Heterogeneity and Density Aware Design of Computing Systems

A dissertation submitted in partial satisfaction of the requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Manish Arora

Committee in charge:

Professor Dean M. Tullsen, Chair
Professor Renkun Chen
Professor George Porter
Professor Tajana Simunic Rosing
Professor Steve Swanson

2018
The dissertation of Manish Arora is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2018
DEDICATION

To my family and friends.
Do not be embarrassed by your failures,
learn from them and start again.
—Richard Branson

If something’s important enough, you should try.
Even if the probable outcome is failure.
—Elon Musk
TABLE OF CONTENTS

Signature Page ........................................ iii
Dedication ........................................ iv
Epigraph ........................................ v
Table of Contents .................................. vi
List of Figures .................................... viii
List of Tables ..................................... x
Acknowledgements ................................ xi
Vita ................................................ xiii
Abstract of the Dissertation ...................... xvi

Chapter 1 Introduction ................................ 1
  1.1 Design for Heterogeneity ....................... 2
    1.1.1 Heterogeneity Aware Design ............. 3
  1.2 Design for Density ............................. 6
    1.2.1 Density Aware Design .................. 7
  1.3 Overview of Dissertation .................... 8

Chapter 2 Background ................................ 10
  2.1 GPGPU Computing ............................... 10
  2.2 CPU-GPU Systems ............................... 13
  2.3 Power Gating .................................. 16
  2.4 Thermal Mitigation ............................ 19

Chapter 3 CPU Design for CPU-GPU Systems ....... 22
  3.1 Introduction .................................. 23
  3.2 Background .................................... 25
  3.3 Benchmarks .................................... 25
  3.4 Experimental Methodology ................... 27
  3.5 Results ....................................... 29
    3.5.1 CPU Execution Time ...................... 29
    3.5.2 ILP ........................................ 31
    3.5.3 Branch Results ............................ 32
    3.5.4 Load and Store Results ................. 34
    3.5.5 Vector Instructions ..................... 35
LIST OF FIGURES

Figure 2.1: Contemporary GPU architecture .................................................. 11
Figure 2.2: Streaming Multiprocessor (SM) architecture ............................. 12
Figure 2.3: Chip integrated CPU-GPU architecture ..................................... 14
Figure 2.4: Utilization and number of idle events for various applications ..... 17
Figure 2.5: (a) Fixed preflush filtering design space. (b) Autodemotion design space . 18

Figure 3.1: Time spent on the CPU. The 11 CPU-Only applications are summarized, since those results do not vary ................................. 30
Figure 3.2: Instruction level parallelism with and without GPU. Not all bars appear in the CPU-Only applications, since they do not vary post-GPU. This is repeated in future plots ........................................... 31
Figure 3.3: Distribution of branch types with and without GPU ..................... 32
Figure 3.4: Distribution of (a) load types and (b) store types with and without GPU ........................................... 33
Figure 3.5: Frequency of vector instructions with and without GPU ............... 35
Figure 3.6: Thread level parallelism with and without GPU .......................... 36

Figure 4.1: (a) Average number of idle events per sec (line) and average percentage of cache dirty on idle (bar). (b) Impact of idle management policies on average power (modeled) for 16nm FinFET ................................. 43
Figure 4.2: Cumulative distribution function (CDF) of idle events .................. 48
Figure 4.3: (a) Duration of active events preceding long idle events. (b) Growth of cache dirtiness with time .................................................. 50
Figure 4.4: CPU behavior and steps in clock gated vs. power gated state ........ 51
Figure 4.5: Measured package power (msec interrupts) ............................... 54
Figure 4.6: Hardware measured vs. modeled at PreFlush 0sec (a) Power gates per second. (b) Active times, .......................................................... 58
Figure 4.7: Variation in energy break-even point ........................................... 60
Figure 4.8: Summary of (a) Energy results. (b) Performance results .............. 64
Figure 4.9: (a) Energy for FinFET 16nm. (c) Performance for FinFET 16nm .... 66
Figure 4.10: Energy efficiency of C6 management schemes for FinFET 16nm. Benchmarks are sorted by decreasing utilization ........................... 68

Figure 5.1: (a) Power per 1U, and (b) Sockets per 1U for 400 server designs released from 2007-2016 ................................................................. 71
Figure 5.2: Ansys Icpeak CFD model of a HPE Moonshot Proliant M700 [Prob] cartridge ................................................................. 74
Figure 5.3: (a) Organization, and (b) Relative performance of Coolest First (CF) and Hottest First (HF) scheduling for coupled and un-coupled designs ........... 75
Figure 5.4: Thermal coupling in density optimized servers ............................ 76
Figure 5.5: (a) Mean socket entry temperature, and (b) Socket entry temperature coefficient of variance, a measure of the variance in heat across the coupled sockets [COV] ........................................... 79
| Figure 5.6: (a) Average job duration, and (b) Coefficient of variance of job durations within benchmark sets. | 82 |
| Figure 5.7: (a) Workload power (90C) with varying p-states. (b) Relative workload performance vs performance at 1900MHz. | 83 |
| Figure 5.8: Side view of modeled M700 cartridge. | 85 |
| Figure 5.9: (a) Temperature differences between hottest and coolest spots, and (b) Max temperature vs. power. | 86 |
| Figure 5.10: Proposed max temperature model validation. | 87 |
| Figure 5.11: Average runtime expansion vs. CF for existing thermal aware scheduling mechanisms at 30% and 70% load for the Computation workload (lower is better). | 92 |
| Figure 5.12: Organization of zones in the HPE Moonshot. Each row consists of 3 cartridges (12 sockets) and 6 zones. There are a total of 15 rows in the system. | 93 |
| Figure 5.13: Average frequency, work performed in front half, back half and even zones of the HPE Moonshot for various schemes at (a) 30% load, and (b) 70% load. | 94 |
| Figure 5.14: Performance vs. CF baseline for various schemes at various load values for different benchmarks (a) Computation, (b) GP, and (c) Storage. | 101 |
| Figure 5.15: ED2 vs. CF baseline for various schemes at various load values for different benchmarks (a) Computation, (b) GP, and (c) Storage. | 102 |
LIST OF TABLES

Table 3.1: CPU-GPU Benchmark Description. ................................. 26

Table 4.1: Benchmark classification, active power (modeled), events per module, and power for different idle management policies (modeled) using 28nm planar technology. Oracle assumes perfect knowledge of both idle durations and dirtiness. ................................................................. 45

Table 4.2: Parameters of the trace based simulation model. ......................... 57

Table 5.1: Recent density optimized systems (B:Blade, CR: Card, C: Cartridge, S:Socket, SL:Sled, T: Tray, and R:Row) .................................................. 77

Table 5.2: Airflow requirements for server systems. ............................... 78

Table 5.3: Overall simulation model parameters. ................................... 89
ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to Professor Dean M. Tullsen for guiding my dissertation. Dean has encouraged, supported and mentored me. For that, I am grateful to Dean. Dean’s smiling and calm face always gave me hope, irrespective of everything else. Dean supported me, when I might have needed it the most, and that support has enabled me to finish this dissertation. My thanks are due to my PhD committee members Steven Swanson, Tajana Rimunic Rosing, George Porter and Renkun Chen for providing me with valuable feedback.

I thank my father Nand Kishore Arora, and my mother Indu Arora. No one can replace what they have done for me. I truly am, standing on the shoulder of giants. I thank my sister Dr. Chetna Bhatia. Chetna has always given me positive energy and encouraged me to go on. I thank my wife Shaveta Wadhwa Arora, for supporting me and enduring this journey with me.

I thank my childhood friends Alex Gedala (founder, RedLizard Studiouz) and Vivek Puri (founder, TabletWise.com, Sarkari Result (SarkariResultNow.com) & TheWiseDictionary.com). They have given me valuable lessons on life and work. I am sincerely thankful to my close friends; Jagpreet Singh, Harmandeep Singh, Poonam Birdi, Daman Singh Ryait, Deborah Lee, Indrani Paul, Sachin Shende (co-founder, KisanHub), and Manish Gupta. I am not sure if I would have been able to finish, had they not been there. I thank my tennis buddies Guruprasad Rao, Sumit Sadana, Matthew Thomas Evans and Jim Miller, for the positivity and laughs.

I would like to thank my colleagues; Sidhartha Nath, Srilatha Manne, Nuwan Jayasena, Yasuko Eckert, Shubra Mazumdar, Ayse Coskun, Fulya Kaplan, Wei Huang, Xudong An, Matt Skach, Lingjia Tang and Jason Mars for sharing their insights. Perhaps, I have missed the contributions of many others who enabled me to finish this journey. My sincere thanks to you.

Chapter 2, in part, is a reprint of the material as it appears in the proceedings of Sigmetrics 2014. Arora, Manish; Manne, Srilatha; Eckert, Yasuko; Paul, Indrani; Jayasena; Nuwan; Tullsen, Dean, A comparison of core power gating strategies implemented in modern hardware. Proceedings of the 2014 ACM SIGMETRICS / International Conference on Measurement and
Modeling of Computer Systems. The dissertation author was the primary investigator and author of this paper.

Chapter 3 in full, is a reprint of the material as in it appears in IEEE Micro. Arora, Manish; Nath, Siddhartha; Subhra Mazumdar; Baden, Scott; Tullsen, Dean, Redefining the Role of the CPU in the Era of CPU-GPU Integration, IEEE Micro, December, 2012. The dissertation author was the primary investigator and author of this paper.

Chapter 4 in full, is a reprint of the material as in it appears in the proceedings of the HPCA 2015. Arora, Manish; Manne, Srilatha; Paul, Indrani; Jayasena; Nuwan; Tullsen, Dean, Understanding Idle Behavior and Power Gating Mechanisms in the Context of Modern Benchmarks on CPU-GPU Integrated Systems. IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015. The dissertation author was the primary investigator and author of this paper.

Chapter 5 in part, is currently being prepared for submission for publication of the material. Arora, Manish; Skach, Matt; An, Xudong; Hung, Wei; Mars, Jason; Tang, Lingjia; Tullsen, Dean, Understanding the Impact of Socket Density In Density Optimized Servers. The dissertation author was the primary investigator and author of this material.
VITA

1996-2000 Bachelor in Technology (Honors), IIT Kharagpur
2000-2001 Sasken Communication Technologies, Bengaluru
2001-2002 Emuzed India Private Limited, Bengaluru
2003-2008 Samsung Electronics, Republic of Korea
2008-2010 Master in Science, University of Texas at Austin
2012-2018 AMD Research, Sunnyvale
2015 C. Phil. in Computer Science, University of California, San Diego
2010-2018 Doctor of Philosophy, University of California, San Diego

PUBLICATIONS


Han-gil Moon, Manish Arora. “Virtual Source Location Information Based Matrix Decoding System”, AES Convention, 2006.


The number of programmable cores that are available in systems continues to increase with advances in device scaling, integration, and iterative improvements. Today, systems are not just integrating more cores, but also integrating a variety of different types of processing cores, resulting in dense heterogeneous systems. However, important questions remain about the design methodology for dense heterogeneous systems. This thesis seeks to address these questions.

One typical methodology for heterogeneous system design is to comprise systems by using parts of homogeneous systems. Another commonly used technique to enable density is replication. However, these design methodologies are “heterogeneous system oblivious” and “density oblivious”. The components of the system are not aware or optimized for the
heterogeneous system they would become a part of. Nor are they aware of the existence of other 
replicated components. This thesis shows that “heterogeneous system oblivious” and “density 
oblivious” design methodologies result in inefficient systems. This thesis proposes heterogeneity 
and density aware approaches to designing dense heterogeneous architectures.

This dissertation proposes heterogeneity aware CPU architecture directions for integrated 
CPU-GPU systems. The thesis demonstrates that, in CPU-GPU systems the CPU executes 
code this is profoundly different than in past CPU-only environments. The new code has 
lower instruction-level parallelism, more difficult branch prediction, loads and stores that are 
significantly harder to predict, and exhibits much smaller gains from the availability of multiple 
CPU cores. Such a heterogeneity aware refactoring of CPU architectures can improve the 
performance of CPU-GPU systems.

