — ***I****nstructions* ***P****er* ***C****ycle*

IPS — ***I****nstructions* ***P****er* ***S****econd*

ISA — ***I****nstruction* ***S****et* ***A****rchitecture*

IT — ***I****nformation* ***T****echnology*

JIT — ***J****ust-****I****n-****T****ime (Compiler)*

JVM — ***J****ava* ***V****irtual* ***M****achine*

L1 — ***L****evel-****1*** *Cache*

L2 — ***L****evel-****2*** *Cache*

LCD — ***L**iquid **C**rystal **D**isplay*

LLC — ***L**ast **L**evel **C**ache*

LLVM — ***L**ow **L**evel **V**irtual **M**achine*

MB — ***M**ega**B**yte*

MEM — ***M**emory*

MHz — ***M**ega **H**ertz*

MIMD — ***M**ultiple **I**nstruction **M**ultiple **I**Data*

MPI — ***M**essage **P**assing **I**nterface*

MRAM — ***M**agnetoresistive **R**andom-**A**ccess **M**emory*

MSR — ***M**odel **S**pecific **R**egister*

NRG — *Energy*

NUMA — ***N**on-**U**niform **M**emory **A**ccess*

NVRAM — ***N**on–**V**olatile **R**andom-**A**ccess **M**emory*

OOO — ***O**ut **O**f **O**rder*

OS — ***O**perating **S**ystem*

PBV — ***P**arallel **B**lock **V**ector*

PC — ***P**ersonal **C**omputer*

PCRAM — ***P**hase **C**hange **R**andom-**A**ccess **M**emory*

QoS — ***Q**uality **o**f **S**ervice*

RAPL — *(Intel's) **R**unning **A**verage **P**ower **L**imit*

RRAM — ***R**esistive **R**andom-**A**ccess **M**emory*

SIMD — ***M**ultiple **I**nstruction **M**ultiple **I**Data*

SMT — ***S**imultaneous **M**ulti**T**hreading*

SPI — ***S**econds **P**er **I**nstruction*

SRAM — ***S**tatic **R**andom-**A**ccess **M**emory*

SoC — ***S**ystem **o**n **C**hip*

TB — ***T**era**B**yte*

TDP — ***T**hermal **D**esign **P**ower / **T**hermal **D**esign **P**oint*

TLP — ***T**hread **L**evel **P**arallelism*

VM — ***V**irtual **M**achine*