This dissertation proposes techniques to improve power gating for integrated CPU-GPU 
processors by accounting for short idle events. Power gating is an effective mechanism to reduce 
idle power. However, power gating incurs non-trivial overheads and can cause negative savings if 
the idle duration is short. Even when durations are long, it may still not be beneficial to power gate 
because of the overheads of cache flushing. As heterogeneity increases the occurrence of such 
idle events, it is important to make idle power management heterogeneity aware. This research 
proposes techniques to accurately predict idle durations and account for dynamic variations in the 
brake-even point caused by varying cache dirtiness.

This dissertation proposes density aware job scheduling algorithms for dense servers. The 
need to pack more compute in the same physical space has led to the development of density 
optimized server designs. This thesis demonstrates that increased socket density leads to large 
temperature variations among sockets due to inter-socket thermal coupling. The work proposes 
density aware scheduling techniques that account for inter-socket thermal coupling and improve 
system performance.
Chapter 1

Introduction

The number of programmable cores that are available in systems continues to increase with advances in device scaling, system integration, and iterative design improvements. We define cores per unit volume as core density. Core density is consistently increasing across various granularities from on die to servers. As system integration capabilities improve, core density increases are primarily heterogeneous. Today, systems are not just integrating more cores but also integrating a variety of different types of processing cores. This is because components of a heterogeneous system can be better tuned to portions of workloads, resulting in performance and energy efficiency improvements.

An example of this trend can be seen in mainstream consumer computing systems. These systems now combine a multi-core CPU and a multi-core GPU on the same chip. These systems provide about 4-8 CPU cores with 8-16 GPU cores. Similarly, high performance computing systems such as the Cray XC-50 have begun to use GPUs with CPUs on the same node. The trend can also be seen in servers. Traditional Blade servers integrate 2–4 CPU sockets per unit volume space. New space optimized server designs such as the HPE Moonshot can integrate as many as 25–45 CPU-GPU sockets, per 1U of space.

Increases in core density are considered beneficial because it can increase the number
of computations that can be performed in a fixed size (e.g., in a standard rack). We define computations per unit volume as computational density. However, core density may not provide a corresponding increase in computational density. This can be because of various reasons. For example, each component of the heterogeneous system may not have been optimally designed, or the different components of the heterogeneous system may not function well when combined to finish a single task. This is problematic as the economic incentives of hardware improvement are strongly dependent on improving computational density and not just core density.

The challenge, then, is to develop techniques that can improve computational density for systems with increasing core density. This thesis studies three state-of-the-art systems and aims to answer the following key question – As the number of heterogeneous cores increase, how can we improve the computational density in these systems? That is, what are the factors that impede computational density for these systems, and how can we further improve it?

This thesis seeks to address these specific questions.

1.1 Design for Heterogeneity

One typical methodology for heterogeneous system design is to comprise systems by using parts of homogeneous systems. This is a commonly practiced methodology in the industry to design heterogeneous systems. In this methodology, components of homogeneous designs are optimized for energy or performance, replicated for density, and then combined to form a heterogeneous system. Such a methodology re-uses existing products, reduces the design cycle, and lowers costs. This methodology is also driven by modular design and is relatively simple to implement.

However, such a design methodology is “heterogeneous system oblivious”. The components of the system are not aware or optimized for the heterogeneous system they would become a part of. Rather, these components have been designed for homogeneous systems. Such a
methodology puts unrealistic constraints on component design. For example, components may be
designed to target a different set of applications that they might eventually target as a part of the
heterogeneous system.

This thesis shows that “heterogeneous system oblivious” design methodology results in
sub-optimal processors. This is because processors designs have not been designed keeping in
mind the eventual target workload. For example, consumer computing systems today combine
4-8 multi-core CPUs with 8-16 GPU cores. However, multi-cores and GPUs may both target
the parallel portions of code, even though only one of them may be used at any given time for
portions of a single program. Hence there is a need to design heterogeneous systems differently.

1.1.1 Heterogeneity Aware Design

This thesis proposes a heterogeneity aware approach to designing heterogeneous archi-
tectures. A heterogeneity aware methodology designs components that are cognizant of their
role in the target system and are specifically optimized for that role. This enables efficiency and
performance as components are better optimized for workloads that would run on them. Specif-
ically, this thesis addresses two specific aspects that introduce inefficiency in “heterogeneous
system oblivious” designs and proposes corresponding heterogeneity aware improvements. This
thesis proposes improvements to CPU architecture and idle power management in heterogeneous
systems.

CPU Architecture

Traditionally, CPU cores are designed to be general purpose. They have to be able to target
a wide variety and diversity of applications. These include applications with irregular control flow
and without high data-level parallelism. These also include applications that have components
that are amenable to parallelization. On the other hand, modern GPUs contain hundreds of ALUs,
hardware thread management, access to fast on-chip memory, and high-bandwidth external
memory. This translates to peak performance that is significantly better than CPUs for data parallel applications. As CPUs and GPUs combine on the die to form heterogeneous systems, there seems to be significant overlap between these architectures. However, the code the CPU will run, once appropriate computations are mapped to the GPU, has significantly different characteristics than the original code (which previously would have been mapped entirely to the CPU). But the CPU has not been optimized for this new code.

This thesis proposes a heterogeneity aware approach to CPU design for CPU-GPU systems. We propose that in a CPU-GPU system, the general-purpose CPU need no longer be fully general-purpose. It will be more effective if it becomes specialized to the code that cannot run on the GPU. Instead of using CPU cores from multi-core systems, we propose specializing the CPU so that it better matches the workloads that it would execute in a CPU-GPU system. Our research identifies the directions in which to push future CPU designs.

Our work shows that the code that runs on CPUs in CPU-GPU systems has significantly lower instruction-level parallelism than CPU only code. That means that continuously increasing the instruction window size may not result in large benefits. We demonstrate that the new CPU code has harder to predict branches. This means that more complicated branch prediction logic will need to be integrated in the future than what is used currently. We show that loads and stores are also significantly harder to predict. That makes the pre-fetching problem more difficult to solve. We demonstrate that applications exhibit much smaller gains from the availability of multiple CPU cores when GPUs are present. This enables designers to put practical limits to the number of CPU cores that should be integrated as part of CPU-GPU systems. In summary, we propose a heterogeneity aware refactoring of CPU architectures that are components of CPU-GPU systems. The proposed guidelines can help improve CPU performance and the computational density of CPU-GPU integrated systems.
Idle Power Management

A CPU-GPU system aims to improve performance by running parts of the application on components that are optimized to run them. For example, data parallel portions on the GPU and irregular code on the CPU. This is advantageous as GPUs are optimized to execute parallel portions and CPUs may be better optimized to run irregular code. These advantages span both performance as well as energy consumption. However, the existence of heterogeneity introduces idleness in the CPU when the GPU is executing its portions of code, and vice-versa. Hence, the overall energy consumption for such a system is a function of energy behavior both when the CPU and GPU are active, as well as, when the CPU and GPU are idle. While prior research has focused on active energy optimizations, idle energy behavior has not been well-studied. Specifically, we demonstrate how heterogeneity causes idleness in the CPU in applications and how an heterogeneity aware optimization of idle power management can be beneficial.

In this thesis, we show that a large number of important consumer computing applications have significant idle time while the application is “running.” Idleness while running applications happens for several reasons, such as the interactive or I/O-intensive nature of the applications or insufficient parallelism to keep all cores busy. The inclusion of general purpose programmable GPUs is expected to further increase the appearance of rapid idleness even in traditional compute intensive application domains. Our research shows that multi-core idle power management schemes are not suited to short idle events and work poorly for heterogeneous CPU-GPU systems.

As overall energy consumption is heavily impacted by idle power, there is a need to develop better idle power management mechanisms for heterogeneous systems.

Our work proposes a set of techniques to improve CPU power gating in integrated CPU-GPU processors. Power gating is an effective mechanism to reduce idle power as it can reduce power to near zero when the core is idle. However, entering and exiting from power gating incurs non-trivial overheads and can cause negative savings if the idle duration is short. Even when durations are relatively long, it may still not be beneficial to power gate because of the
overheads of cache flushing. This is because the benefits of power gating are primarily determined by the duration the core will stay power gated. Our research proposes techniques to accurately predict idle durations. We show that the availability of a good estimate of idle duration can help us make much better power gating decisions. We demonstrate that a fixed break-even point is invalid for core power gating, as unlike functional units, cores are not stateless. We propose new power gating mechanisms that combine idle duration prediction with cache dirtiness measurement to account for dynamic variations in the break-even point. Our work demonstrates an energy reduction of over 8% over existing power gating mechanisms.

1.2 Design for Density

System integration and iterative design improvements are enabling new generations of systems with extreme density. The density optimizations in these systems are being enabled primarily with replication. An example of such a system is the HPE Moonshot. The HPE Moonshot packs 180 sockets in a 4U space. Sockets in the HPE Moonshot are organized as cartridges and rows. There are four sockets per cartridge, and three cartridges per row. Fifteen such rows of three cartridges each are replicated in a 4U space to create a server node with 180 sockets. All of these sockets share the chassis, power supply, networking, and the cooling system.

However, sockets in such a system may be “density oblivious”. That is, individual sockets of these systems may not know that many such sockets exist, even though they might interact and be affected by others. An example of this is thermal management in individual sockets. Thermal management of each socket occurs individually, where each socket may change power and performance levels individually in response to its thermal conditions. However, as the cooling system is shared between sockets, power dissipation in an individual socket may impact thermals on several other sockets. This example shows that such “density oblivious” designs ignore component interactions. Hence there is a need to design dense systems differently.
1.2.1 Density Aware Design

This thesis proposes a density aware approach to design. In the proposed methodology, components of the system are cognizant that other components exists, and understand how they might be affected or interact with other components. This enables efficiency as components can better account for interactions and mitigate potential sources of performance loss. Specifically, this thesis addresses one design aspect that introduces inefficiency in “density oblivious” designs. The thesis proposes improvements to job scheduling in dense server systems by accounting for thermal coupling amongst sockets in dense servers. Thermal coupling refers to sharing of heat between sockets because of hot air from one socket blowing over another socket.

Job Scheduling

The goal of a job scheduler is to maximize performance and energy efficiency for a given set of jobs. Traditional schedulers can account for various factors such as socket temperature, performance, or workload balancing across sockets. However, most existing schedulers do not account for negative interactions across sockets. For example, scheduling a job on a socket may cause the socket to heat up. This causes the air that flows over this particular socket to also heat up. This hot air may then flow over a downstream socket in a dense system that shares cooling between sockets. Hence, “density oblivious” scheduling algorithms that account for the thermals of only a single socket may not work well in dense systems.

In this thesis, we investigate the degree of thermal interactions that occur between sockets in a dense server that are caused by the sharing of the cooling system. We find that thermal interactions in these designs are primarily uni-directional and fairly severe. At the same time, these interactions are unavoidable, at least amongst certain sockets because of how the cooling system works.

This dissertation proposes better scheduling job algorithms for dense servers that suffer from such thermal interactions. Using a computational fluid dynamics model, this research
quantifies temperature heterogeneity among sockets due to inter-socket thermal coupling. Based on our thermal analysis, we propose a new scheduling policy named CouplingPredictor (CP). The CP algorithm extends the predictive job scheduling algorithms by taking into account thermal coupling effects. CP predicts the performance of not only the socket where the job is scheduled but also the performance of all other sockets that are downstream to this socket. It chooses sockets for scheduling that result in overall benefit and significantly improves system performance.

1.3 Overview of Dissertation

Chapter 2 gives background information on design and evolution of CPU-GPU systems. GPU computing has emerged in recent years as a viable execution platform for throughput oriented applications or regions of code. The chapter describes the state of the art general purpose programmable GPU architectures, including collaborative CPU-GPU execution schemes. The chapter provides background on power gating strategies employed in current generation hardware. Last, the chapter provides background and related work information for commonly used scheduling techniques.

Chapter 3 studies CPU architecture directions for integrated CPU-GPU systems. The work shows that in such systems, the CPU executes code this is profoundly different than in past CPU-only environments. The new code has lower instruction-level parallelism, more difficult branch prediction, loads and stores that are significantly harder to predict, and exhibits much smaller gains from the availability of multiple CPU cores. Refactoring CPU architectures using these guidelines can help improve computational density of integrated CPU-GPU systems.

Chapter 4 proposes techniques to improve power gating for integrated CPU-GPU processors. Power gating is an effective mechanism to reduce idle power. However, power gating incurs non-trivial overheads and can cause negative savings if the idle duration is short. Even when durations are long, it may still not be beneficial to power gate because of the overheads of cache
flushing. This research proposes techniques to accurately predict idle durations and account for dynamic variations in the break-even point caused by varying cache dirtiness.

Chapter 5 proposes better scheduling job algorithms for dense servers. The need to pack more compute in the same physical space has led to the development of density optimized server designs. This research demonstrates that increased socket density leads to large temperature heterogeneity among sockets due to inter-socket thermal coupling. The work proposes scheduling techniques that account for inter-socket thermal coupling and improve system performance.

Chapter 6 summarizes the contributions of this dissertation.
Chapter 2

Background

This chapter provides background information related to this thesis. Section 2.1 explains how GPU architecture has evolved to handle general purpose computing workloads. Section 2.2 provides background on the architecture and design of CPU-GPU systems. Section 2.3 provides details of power gating strategies used in current hardware. Section 2.4 gives details and background information related to thermal mitigation strategies employed in computing systems.

2.1 GPGPU Computing

GPU computing has emerged in recent years [Aro12] as a viable execution platform for throughput oriented applications or regions of code. In this section, we seek to understand GPU architectures and how they have evolved recently for general purpose computing.

The modern GPU has evolved from a fixed function graphics pipeline which consisted of vertex processors running vertex shader programs and pixel fragment processors running pixel shader programs. Vertex processing consists of operations on point, line and triangle vertex primitives. Pixel fragment processors operate on rasterizer output to fill up the interiors of triangle primitives with interpolated values. Traditionally, workloads consisted of more pixels than vertices and hence there were greater number of pixel processors. However, unbalance in
modern workloads influenced a unified vertex and pixel processor design. Unified processing, first introduced with the NVIDIA Tesla [LNOM08], enabled higher resource utilization, and enabled the development of generalized programmable GPU processor designs.

Figure 2.1 shows a block diagram of contemporary NVIDIA GPGPUs [LNOM08, WKP11, GPU12]. The GPU consists of streaming multiprocessors (SMs), 6 high-bandwidth DRAM channels and on-chip L2 cache. The number of SMs and cores per SM varies as per the price and target market of the GPU. Figure 2.2 shows the structure of an SM. An SM consists of 32 single instruction multiple thread (SIMT) lanes that can collectively issue 1 instruction per cycle per thread for a total of 32 instructions per cycle per SM. Threads are organized into groups of 32 threads called “Warps”. Scheduling happens at the granularity of warps and all the threads in a warp execute together using a common program counter. As shown in figure 2.2, SIMT lanes have access to a fast register file and on-chip low latency scratch pad shared memory / L1 caches. Banking of the register file enables sufficient on-chip bandwidth to supply each thread with two
input and 1 output operand each cycle. The operand buffering unit acts as a staging area for computations.

GPUs rely on massive hardware multi-threading to keep arithmetic units occupied. They maintain a large pool of active threads organized as warps. For example, NVIDIA Fermi supports 48 active warps for a total of 1536 threads per SM. To accommodate the large set of threads, GPUs provide large on-chip register files. Fermi has a per SM register file size of 128KB or 2132-bit registers per thread at full occupancy.

GPUs are designed to reduce the cost of instruction and data supply. For example, SIMT processing allows GPUs to amortize cost of instruction fetch since a single instruction needs to be fetched for a warp. Similarly, large on-chip register files reduce spills to main memory. Programmers have the additional option of manually improving data locality by using scratchpad style shared memory. There is explicit programmer support to enable this.

GPUs have been designed to scale. This has been achieved with the lack of global structures. For example, unlike CPUs, the SMs have simple in-order pipelines, albeit at a much lower single thread performance. Instead of seeking performance via caches and out-of-order
processing over large instruction windows, GPUs incorporate zero overhead warp scheduling and hide large latencies via multithreading. There is a lack of global thread synchronization i.e. only threads within an SM can synchronize together and not across the whole machine. Lastly, there is a lack of global wires to feed data. Instead, a rich on-chip hierarchy of large registers files, shared memory and caches is used to manage locality. Such features reduce power consumption and allow GPUs to scale with lower technology nodes [ND10, KDK+11].

The details provided in this section demonstrate the suitability of current GPU architectures for throughput computing applications. Next, we examine how GPUs are being integrated with CPUs on the same die.

### 2.2 CPU-GPU Systems

In this section, we examine existing research directions for chip integrated CPU-GPU systems. We discuss proposals to improve the performance of CPU-GPU systems by looking at how shared components such as last-level caches and memory controllers could be evolved. We then discuss collaborative CPU-GPU execution and management schemes.

Recently AMD (Fusion APUs) [The], Intel (Sandy Bridge) [Jia] and ARM (MALI) [ARM] have released solutions that integrate general purpose programmable GPUs together with CPUs on the same die. In this computing model, the CPU and GPU share memory and a common address space. These solutions are programmable using OpenCL [Khr] or solutions such as DirectCompute [Mic]. Integrating a CPU and GPU on the same chip has several advantages. First is cost savings because of system integration and the use of shared structures. Second, this promises to improve performance because no explicit data transfers are required between the CPU and GPU [AMDb]. Third, programming becomes simpler because explicit GPU memory management may not be required.

Not only does CPU-GPU chip integration offer performance benefits but it also enables
new directions in system development. Reduced communication costs and increased bandwidth have the potential to enable new optimizations that were previously not possible. At the same time, there are new problems to consider.

Figure 2.3: Chip integrated CPU-GPU architecture.

Figure 2.3 shows a block diagram of an integrated CPU-GPU system. As we can see from the figure, last level caches and the memory controller are shared amongst the CPU and GPU. The integrated system brings new challenges for these components because of CPU and GPU architectural differences. Also, a new modeling and design space exploration techniques are needed to keep design time manageable. While CPUs depend on caches to hide long memory latencies, GPUs employ multi-threading together with caching to hide latencies. This creates opportunities for optimizations of shared structures.

TAP utilizes this architectural difference to allocate cache capacity for GPU workloads in CPU-GPU systems. GPUs trade-off memory system latency to bandwidth by having a lot of outstanding requests to the memory system. Memory intensive CPU workloads can potentially cause GPU delays and lead to missed real-time deadlines on graphics workloads.

Jeong et al. propose dynamic partitioning of off-chip memory bandwidth
between the CPU and GPU to maintain a high quality of service for the overall system. Typical memory controllers prioritize CPU requests over GPU requests as the CPU is latency sensitive and the GPU is designed to tolerate long latencies. However, such a static memory controller policy can lead to an unacceptably low frame rate for the GPU. Correspondingly, prioritizing GPU requests can degrade the performance of the CPU. The authors scheme is targeted towards system-on-chip architectures with multicore CPUs and graphics only GPUs.

COMPASS [WL10] proposes the use of idle GPU resources to act as data prefetchers for CPU execution. The authors suggest using GPU resources in two specific ways. First, they propose the use of large GPU register files as prefetcher storage structures. The 32KB – 64KB of register file space per SM provides sufficient storage for the implementation of state of the art prefetching algorithms. These schemes have prohibitive costs which makes their inclusion into commercial designs difficult. Using GPU resources drastically reduces overhead. Second, the authors propose the use of programmable GPU execution threads as logic structures to flexibly implement prefetching algorithms.

Yang et al. [YXMZ12] propose the use of CPU based execution to prefetch requests for GPGPU programs. First, they develop a compiler based infrastructure to extract memory address generation and accesses from GPU kernels to create a CPU pre-execution program. Once the GPU kernel is launched, the CPU runs the pre-execution program. To make the pre-execution effective, the CPU needs to run sufficiently ahead so as to bring relevant data into the shared LLC. However, the execution should not run too far ahead that the prefetched data are replaced before being utilized. The authors propose schemes to manage prefetch effectiveness.

A new development beyond on die CPU and GPU integration is the integration of CPUs and GPUs along with heterogeneous memories on the same die. This is leading to new research in the areas of efficient multi-threading [LG18a, LG18b], programing models [LGA+12], data placement [GRM+17], and reliability [Gup17, GSR+18, GLK+17, GRL+17, GRM+16].

This section provided details of how CPUs and GPUs are being integrated on the same
2.3 Power Gating

Idle power is a significant contributor to overall energy consumption in modern processors. In this section, we examine state-of-the-art C6 entry algorithms and present a comparative analysis in the context of consumer and CPU-GPU benchmarks.

Leakage power minimization is a first-order goal of modern processor design. While leakage power scales with supply voltage, the only way to eliminate it completely is by power gating (C6 entry). What makes exploiting core idle time difficult is that it is not simply a matter of power gating the core when it is idle for long periods (not running any jobs). Rather, many important computing applications, especially in the consumer space, have significant idle time during which power gating may be invoked while the application is “running”. This can happen for many reasons, including the interactive or I/O-intensive nature of the applications, insufficient work in the CPU, or insufficient parallelism [BDMF10]. Even in traditional compute intensive application domains, CPU cores that were previously active are now being rendered idle during application execution because of the inclusion of general purpose programmable GPUs [FSB+11] and hardware accelerators [HSA].

Exploiting C6 entry faces 2 challenges. First, transitioning to and from active to C6 sleep state requires time and power to save architectural state and cold-miss penalties of lost micro-architectural state. Second, much of the idle time in modern client applications is composed of short duration idle events.

Figure 2.4 shows the average utilization and the average number of idle events per second across a subset of client applications from the consumer [PCM11] and CPU-GPU space [CBM+09]. The y-axis shows utilization (the percent of time the application is active). The line shows the idle events per second for each benchmark. For optimal power savings, the idle
time of the application should be characterized by very few long idle events. Unfortunately, most applications, regardless of their utilization, have a high frequency of idle events, ranging from an average of 6ms (*Heartwall*) to 125 usec between idle events (*Myocyte*), making it difficult to predict when to enter the C6 state.

Whenever a core idles there is an option to either stay in clock gated state or choose to initiate entry into C6. We examine two specific C6 entry algorithms, fixed preflush filtering (FPF) and autodemotion that represent the state of the art in commercially available hardware.

The FPF algorithm delays C6 entry for a fixed amount of time, referred to as the PreFlush latency, in order to filter out the high frequency, low duration idle events that can be detrimental to performance and energy. This scheme is used in production in the AMD 15h Trinity processor systems [BIO12].

Autodemotion assumes that short duration idle events are clustered and come in *interrupt storms*. Hence Autodemotion either enables or disables C6 entry based on the frequency of idle events. If the number is greater than or equal to a threshold of events/sec for the current time interval, C6 is disabled for the next time interval. Otherwise, C6 is enabled and the system immediately enters C6. The Autodemotion scheme is implemented in the Intel SandyBridge
family of processors [Jia].

Both the FPF and Autodemotion algorithms rely heavily on a threshold – FPF on the preflush latency and Autodemotion on the number of events that constitute a storm. A trace driven simulator was used to collect idleness and activeness traces together with a power modeling framework to perform a design space evaluation of both the schemes.

We evaluate the FPF design space, varying the PreFlush latency from 0-500 usec. Similarly, we varied the Autodemotion threshold from 2000-10,000 events per second. Figure 2.5 shows the results of our evaluation. We show the average power, performance, and energy across twenty seven consumer and CPU-GPU benchmarks. All numbers are provided as overheads.
relative to Oracle gating. Oracle gating is the best possible power gating scheme – it uses precise information on the length of the idle event to power gate (immediately) only when it saves power.

As shown, there is a severe power and performance penalty when the PreFlush latency is low because of the overheads of power gating. Increasing PreFlush latency always improves the performance, as power gating is never a performance win. A high PreFlush latency sacrifices power and energy because we spend more time in the PreFlush state instead of entering C6. The energy minimum for the FPF scheme is at 200 usec.

Figure 2.5 (b) shows the results for the Autodemotion scheme. The threshold of 2000 events per second turns off power gating if two events occur during the last 1 msec interval. This most conservative value preserves the most performance, but suffers a large power penalty as nearly all power gating opportunities are missed. As the threshold increases, the scheme is more aggressive, resulting in performance loss and energy gains. Beyond the energy minimum of 5000 events per second, this scheme begins to allow too many small events to occur, resulting in lost performance and increased energy.

The results of the design space evaluation indicate that on average the best operating points for FPF and Autodemotion schemes are still short of Oracle gating by 8% and 6.2% on average.

In this section, we presented a comparative analysis of existing state-of-the-art power gating entry algorithms on modern consumer and CPU-GPU benchmarks. In the next section, we provide background related to thermal mitigation techniques in computing systems.

### 2.4 Thermal Mitigation

There have been a large number of studies examining thermal mitigation in processors [IBC+06] [DM06] [SA03] [GPV04] [PL07] [ZXD+08] [KVH+14], in servers [TWR+09] [CKS+07] [AIR11] [AR10] [SAH+15a] [SAH+15b], and in the data center [BF07] [SBP+05].
Thermal mitigation could be achieved via (1) power and thermally efficient system design including micro architectural techniques [BCGM07] [SSS+04] [PL07] [KHM01]; (2) power management for energy efficiency [SBA+01] [BBDM00] [BJ12] [AMP+15]; (3) use of efficient packaging and cooling techniques [AAH+18] [AAH+17] including management of cooling systems [CAR+10] [BTS+11] [SAH+15a]; and (4) runtime thermal management including scheduling methods. The research in this thesis is most closely related to the last category of dynamic thermal management at runtime (DTM) techniques. Prior DTM research can be further classified in to four categories.

Voltage and Frequency Scaling. The goal of dynamic voltage and frequency scaling techniques is to control processor overheating by keeping the temperature below a critical threshold [BIO12] [Jia]. Modern processors implement such techniques at finer granularities with the use of temperature estimation or sensor implementations (several thermal entities per chip [PMA+13] or per core DVFS proposals [DM06]), or co-ordinated energy management [PRM+13] [PRM+14].

Resource Throttling. Prior research involves controlling the behavior of the processor when it approaches a critical temperature limit. Proposals include global clock gating by Brooks et al. [BM01], local feedback controlled fetch-toggling by Skadron et al. [SSS+04], and decode-throttling by Sanchez et al. [SKO+97]. These techniques can be implemented with any scheduler as they manage thermals within a single socket.

Workload Profiling and Migration. Srinivasan et al. [SA03] propose off-line workload analysis techniques to decide processor operating frequencies in order to mitigate thermals. Others propose techniques to migrate work reactively [HBA03] [CRWG08] (after reaching a temperature threshold), pro-actively [CRG08] (before reaching a limit) or predictive [YLK08] [CRG09] (by estimating future temperature). Migration is analogous to scheduling and may be useful when job durations are long.

Scheduling and Dynamic Thread Assignment. A variety of scheduling techniques...
have been proposed in the past including techniques that make decisions based on current temperature [MCRS05] [WvLD+09], history [CRWG08], randomized [TGSC06], prevent heat re-circulation [MCRS05] and predictive temperature estimation [YLK08] [AR09]. Heat-and-Run [GPV04] proposes the loading of cores as much as possible by scheduling threads that use orthogonal resources on to an SMT system.

Acknowledgements

Chapter 2 in part, is a reprint of the material as it appears in the proceedings of Sigmetrics 2014. Arora, Manish; Manne, Srilatha; Eckert, Yasuko; Paul, Indrani; Jayasena; Nuwan; Tullsen, Dean, A comparison of core power gating strategies implemented in modern hardware. Proceedings of the 2014 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems. The dissertation author was the primary investigator and author of this paper.
Chapter 3

CPU Design for CPU-GPU Systems

GPU computing has emerged as a viable alternative to CPUs for throughput oriented applications or regions of code. Speedups of $10 \times$ to $100 \times$ over CPU implementations have been reported. This trend is expected to continue in the future with GPU architectural advances, improved programming support, scaling, and tighter CPU-GPU chip integration.

However, not all code will get mapped to the GPUs, even for many of those applications which map well to the GPU – the CPU still runs code that is not targeted to the GPU, and often that code is still very much performance-critical. This work demonstrates that the code that the CPU will be expected to execute in an integrated CPU-GPU environment is profoundly different than the code it has been optimized for over the past many generations. The characteristics of this new code should drive future CPU design and architecture. Specifically, this work shows that post-GPU code tends to have lower ILP, significantly more difficult to predict loads, harder to predict stores, and more difficult branch prediction. Post-GPU code exhibits smaller gains from the availability of multiple cores because of reduced thread level parallelism.
3.1 Introduction

Fueled by high computational throughput and energy efficiency, we have seen the quick adoption of GPUs as general purpose computing engines in recent years. We are seeing heavier integration of the CPU and the GPU, including the GPU appearing on the same die, further decreasing barriers to use of the GPU to offload the CPU. Much effort has been made to adapt GPU designs to anticipate this new partitioning of the computation space, including better programming models, more general processing units with support for control flow, etc. However, little attention has been placed on the CPU and how it needs to adapt to this change.

This work demonstrates that the coming era of CPU and GPU integration requires us to rethink the design and architecture of the CPU. We show that the code the CPU will run, once appropriate computations are mapped to the GPU, has significantly different characteristics than the original code (which previously would have been mapped entirely to the CPU).

Modern GPUs contain hundreds of ALUs, hardware thread management, and access to fast on-chip and high-bandwidth external memories. This translates to peak performance of teraFlops per device [NV109]. There has also been an emergence of new application domains [ABC+06] capable of utilizing this performance. These new applications often distill large amounts of data. GPUs have been architected to exploit application parallelism even in the face of high memory latencies. Reported speedups of 10 - 100× are common, although another study shows speedups over an optimized multicore CPU of 2.5× [LKC+10].

These speedups by no means imply that CPU performance is no longer critical. Many applications do not map at all to GPUs; others map only a portion of their code to the GPU. Examples of the former include applications with irregular control flow and without high data-level parallelism, as exemplified by many of the SPECint applications. Even for applications with data-level parallelism, there are often serial portions that are still more effectively executed by the CPU. Further, GPU programming currently requires considerable programmer effort, and that
effort grows rapidly as the code maps less cleanly to the GPU. As a result, it is most common to only map to the GPU those portions of the code which map easily and cleanly.

Even when a significant portion of the code is mapped to the GPU, the CPU portion will in many cases be performance critical. Consider the case of Kmeans. We study an optimized GPU implementation from the Rodinia \[CBM+09\] benchmark suite. The GPU implementation achieves a speedup of \(5\times\) on kernel code. Initially, about 50% of execution time is non-kernel code, yet because of the GPU acceleration, over \(4/5\) of execution time is spent in the CPU and less than \(1/5\) is spent on the GPU.

Kumar, et al. \[KTJ06\] argue that the most efficient heterogeneous designs for general-purpose computation contain no general-purpose cores (i.e., cores that run everything well), but rather cores that each run a subset of codes well. The GPU already exemplifies that, running some code lightning fast, other code very poorly. As one of the first steps toward core heterogeneity will likely be CPU-GPU integration, the general-purpose CPU need no longer be fully general-purpose. It will be more effective if it becomes specialized to the code that cannot run on the GPU. This research seeks to understand the nature of that code, and begin to identify the direction in which that should push future CPU designs.

When we compare the code running on the CPU before and after CPU integration, we find several profound changes. We see significant decreases in ILP, especially for large window sizes (10.9% drop). We see significant increases in the percentage of “hard” loads (17.2%) and “hard” stores (12.7%). We see a dramatic overall increase in the percentage of “hard” branches, which translates into a large increase in the mispredict rate of a reasonable branch predictor (55.6%). Average thread level parallelism (defined by 32-core speedup), drops from 5.5 to 2.2.
3.2 Background

Initial attempts at using GPUs for general purpose computations used corner cases of the graphics APIs \cite{OLG+05}. Programmers mapped data to the available shader buffer memory and used the graphics-specific pipeline to process data. NVIDIA’s CUDA and AMD’s Brook+ platform added hardware to support general computations and exposed the multi-threaded hardware via a programming interface. With GPU hardware becoming flexible, new programming paradigms like OpenCL emerged. Typically, the programmer is given an abstraction of a separate GPU memory address space similar to CPU memory where data can be allocated and threads launched. While this computing model is closer to traditional computing models, it has several limitations. Programming GPUs still requires architecture-specific optimizations, which impacts performance portability. There is also performance overhead resulting from separate discrete memory used by GPUs.

Recently AMD (Fusion APUs), Intel (Sandy Bridge), and ARM (MALI) have released solutions that integrate general purpose programmable GPUs together with CPUs on the same chip. In this computing model, the CPU and GPU may share memory and a common address space. Such sharing is enabled by the use of an integrated memory controller and coherence network for both the CPU and GPU. This promises to improve performance because no explicit data transfers are required between the CPU and GPU, a feature sometimes known as zero-copy \cite{AMDb}. Further, programming becomes easier because explicit GPU memory management is not required.

3.3 Benchmarks

Over the last few years a large number of CPU applications have been ported to GPUs. Some implementations almost completely map to the GPU while other applications only map
certain kernel codes to the GPU. For this study, we seek to examine a spectrum of applications with varying levels of GPU offloading.

We rely as much as possible on published implementations. This ensures that the mapping between GPU code and CPU code would not be driven by our biases or abilities, but rather by the collective wisdom of the community. We make three exceptions, for particularly important applications (SPEC) where the mapping was clear and straightforward. We perform our own CUDA implementations and used those results for these benchmarks.

We use 3 mechanisms to identify the partitioning of the application between the CPU and GPU. First, if the GPU implementation code was available in the public domain, we study it to identify CPU mapped portions. If the code was not available, we obtain the partitioning information from publications. Lastly we ported the three mentioned benchmarks to the GPU

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Suite</th>
<th>Application Domain</th>
<th>GPU Kernels</th>
<th>Normalized Kernel Speedup (×)</th>
<th>Implementation Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kmeans</td>
<td>Rodinia</td>
<td>Data Mining</td>
<td>2</td>
<td>5.0</td>
<td>Che et al. [CBM*09]</td>
</tr>
<tr>
<td>H264</td>
<td>Spec2006</td>
<td>Multimedia</td>
<td>2</td>
<td>12.1</td>
<td>Hwu et al. [HKR*07]</td>
</tr>
<tr>
<td>SRAD</td>
<td>Rodinia</td>
<td>Image Processing</td>
<td>2</td>
<td>15.0</td>
<td>Che et al. [CBM*09]</td>
</tr>
<tr>
<td>Sphinx3</td>
<td>Spec2006</td>
<td>Speech Recognition</td>
<td>1</td>
<td>17.7</td>
<td>Harish et al. [HBV*11]</td>
</tr>
<tr>
<td>Particlefilter</td>
<td>Rodinia</td>
<td>Image Processing</td>
<td>2</td>
<td>32.0</td>
<td>Goomrum et al. [GTA*10]</td>
</tr>
<tr>
<td>Blackscholes</td>
<td>Parsec</td>
<td>Financial Modeling</td>
<td>1</td>
<td>13.7</td>
<td>Kolb et al. [KP]</td>
</tr>
<tr>
<td>Swim</td>
<td>Spec2000</td>
<td>Water Modeling</td>
<td>3</td>
<td>25.3</td>
<td>Wang et al. [WFR09]</td>
</tr>
<tr>
<td>Milc</td>
<td>Spec2006</td>
<td>Physics</td>
<td>18</td>
<td>6.0</td>
<td>Shi et al. [SGK*09]</td>
</tr>
<tr>
<td>Hmmer</td>
<td>Spec2006</td>
<td>Biology</td>
<td>1</td>
<td>19.0</td>
<td>Walters et al. [WBKC09]</td>
</tr>
<tr>
<td>LUD</td>
<td>Rodinia</td>
<td>Numerical Analysis</td>
<td>1</td>
<td>13.5</td>
<td>Che et al. [CBM*09]</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>Parsec</td>
<td>Physics</td>
<td>1</td>
<td>26.0</td>
<td>Che et al. [CBM*09]</td>
</tr>
<tr>
<td>Bwaves</td>
<td>Spec2006</td>
<td>Fluid Dynamics</td>
<td>3</td>
<td>18.0</td>
<td>Ruetsche et al. [RF]</td>
</tr>
<tr>
<td>Equake</td>
<td>Spec2000</td>
<td>Wave Propagation</td>
<td>2</td>
<td>5.3</td>
<td>Own implementation</td>
</tr>
<tr>
<td>Libquantum</td>
<td>Spec2006</td>
<td>Physics</td>
<td>4</td>
<td>28.1</td>
<td>Gutierrez et al. [GRTZ07]</td>
</tr>
<tr>
<td>Ammp</td>
<td>Spec2000</td>
<td>Molecular dynamics</td>
<td>1</td>
<td>6.8</td>
<td>Own implementation</td>
</tr>
<tr>
<td>CFD</td>
<td>Rodinia</td>
<td>Fluid Dynamics</td>
<td>5</td>
<td>5.5</td>
<td>Solano-Q et al. [SQWBS11]</td>
</tr>
<tr>
<td>Mgrid</td>
<td>Spec2000</td>
<td>Grid Solver</td>
<td>4</td>
<td>34.3</td>
<td>Wang et al. [WFR09]</td>
</tr>
<tr>
<td>LBM</td>
<td>Spec2006</td>
<td>Fluid Dynamics</td>
<td>1</td>
<td>31.0</td>
<td>Stratton et al. [Joh]</td>
</tr>
<tr>
<td>Leukocyte</td>
<td>Rodinia</td>
<td>Medical Imaging</td>
<td>3</td>
<td>70.0</td>
<td>Che et al. [CBM*09]</td>
</tr>
<tr>
<td>ART</td>
<td>Spec2000</td>
<td>Image Processing</td>
<td>3</td>
<td>6.8</td>
<td>Own implementation</td>
</tr>
<tr>
<td>Heartwall</td>
<td>Rodinia</td>
<td>Medical Imaging</td>
<td>6</td>
<td>7.9</td>
<td>Szafaryn et al. [SSS09]</td>
</tr>
<tr>
<td>Fluidanimate</td>
<td>Parsec</td>
<td>Fluid Dynamics</td>
<td>6</td>
<td>3.9</td>
<td>Sinclair et al. [SDS11]</td>
</tr>
</tbody>
</table>

Table 3.1: CPU-GPU Benchmark Description.
ourselves. Table 3.1 summarizes the characteristics of our benchmarks. The table lists the GPU mapped portions, and provides statistics such as GPU kernel speedup. The kernel speedups reported in the table are from various public domain sources, or our own GPU implementations. Since different publications tend to use different processor baselines and/or different GPUs, we normalized numbers to a single core AMD Shanghai processor running at 2.5GHz and NVIDIA GTX 280 GPU with 1.3GHz shader frequency. We used published SPECrate numbers and linear scaling of GPU performance with number of SMs/frequency to perform the normalization.

We also measure and collect statistics for pure CPU benchmarks, benchmarks with no publicly known GPU implementation. These, combined with the previously mentioned benchmarks, give us a total of 11 CPU-Only benchmarks, 11 GPU-Heavy benchmarks, and 11 Mixed applications where some, but not all, of the application is mapped to the GPU. We do not show the CPU-Only benchmarks in Table 3.1 because no CPU-GPU mapping was done.

3.4 Experimental Methodology

This section describes our infrastructure and simulation parameters. Our goal is to identify fundamental characteristics of the code, rather than the effects of particular architectures. This means, when possible, measuring inherent ILP and characterizing loads, stores, and branches into types, rather than always measuring particular hit rates, etc. We do not account for code that might run on the CPU to manage data movement, for example – this code is highly architecture specific, and more importantly, expected to go away in coming designs. We simulate complete programs whenever possible.

While all original application source code was available, we were limited by the non-availability of parallel GPU implementation source code for several important benchmarks. Hence, we use the published CPU-GPU partitioning information and kernel speedup information to drive our analysis.
We develop a PIN based measurement infrastructure. Using the CPU/GPU partitioning information from each benchmark, we modify the original benchmark code without any modifications for GPU implementation. We insert markers indicating the start and end of GPU code, allowing our microarchitectural simulators built on top of PIN to selectively measure CPU and GPU code characteristics. All benchmarks are simulated for the largest available input sizes. Programs were run to completion or for at least 1 trillion instructions.

CPU Time is calculated by using the following steps. First, the proportion of application time that gets mapped to the GPU/CPU is calculated. This is done by inserting time measurement routines in marker functions and running the application on the CPU. Next, we use the normalized speedups to estimate the CPU time with the GPU. For example, consider an application with 80% of execution time mapped to the GPU and a normalized kernel speedup of 40×. Originally, just 20% of the execution time is spent on the CPU. However, post-GPU, \( \frac{20}{20 + 80/40} \times 100\% \) or about 91% of time is spent executing on the CPU. Time with conservative speedups was obtained by capping the maximum possible GPU speedup value to 10.0. A value of 10.0 was used as a conservative single-core speedup cap [LKC+10]. Hence, for the prior example, post-GPU with conservative speedups \( \frac{20}{20 + 80/10} \times 100\% \) or about 71% of time is spent executing on the CPU.

We categorize loads and stores into four categories, based on measurements on each of the address streams. Those categories are static (address is a constant), strided (predicted with 95% accuracy by a stride predictor that is able to track up to 16 strides per PC), patterned (predicted with 95% accuracy by a large Markov predictor with 8192 entries, 256 previous addresses, and 8 next addresses), and hard (all other loads or stores).

Similarly, we categorize branches as biased (95% taken or not taken), patterned (95% predicted by a large local predictor, using 14 bits of branch history), correlated (95% predicted by a large gshare predictor, using 17 bits of global history), and hard (all other branches). To measure branch mispredict rates, we construct a tournament predictor out of the mentioned gshare
and local predictors, combined through a large chooser.

We use Microarchitecture Independent Workload Characterization (MICA) \cite{HE07} to obtain instruction level parallelism information. MICA calculates the perfect ILP by assuming perfect branch prediction and caches. Only true dependencies affect the ILP. We modify the MICA code to support instruction windows up to 512 entries.

We use a simple definition of thread-level parallelism, based on real machine measurements, and exploiting parallel implementations available for Rodinia, Parsec, and some Spec2000 (those in SpecOMP 2001) benchmarks. Again restricting our measurements to the CPU code marked out in our applications, we define thread level parallelism as the speedup we get on an AMD Shanghai quad core \( \times 8 \) socket machine. The TLP results, then, cover a subset of our applications for which we have credible parallel CPU implementations.

## 3.5 Results

In this section we examine the characteristics of code executed by the CPU, both without and with GPU integration. For all of our presented results, we partition applications into three groups — those where no attempt has been made to map code to the GPU (CPU-Only), those where the partitioning is a bit more evenly divided (Mixed), and those where nearly all the code is mapped to the GPU (GPU-Heavy). We first look at CPU time — what portion of the original execution time still gets mapped to the CPU, and what is the expected CPU time spent running that code. We then go on to examine other dimensions of that code that still runs on the CPU.

### 3.5.1 CPU Execution Time

We start by measuring time spent on the CPU. To identify the utilization and performance-criticality of the CPU after GPU offloading, we calculate the percentage of time in CPU execution after the GPU mapping takes place. We use, initially, the reported speedups from the literature
for each GPU mapping.

The first bar in Figure 3.1 is the percentage of the original code that gets mapped to the CPU. The other two bars represent time actually spent on the CPU (as a fraction of total run time), assuming that the CPU and GPU run separately (if they execute in parallel, CPU time increases further). Thus, the second and third bars account for the reported speedup expected on the GPU. The only difference is the third bar assumes GPU speedup is capped at $10 \times$. For the mixed set of applications in the middle, even though 80% of the code on average is mapped to the GPU, the CPU is still the bottleneck. Even for the GPU-heavy set on the right, the CPU is executing 7-14% of the time. Overall, the CPU is still executing more often than the GPU and remains highly performance-critical. The benchmarks are sorted by CPU time – we’ll retain this ordering for subsequent graphs.

In future graphs, we will use the conservative cpu time (third bar) to weight our average (after GPU integration) results – e.g., if you were to run sphinx3 and hmmer in equal measure, the CPU would be executing sphinx3 code about twice as often as hmmer code after CPU-GPU integration.
Figure 3.2: Instruction level parallelism with and without GPU. Not all bars appear in the CPU-Only applications, since they do not vary post-GPU. This is repeated in future plots.

3.5.2 ILP

ILP captures the inherent parallelism in the instruction stream – it can be thought of as measuring (the inverse of) the dependence critical path through the code. For out-of-order processors, ILP is heavily dependent on window size – the number of instructions the processor can examine at once looking for possible parallelism.

As seen in Figure 3.2 in 17 of the 22 applications, ILP drops noticeably, particularly for large window sizes. For swim, milc, cfd, mgrid and fluidanimate, it drops by almost half. Between the outliers (ILP actually increases in 5 cases), and the damping impact of the non-GPU applications, the overall effect is a 10.9% drop in ILP for larger window sizes and a 4% drop for current generation window sizes. For the mixed applications, the result is much more striking, a 27.5% drop in ILP for the remaining CPU code. In particular, we are seeing that potential performance gains from large windows is significantly degraded in the absence of the GPU code.

In the common case, independent loops are being mapped to the GPU. Less regular code, and loops with loop-carried dependencies restricting parallelism are left on the CPU. This is the
case with h264 and milc, for example; key, tight loops with no critical loop-carried dependencies are mapped to the GPU, leaving less regular and more dependence-heavy code on the CPU.

![Figure 3.3: Distribution of branch types with and without GPU.](image)

### 3.5.3 Branch Results

We classify static branches into four categories. The categories are biased (nearly always taken or not taken), patterned (easily captured by a local predictor), correlated (easily captured by a correlated predictor), or hard (none of the above). Figure 3.3 plots the distribution of branches found in our benchmarks.

Overall, we see a significant increase in hard branches. In fact, the frequency of hard branches increases by 65% (from 11.3% to 18.6%). The increase in hard branches in the overall average is the result of two factors – the high concentration of branches in the CPU-only workloads (which more heavily influence the average) and the marked increase in hard branches in the Mixed benchmarks. The hard branches are primarily replacing the reduced patterned branches, as the easy (biased) branches are only reduced by a small amount.

Some of the same effects discussed in the previous section apply here. Small loops with
The outliers (contrary results) in this case are instructive, however. equake and cfd map data-intensive loops to the GPU. That includes data-dependent branches, which in the worst case can be completely unpredictable.

Even with individual branches getting hard to predict, it is not clear prediction gets worse, as it is possible that with fewer static branches being predicted, aliasing would be decreased. However, experiments on a realistic branch predictor confirmed that the new CPU code indeed
stresses the predictor heavily. We found the frequency of mispredicts, for the modeled predictor, to increases dramatically, by 56% (from 2.7 to 4.2 misses per kilo instructions). The graph is not shown here to conserve space. The increase in misses per instruction is primarily a reflection of the increased overall mispredict rate, as the frequency of branches per instruction actually changes by less than 1% between the pre-GPU and post-GPU code.

These results indicate that a branch predictor tuned for generic CPU code may in fact not be sufficient for post-GPU execution.

### 3.5.4 Load and Store Results

Typically, code maps to the GPU most effectively when the memory access patterns are regular and ordered. This means we would expect to see a significant drop in ordered (easy) accesses for the CPU.

Figure 3.4(a) shows the classification of CPU loads. What is graphed in this figure is the breakdown of loads as a percentage of all non-static loads. That is, we have already taken out those loads that will be trivially handled by the cache. In this figure, we see a sharp increase in hard loads, which is perhaps more accurately characterized as a sharp decrease in strided loads. Thus, of the non-trivial loads that remain, a much higher percentage of them are not easily handled by existing hardware prefetchers or inline software prefetching. The percentage of strided loads is almost halved, both overall and for the mixed workloads. Patterned loads are largely unaffected, and hard loads increase very significantly, to the point where they are the dominant type. Some applications (e.g., lud and hmmer) go from being almost completely strided, to the point where a strided prefetcher is useless.

*kmeans, srad and milc* each show a sharp increase in the number of hard loads. We find that the key kernel of *kmeans* generates highly regular, strided loads. This kernel is offloaded to the GPU. *srad* and *milc* are similar.

Though the general trend shows an increase in hard loads, we see a notable exception in
bwaves, in which an important kernel with highly irregular loads is successfully mapped to the GPU.

Figure 3.4(b) shows that these same trends are also exhibited by the store instructions, as again the strided stores are being reduced, and hard stores increase markedly. Interestingly, the source of the shift is different. In this case, we do not see a marked decrease in the amount of easy stores in our CPU-GPU workloads. However, the high occurrence of hard stores in our CPU-Only benchmarks results in a large increase in hard stores overall.

Similar to the loads, we see that many benchmarks have kernels with strided stores which go to the GPU. This is the case with swim and hmmer. On the other hand, in bwaves and equake, the code that gets mapped to the GPU does irregular writes to an unstructured grid.

![Figure 3.5: Frequency of vector instructions with and without GPU.](image)

3.5.5 Vector Instructions

We were also interested in the distribution of instructions, and how it changes post-GPU. Somewhat surprisingly, we find little change in the mix of integer and floating point operations. However, we find that the usage of SSE instructions drops significantly, as shown in figure 3.5.
We see an overall reduction of 44.3% in the usage of SSE instructions (from 15.0% to 8.5%). This is an expected result, as the SSE ISA enhancements target in many cases the exact same code regions as the general-purpose GPU enhancements. For example, in \textit{kmeans} the find\_nearest\_point functions heavily utilizes MMX instructions, but this function gets mapped to the GPU.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{thread-level-parallelism.png}
\caption{Thread level parallelism with and without GPU}
\end{figure}

### 3.5.6 Thread Level Parallelism (TLP)

TLP captures parallelism that can be exploited by multiple cores or thread contexts, enabling us to measure the utility of having an increasing number of CPU cores. Figure 3.6 shows measured TLP results for our benchmarks.

Let us first consider the GPU-heavy benchmarks. CPU implementations of the benchmarks show abundant TLP. We see an average speedup of 14.0× for 32 cores. However, post-GPU the TLP drops considerably, yielding only a speedup of 2.1×. Five of the benchmarks exhibit no TLP post-GPU, in contrast, five benchmarks originally had speedups greater than 15×. Perhaps the most striking result (also true for the mixed benchmarks) – no benchmark’s post-GPU code sees any significant gain going from 8 cores to 32.
Overall for the mixed benchmarks, we again see a considerable reduction in post-GPU TLP; it drops by almost 50% for 8 cores and about 65% for 32 cores. CPU-Only benchmarks exhibit lower TLP than both the Mixed and GPU-Heavy sets, but do not lose any of that TLP because no code runs on the GPU. Overall, we see that applications with abundant TLP are good GPU targets. In essence, both multicore CPUs and GPUs are targeting the same parallelism. However, as we have seen, post-GPU parallelism drops significantly.

On average, we see a striking reduction in exploitable TLP, 8-core TLP dropped by 43% from 3.5 to 2.0 and 32-core TLP dropped by 60% from 5.5 to 2.2. While going from 8 cores to 32 cores yields a nearly two fold increases in TLP in the original code, post-GPU the TLP grows by just 10% over that region – extra cores are nearly useless.

### 3.6 Impact on CPU Design

Good architectural design is tuned for the instruction execution stream that is expected to run on the processor. This work indicates that, for those general purpose CPUs, the definition of “typical” code is in the process of changing. This work is the first attempt to isolate and characterize the code the CPU will now be executing. This section identifies some architectural implications of the changing code base.

**Sensitivity to Window Size.** It has long been understood that out-of-order processors benefit from large instruction windows. As a result, much research has sought to increase window size, or create the illusion of large windows [CSMV04]. While we do not see evidence that large windows are not useful, the incremental gains may be more modest.

**Branch Predictors.** We show that post-GPU code dramatically increases pressure on the branch predictor. This is despite the fact that the predictor is servicing significantly fewer static branches. Recent trends targeting very difficult branches using complex hardware and extremely long
histories [Sez07] seem to be a promising direction because they better attack fewer, harder branches.

**Load and Store Prefetching.** Memory access will continue to be perhaps the biggest performance challenge of future processors. Our results touch particularly on the design of future prefetchers, which currently have a heavy influence on CPU and memory performance. Stride-based prefetchers are commonplace on modern architectures, but are likely to become significantly less relevant on the CPU. What is left for the CPU are very hard memory accesses. Thus, we expect the existing hardware prefetchers to struggle.

We actually have fewer static loads and stores that the CPU must deal with, but those addresses are now hard to predict. This motivates an approach that devotes significant resources toward accurate prediction of a few problematic loads/stores. Several past approaches had exactly this flavor, but have not yet had a big impact on commercial designs. These include Markov-based predictors [JG97] which target patterned accesses but can capture very complex patterns, and predictors targeted at pointer-chain computation [CJG02, CSCT02]. These types of solutions should be pursued with new urgency. We have also seen significant research into helper-thread prefetchers which have impacted some compilers and some hardware, but their adoption is still not widespread.

**Vector Instructions.** SSE instructions have not been rendered unnecessary, but certainly less important. Typical SSE code can be executed faster and at lower power on GPUs. Elimination of SSE support may be unjustified, but every core need not support it. In a heterogeneous design, some cores could drop support for SSE, or even in a homogeneous design, multiple cores could share hardware.

**Thread Level Parallelism.** Heterogeneous architectures are most effective when diversity is high [KTJ06]. Thus, recent trends in which CPU and GPU designs are converging more than
diverging are suboptimal. One example is that both are headed to higher and higher core and thread counts. Our results indicate that the CPU will do better by addressing codes that have low parallelism and irregular code, and seeking to maximize single-thread, or few-thread, throughput.

### 3.7 Conclusion

As GPUs become more heavily integrated into the processor, they will inherit computations that have been traditionally executed on the CPU. As a result, the nature of the computation that remains on the CPU will change. This research looks at the changing workload of the CPU as we progress towards higher CPU-GPU integration. This changing workload impacts the way that future CPUs should be designed and architected.

This research shows that even when significant portions of the original code are offloaded to the GPU, the CPU is still frequently performance critical. It further shows that the code the CPU is running is different than before GPU offloading along several dimensions. ILP becomes harder to find. Loads become significantly more difficult to prefetch. Store addresses become more difficult as well. Post-GPU code places significantly higher pressure on the branch predictor. We also see a decrease in the importance of vector instructions and the ability to exploit multiple cores. Hence the coming era of CPU-GPU integration requires us to rethink CPU design and architecture.

**Acknowledgements**

Chapter 3 in full, is a reprint of the material as in it appears in IEEE Micro. Arora, Manish; Nath, Siddhartha; Subhra Mazumdar; Baden, Scott; Tullsen, Dean, Redefining the Role of the CPU in the Era of CPU-GPU Integration, IEEE Micro, December, 2012. The dissertation author was the primary investigator and author of this paper.
Chapter 4

CPU Power Gating Techniques for CPU-GPU Systems

Overall energy consumption in modern computing systems is significantly impacted by idle power. Power gating, also known as C6, is an effective mechanism to reduce idle power. However, C6 entry incurs non-trivial overheads and can cause negative savings if the idle duration is short. As CPUs become tightly integrated with GPUs and other accelerators, the incidence of short duration idle events are becoming increasingly common. Even when idle durations are long, it may still not be beneficial to power gate because of the overheads of cache flushing, especially with FinFET transistors.

This research presents a comprehensive analysis of idleness behavior of modern CPU workloads, consisting of both consumer and CPU-GPU benchmarks. It proposes techniques to accurately predict idle durations and develops power gating mechanisms that account for dynamic variations in the break-even point caused by varying cache dirtiness. Accounting for variations in the break-even point is even more important for FinFET transistors. In systems with FinFET transistors, the proposed mechanisms provide average energy reduction exceeding 8% and up to 36% over three currently employed schemes.
4.1 Introduction

Dynamic and leakage power reduction is a first-order design goal of modern processor architects. Dynamic power is lowered by reducing the work required for accomplishing a task [MKG98], or by utilizing dynamic voltage and frequency scaling (DVFS) to run the task at lower frequency and voltage [ACP11]. Leakage power, however, is not related to processor activity but is more closely tied to the design choices required to achieve high frequency operation with smaller devices and lower maximum supply voltages [BMMF02]. Leakage power scales with supply voltage, but reducing the supply voltage only gets you so far; the only way to eliminate it completely is by removing or gating the power supply, referred to as power gating. It is also referred to as entering core-C6 or C6.

Past work has addressed the problem of reducing leakage power during active times by power gating cores [FAB+11] [FSB+11] [Whi11], caches [RKQK12], and components within the core [HBS+04] [LMT+09]. However, as [FAB+11] [AME+14] and [Nus12] show, leakage power and global clock power is also a significant issue when the processor is idle or halted. In the AMD™ 15h family of processors, for example, the core power drops significantly when the machine is halted because the pipeline is idle and the local clocks are gated. However, global clocks and leakage power from caches and logic still dissipate 35% of maximum power. The core power diminishes to near zero only when the module is power gated with the core and L2 cache entering C6 [FAB+11].

Fully exploiting core idle time requires more than just power gating the core when it is not running any jobs. Rather, a large number of important consumer computing applications have significant idle time while the application is “running.” Such idle time is characterized by many short idle events. Idleness while running applications happens for several reasons, such as the interactive or I/O-intensive nature of the applications or insufficient parallelism to keep all cores busy [FURM00] [BDMF10]. The inclusion of general purpose
programmable GPUs [FSB+11] [Whi11] and hardware accelerators [ANM+12] [CMHM10] [HZH+11] [CGG+12] [HSA] along with new architectural directions (e.g., addressing the dark silicon challenge [EBSA+11] [GHSV+11] [ASG+11] [SAGH+11]) and software directions [KST10] are expected to further increase the appearance of rapid idleness even in traditional compute intensive application domains.

To achieve energy proportional computing [BH07], the overall power of the system should be proportional to the work accomplished. This is only possible if the system dissipates little power when idle. However, there are three reasons why it may be difficult to reduce or eliminate idle power. First, transitioning between active and C6 requires time and energy to move state into and out of cores/caches and ramp voltage supply. Second, much of the idle time in modern applications is composed of short duration idle events, where it is not beneficial to enter C6. Third, the amount of state that needs to be moved varies – depending on workload and how long the core was active before becoming idle.

Existing power-gating techniques assume the existence of a break-even point – if the core will be idle longer than that break-even time, it should be gated, otherwise it should not. This work demonstrates that no such static point exists, and in fact the break-even point is dynamic, different for every idle event. This is because the cost of gating is heavily influenced by the time to flush the cache, which varies according to the number of dirty lines.

Fig. 4.1(a) shows the average number of idle events per second (line) and average percentage of the cache that is dirty when the core becomes idle (bar), across a subset of consumer and CPU-GPU applications discussed in Section 4.2. For optimal power savings, the idle time of the application should be characterized by very few idle events. Unfortunately, most applications have a high frequency of idle events, ranging from an average of about 110 events per sec (or 9ms between idle event starts) (VideoScaling) to 3,000 events per sec (or about 300 us between idle events) (StartApps). Also, applications differ significantly in the amount of cache that is dirty on idle, ranging from nearly 0% (ColorFill) to as much as 60% (WebBrowser). This makes it
Figure 4.1: (a) Average number of idle events per sec (line) and average percentage of cache dirty on idle (bar). (b) Impact of idle management policies on average power (modeled) for 16nm FinFET.

difficult to predict when it is advantageous to enter the C6 state.

Fig. 4.1(b) shows the power associated with each workload using different idle power management schemes. All bars show the average power for the application for FinFET transistors at 16nm, broken down into power while active (Active) and power while idle. The idle power is further segmented into power when the core is idle but has not entered C6 (Clock Gated) and the power associated with the core entering and exiting C6 (C6 Overheads). Results are shown for never entering C6 (Never), always entering C6 (Always), a perfect duration predictor for entering
C6 (*Duration Oracle*) that assumes an exact knowledge of idle durations but cache dirtiness as the average of actual dirtiness across all benchmarks, and a perfect duration and dirtiness predictor (*Duration & Dirtiness Oracle*) that has precise knowledge of both durations and dirtiness.

The results demonstrate that there can be significant benefit from entering C6. However, if not done carefully, C6 overheads may completely erase the gains from power gating (*StartApps, WebBrowser*). This could result in a loss of both power reduction and performance [LBBS09] [BJ12] [MBBA11] [NRM+06]. Also, even perfect knowledge of the durations of idle events is not enough (*ColorFill, Particles*), unless combined with the knowledge of cache dirtiness.

The era of increasing idleness in CPU cores presents a set of new challenges to the research and industrial design community. Traditional architectural research tends to focus on applications (SPEC®, PARSEC, etc.) with unrealistically few idle periods. These are crafted, in large part, to evaluate the active behavior of the CPU; in that context, CPU idleness is uninteresting. The modeling infrastructure often artificially removes I/O latencies, cannot typically account for accelerators such as GPUs, and simulates time frames too short to capture interesting idle behavior. This chapter addresses these challenges, allowing us to accurately model idle behavior and its impact on power and performance. Specifically, this research:

1. Examines the idleness and cache dirtiness characteristics of current and emerging applications, especially in the realm of CPU/GPU heterogeneous applications;

2. Clearly delineates the overheads associated with power gating and develops a power and performance model to characterize the costs and benefits of power gating;

3. Evaluates power gating methodologies in the context of current technology as well as with future FinFET transistors, where the leakage and dynamic power trade-offs deviate significantly vs. existing process technologies; and

4. Using the model, revisits state-of-the-art idle power management schemes and develops new algorithms that not only reduce energy, but also significantly reduce the incidence of large
outliers.

The rest of the chapter is organized as follows. Section 4.2 presents an idleness and cache dirtiness analysis of current consumer and emerging heterogeneous applications. Section 4.3 describes the process used for power gating cores, along with measurements of the time and power associated with each step in the process. In addition, it presents and verifies the modeling infrastructure developed for analyzing idle power management algorithms. Section 4.4 describes existing idle power management algorithms, their key shortcomings, and describes our proposed enhancements. Section 4.5 presents results and Section 4.6 concludes the chapter.

**Table 4.1**: Benchmark classification, active power (modeled), events per module, and power for different idle management policies (modeled) using 28nm planar technology. Oracle assumes perfect knowledge of both idle durations and dirtiness.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Type</th>
<th>Active (W)</th>
<th>Events / Sec</th>
<th>Util (%)</th>
<th>Never (W)</th>
<th>Always (W)</th>
<th>Oracle (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColorFill</td>
<td>Graphics color filling</td>
<td>Graphics [PCM11]</td>
<td>0.3</td>
<td>1001</td>
<td>0.4</td>
<td>27.1</td>
<td>10.7</td>
<td>10.0</td>
</tr>
<tr>
<td>TextureFill</td>
<td>Graphics texture filling</td>
<td>Graphics [PCM11]</td>
<td>0.5</td>
<td>1039</td>
<td>0.9</td>
<td>27.2</td>
<td>11.1</td>
<td>10.2</td>
</tr>
<tr>
<td>HeartWall</td>
<td>Medical imaging</td>
<td>CPU-GPU [CBM+09]</td>
<td>1.6</td>
<td>168</td>
<td>2.7</td>
<td>26.4</td>
<td>6.8</td>
<td>6.2</td>
</tr>
<tr>
<td>Particles</td>
<td>DirectX® GPU test</td>
<td>Graphics [PCM11]</td>
<td>3.2</td>
<td>1131</td>
<td>5.4</td>
<td>28.9</td>
<td>14.1</td>
<td>13.1</td>
</tr>
<tr>
<td>PicStore</td>
<td>Importing pictures</td>
<td>Storage [PCM11]</td>
<td>10.1</td>
<td>819</td>
<td>16.9</td>
<td>32.4</td>
<td>18.5</td>
<td>17.9</td>
</tr>
<tr>
<td>StartApps</td>
<td>Starting applications</td>
<td>Storage [PCM11]</td>
<td>10.1</td>
<td>2977</td>
<td>17.0</td>
<td>34.3</td>
<td>32.6</td>
<td>24.3</td>
</tr>
<tr>
<td>LavaMD</td>
<td>Molecular dynamics</td>
<td>CPU-GPU [CBM+09]</td>
<td>13.4</td>
<td>1239</td>
<td>22.5</td>
<td>33.6</td>
<td>24.8</td>
<td>18.4</td>
</tr>
<tr>
<td>StartGame</td>
<td>PC game storage</td>
<td>Storage [PCM11]</td>
<td>16.1</td>
<td>1611</td>
<td>27.1</td>
<td>36.5</td>
<td>29.1</td>
<td>25.7</td>
</tr>
<tr>
<td>KMeans</td>
<td>Clustering</td>
<td>CPU-GPU [CBM+09]</td>
<td>20.1</td>
<td>261</td>
<td>34.2</td>
<td>37.0</td>
<td>24.3</td>
<td>23.6</td>
</tr>
<tr>
<td>Pathfinder</td>
<td>Dynamic programming</td>
<td>CPU-GPU [CBM+09]</td>
<td>21.6</td>
<td>2266</td>
<td>36.6</td>
<td>38.5</td>
<td>39.1</td>
<td>27.0</td>
</tr>
<tr>
<td>Backprop</td>
<td>Pattern recognition</td>
<td>CPU-GPU [CBM+09]</td>
<td>23.0</td>
<td>5561</td>
<td>38.8</td>
<td>40.1</td>
<td>61.3</td>
<td>30.6</td>
</tr>
<tr>
<td>LUD</td>
<td>Dense linear algebra</td>
<td>CPU-GPU [CBM+09]</td>
<td>23.2</td>
<td>1041</td>
<td>39.4</td>
<td>39.0</td>
<td>32.4</td>
<td>27.3</td>
</tr>
<tr>
<td>DirectX9</td>
<td>DirectX 9 graphics test</td>
<td>Graphics [PCM11]</td>
<td>23.4</td>
<td>1988</td>
<td>39.8</td>
<td>40.8</td>
<td>39.2</td>
<td>32.8</td>
</tr>
<tr>
<td>Leukocyte</td>
<td>Medical imaging</td>
<td>CPU-GPU [CBM+09]</td>
<td>24.3</td>
<td>3821</td>
<td>41.2</td>
<td>40.6</td>
<td>51.7</td>
<td>31.2</td>
</tr>
<tr>
<td>NN</td>
<td>Data mining</td>
<td>CPU-GPU [CBM-09]</td>
<td>27.6</td>
<td>3048</td>
<td>46.4</td>
<td>42.2</td>
<td>49.2</td>
<td>33.4</td>
</tr>
<tr>
<td>Gaussian</td>
<td>Dense linear algebra</td>
<td>CPU-GPU [CBM-09]</td>
<td>29.8</td>
<td>1146</td>
<td>50.7</td>
<td>42.7</td>
<td>39.2</td>
<td>33.2</td>
</tr>
<tr>
<td>TextEditor</td>
<td>Interactive text editing</td>
<td>Interactive [PCM11]</td>
<td>42.6</td>
<td>2281</td>
<td>60.1</td>
<td>54.8</td>
<td>59.1</td>
<td>50.8</td>
</tr>
<tr>
<td>WebDecrypt</td>
<td>Browsing &amp; decryption</td>
<td>Interactive [PCM11]</td>
<td>61.8</td>
<td>416</td>
<td>87.0</td>
<td>65.3</td>
<td>64.7</td>
<td>62.9</td>
</tr>
<tr>
<td>VideoScaling</td>
<td>Video downscaling</td>
<td>Media [PCM11]</td>
<td>62.7</td>
<td>107</td>
<td>88.2</td>
<td>65.8</td>
<td>63.9</td>
<td>63.6</td>
</tr>
<tr>
<td>WebBrowser</td>
<td>Multi-tabbed Chrome</td>
<td>Interactive [PCM11]</td>
<td>67.4</td>
<td>530</td>
<td>94.9</td>
<td>68.9</td>
<td>70.2</td>
<td>68.2</td>
</tr>
</tbody>
</table>
4.2 Understanding Application Idle Behavior

Much of the research on power reduction and management has focused on power during active computation. However, recent work has shown that idle behavior and idle power management is as or more important in systems ranging from single processor clients [Nus12] [FAB+11] [Jia] [PHAY15] to large scale data-centers [BH07]. Managing idle power for cores is straightforward if the idleness duration and cache dirtiness behavior is predictable or if the idle periods are always long enough to compensate for the power and performance cost of power gating. However, neither of these conditions hold in many current and emerging applications. In this section, we present a set of important consumer and CPU-GPU benchmarks, and examine their behavior to determine the viability of different C6 strategies.

4.2.1 Applications

If we are going to model and evaluate techniques for idle-period power reduction, the first thing we need to do is identify workloads that have realistic idle behavior. For this study, we composed a suite of 20 benchmarks (see Table 4.1) derived from (1) a classic consumer application benchmarking suite (PCMark® [PCM11]) popularly used in industry; and (2) heterogeneous CPU-GPU applications (Rodinia [CBM+09]). Many of these applications are common to the consumer domain such as graphics applications (ColorFill, DirectX9 etc.), storage applications (PicStore, StartApps etc.), and interactive (TextEditor, WebBrowser). However, others are targeted at heterogeneous processors such as AMD APUs [Nus12].

Heterogeneous processors with frameworks such as the heterogeneous system architecture (HSA) [HSA] offer opportunities for tightly-coupled interaction and easy compute offloading between the CPU, GPU, and other accelerators. Hence, we expect to see the CPU sharing computation tasks with other processors and accelerators in the future [HZH+11] [CGG+12], leading to finer-grained interaction between compute entities on the chip. This will produce more
idle time and a higher frequency of idle events on the CPU.

The *Events* column in Table 4.1 shows the number of idle events per second per CPU module. There are a total of two CPU modules for the chip under study as described in Section 4.3. An idle event is defined as the transition from active execution to idle. The *Utilization* column shows the proportion of active to total time for each application. The higher the utilization, the more active the application and higher the power attributed to active execution (*Active* column). The table is sorted by increasing utilization, and it ranges from less than 1% to greater than 94%.

The table also shows the total power associated with each application when C6 entry is never performed (*Never*), always performed (*Always*) and only performed when beneficial using precise knowledge of durations and cache dirtiness (*Oracle*). These values are derived using the modeling infrastructure described in Section 4.3. In general, the lower the utilization, the more opportunity for idle power savings, and the more significant the relative power difference between never entering C6 and always or perfectly entering C6. Generalities, however, do not always hold. For instance, both *ColorFill* and *TextureFill* have very low utilization. However, even with a perfect predictor, the resulting power is still over 10W, with active state power contributing less than 1W in both cases. For *Heartwall*, however, the power can be brought down to less than 7W even without the Oracle predictor. Similarly, the higher the number of events, the more power but *Leukocyte* has almost twice the number of events and higher utilization than *DirectX9*, but still consumes less power with the Oracle predictor. The relationship between idle time, idle event frequency, and power savings is complicated, and the next section examines some of the characteristics that dictate when and by how much applications benefit from C6.

### 4.2.2 Idle Event Distribution

Table 4.1 shows the frequency of events and an estimate of idle time (100%-*Utilization*), but it does not provide any insights into how the events are distributed. For instance, *Backprop*, contains many idle events, and we can calculate the average duration, but the potential for gating
will depend more on the distribution of idle event times than the average value. This section examines some of those distributions more closely.

Fig. 4.2 shows the distribution of idle events of our applications collected using the AMD A8-5600K APU and the Xperf tool set as described in Section 4.3. The y-axis shows the cumulative distribution function (CDF) of idle events, as a function of the idle event duration, and the x-axis represents the event duration (in μsec). The figure shows several interesting trends. For example, a large number of applications have a majority of their events as short idle events of durations less than 100μsec. Graphics applications such as ColorFill regularly interrupt the CPU at 1msec granularity because of the use of multimedia timers in Windows® to maintain system responsiveness. Lastly, we observe the OS timer tick inducing activity every 15.6msec for some of the applications.

Power gating is most effective with long idle events, or at least predictable-duration events. Examples of predictable events are where all idle events are of a fixed duration, or the idle events are bimodal, either very short or very long, or the short events come in bursts that can be predicted.

Some of the applications do fall into these predictable categories. A significant portion of the idle events in Backprop are events of duration less that 100μsec. Hence filtering out the short duration events (e.g., by delaying entry into C6 until some time has passed), which represent about 90% of the total number of events, will result in good idle power savings. However, filtering
out short duration events might be detrimental for other applications such as *ColorFill*, and *TextureFill*. These benchmarks do not have many short duration idle events, and most of the events and idle time are composed of 1msec events. By having a static algorithm that always delays entry into C6 in order to respond to the needs of *Backprop*, we will not be able to capture the full power savings possible for *ColorFill* and *TextureFill*.

Other applications are less predictable and thus more problematic. *StartApps*, which has low utilization and should be a good candidate for C6 entry, has many short duration idle events (>95% are 1msec or less). Therefore, there are very few long duration idle events, and it is difficult to identify those long duration events since the events in general are widely distributed with respect to their idle durations.

Another important factor that can significantly impact C6 decisions is the amount of cache dirtiness that is introduced by the active execution prior to the CPU becoming idle. Next we will examine active periods and cache dirtiness properties of different benchmarks.

### 4.2.3 Cache Dirtiness Characterization

Whenever the CPU becomes idle, dirty lines in the deactivated caches need to be flushed to either main memory or caches that are not power gated. Cache dirtiness depends on the benchmark, as well as the duration that the benchmark was active before the CPU became idle. This section examines active time periods and the induced dirtiness.

Fig. 4.3(a) shows the average durations of active events that are followed by long idle events. For this particular analysis, we refer to idle events great than 200μsec as long, as they may potentially be C6 entry opportunities. Fig. 4.3(b) shows the growth in cache dirtiness as a function of the active time. Dirtiness was evaluated by collecting memory traces and doing cache simulation as explained in Section 4.3.

Fig. 4.3(b) shows that even for the same active execution period, benchmarks differ in the cache dirtiness that they induce. *VideoScaling* and *WebBrowser* cause more than 50% of the
Figure 4.3: (a) Duration of active events preceding long idle events. (b) Growth of cache dirtiness with time.

cache to be dirty for their average active execution periods (∼10msec). However, StartGame and PicStore exhibit a very low growth rate and final level of dirtiness. Although, WebDecrypt exhibits large active times, the maximum dirtiness it causes in the cache is ∼20%. Benchmarks also vary in the time when dirtiness stabilizes; however, all benchmarks exhibit cache dirtiness growth up to 10msec of active time. On average across all benchmarks, an active time lasts about 3msec and causes about ∼15% of the cache to be dirty.

To craft an effective power gating application, we must understand both application behavior and the power gating mechanism itself. The next section describes the latter.
4.3 Modeling C6 Timing and Overheads

This section first describes the C6 process used in current hardware to establish the goals and challenges of different power gating control policies. Next, it explains our modeling infrastructure and presents results to validate our methodology. We use the AMD 2nd Generation A-Series APU (formerly codenamed “Trinity”) A8-5600K with a 100W TDP running Windows 7 in performance mode at 3.6GHz (fixed P0 state when active) in our study. The A8-5600K contains two dual-core modules and a GPU core. Each module has two cores (for a total of four), and the cores in a single module share a front end, the floating-point unit, and a 2MB L2 cache. Each module can also be power gated, independent of the other module. A module is a candidate for power gating if both cores within the module are idle.

**Figure 4.4**: CPU behavior and steps in clock gated vs. power gated state.
4.3.1 C6 Entry and Exit Process

Fig. 4.4 shows the details of the C6 entry and exit process. The top half of the figure shows the general steps that occur when a module is halted but does not enter the C6 sleep state. As soon as the halt occurs, the local clocks for the module are gated, resulting in some power savings. However, global clocks are still operational. Next, P-state changes are initialized to drop to lower active states in order to reduce voltage and hence reduce leakage power consumption. The P-State transition time is dependent on the absolute voltage change, the rate of voltage drop and minimum P-state residency requirements imposed by system management. The cores then continue to be in a clock gated state. Once work arrives, execution restarts with near negligible overheads.

In the bottom half of Fig. 4.4, we show the steps necessary for a core to enter C6 (i.e., be power-gated). Once idle, the core can choose to continue to remain in a clock gated state for some time, a state referred to as PreFlush. The PreFlush state is used by current generation AMD processors from the 15h family to filter out very short duration idle events [BIO12] and is conceptually equivalent to the IdlePG heuristic proposed in [MBBA11]. Note that the PreFlush timer can be set to values less than the time it takes to waterfall P-state changes to the lowest active state.

As soon as the PreFlush timer expires, all modified data in the caches are flushed to main memory in the CacheFlush step. In other systems, the dirty data may be flushed to a non-gated cache (such as an L3 cache). This operation requires a variable amount of time depending on the number of dirty lines in the cache. The PreFlush and CacheFlush steps are both interruptible (i.e., they can be stopped once started).

After the caches are flushed, the core architectural state including machine state registers (MSRs) are copied into memory during StateSave and power is ramped down. The duration of this step depends on the voltage ramp down time and reliability limits imposed to reduce voltage swings. After saving the state, the core enters the PowerGate state. PowerGate is the lowest
possible power state in which header or footer power transistors are switched off, effectively removing voltage from the core and its caches. The power consumed in C6 state is very low but non-zero [RKQK12]. The duration of time at which the module remains in this state is variable depending on the length of the idle period.

Once work is again available, the process to bring the core out of C6 is initiated. Depending on the source, the wake up interrupt propagates through the northbridge or southbridge controller to the system management unit and initiates a power ramp up. After power is ramped up, reset, fuse propagation, and micro-code patches are applied, followed by the StateRestore step to complete the C6 exit. StateRestore copies the saved system state from the back-up location in the main memory to the core and resumes normal operation.

Note that the time taken to propagate the interrupt, ramp up power, and perform StateRestore, manifests as delay and impacts the start of execution of the next available work (Task 1 execution is delayed in Fig. 4.4).

StateRestore restores only the architectural state of the core. Cold cache, TLB, and branch effects are an additional cost of a C6 event and can increase the duration of the post-C6 active event Task 1 as shown in Fig. 4.4. The combined effects of late start and cold miss penalties reduce the responsiveness of applications, and may manifest as actual delay in overall program run-time. Thus, the performance and energy cost of C6 exit is the sum of physical C6 overheads (fairly constant), cache flush time (dependent on the workload), and cold start effects (also dependent on the workload).

4.3.2 Simulation Model

Architectural simulation of benchmarks under study is not possible because of the unavailability of timing-accurate public domain simulation tools that can execute PCMark 7 under Windows 7 or realistically model the execution of CPU-GPU benchmarks on real hardware, including graphics drivers and OS activity. Thus, we developed a trace-driven simulator to estimate
the impact of C6 management on performance and energy. The simulator replays traces of idle and active states that are captured using runs of these benchmarks on our target system hardware (the A8-5600K APU) and is able to quantify the effects of different C6 policies. The trace driven methodology we employ is similar to that used in [MGW09] and [BJ12]. This methodology allows us to combine the accuracy of real traces of complex application behavior collected on real hardware with the ability to manipulate the power gating policies implemented in the architecture. The duration and power cost of power state transitions is assessed based on measurements on our actual hardware as described below.

![Figure 4.5](image.png)

**Figure 4.5**: Measured package power (msec interrupts).

**Trace Collection.** We capture hardware traces using the Windows Xperf tool [XPc]. Xperf is an event-based performance analysis tool provided in the Windows Performance Toolkit. Xperf captures all idle and active transition events at one microsecond granularity for each core. We obtain Xperf traces at the core level and convert them to activity and idleness traces at the module level. If either of the cores in the module is active, the module is modeled as active.

To obtain estimates of cache dirtiness, we use the AMD SimNow™ [Sim] simulation infrastructure to model a Windows 7 based AMD A-Series APU system and collect traces of
memory instructions. SimNow is a functionally accurate instruction level platform simulator, allowing us to run full system simulations to completion. However, SimNow is not timing accurate and hence its role is limited to only collecting memory traces in this work. After collecting memory traces, we use an in-house cache simulator to simulate a 2MB sized cache and generate average statistics on the expected dirtiness of the cache for a specified active time.

**Micro-benchmarking.** In order to determine latencies and power associated with different steps of the C6 process, we perform a micro-benchmark analysis on our system when connected to a power measurement infrastructure. We use the windows multimedia libraries to initiate interrupts every 1 msec and recorded the change in power at a sample rate exceeding 1MHz for the package. We also vary the PreFlush latency and perform measurements at latencies of 0 usec, 300 usec and 800 usec respectively. The results of our analysis are shown in Fig. 4.5. Note that the power shown in the figure is the combined power of the CPU and on-chip GPU. The figure shows a single timer interrupt window, but with three different values for the PreFlush latency.

Fig. 4.5 enables us to clearly visualize the time and power associated with the different C6 steps. We observe active execution that lasts until 125-200 usec followed by P-State changes or direct entry into power gated stated (for PreFlush 0 usec). After P-State changes, we see the processor waiting in the PreFlush state followed by power gating entry on timer expiry around 400 usec or 850 usec. This methodology enables us to capture precise time periods of the various C6 steps. For example, StateRestore including power ramp-up takes about 80 usec to complete and P-state changes last about 125 usec. We also use this analysis to derive average power levels for different phases on the C6 process. The analysis gives us good insights into the C6 process. We see the relatively high in-efficiencies of staying at clock gated state by comparing the power of active execution, PreFlush state, and power gated states. Note that the clock gated power is actually much closer to active power than power-gated state.

To derive the remaining parameters, we use a combination of measurements and analytical
models. For example, CacheFlush step latencies are modeled using an analytical analysis that calculates the time required to flush the cache by estimating the maximum time required to flush a completely dirty cache and estimated dirtiness based on the active time prior to becoming idle supplied by the active time trace. The maximum time to flush the cache is determined by the time required to flush a total of 32K cache lines (2MB cache with 64byte cache lines) to the main memory. Assuming reasonable values of dram and queuing latencies, we derived a time value in the range of 1-1.5\( \text{msec} \) for the CacheFlush step for a fully dirty cache. We assume a value of \( \text{1msec} \) as a lower bound for this step. This is consistent with values in [BIO12]. We assume no extra power to flush the cache lines to memory beyond the power consumed by the processor in clock gated state. This gives a lower bound on the energy costs of power gating.

We derive active power associated with the different cores of the modules by running single and multi-threaded versions of compute intensive benchmarks. Cold miss time penalties are derived by building a regression model based on trace collection. We first collected active times with power gating off and then compared them with active times with power gating always on (PreFlush timer 0usec). We derived a single value of the active time overhead of cold miss penalties using this approach. Although we model a single value for cold miss overhead, this value will also vary based on runtime behavior and cause even greater variance in the "break-even point." The analysis of this effect is reserved for future work. Lastly, the power consumed during the cold miss period is assumed to be the same as the active period.

Table 4.2 provides a detailed list of values of different parameters used in our model along with information as to how they were derived. Note that the module drops to the voltage level of P-State P5 to save leakage power during the PreFlush stage.

Lastly, we use process development kit (PDK) information and ITRS projections to derive dynamic and leakage scaling factors from planar 28nm to FinFET 16nm technologies. We are unable to publicly declare the values, but the important trends to note are the relative scaling values of dynamic and leakage power for FinFET transistors. While leakage power
reduces significantly vs. planar transistors, dynamic power does not scale (or increases) because of increased capacitance of FinFET transistors. Hence the overheads of power gating that cause dynamic power consumption proportionally increase for FinFET transistors vs. planar technologies.

**Table 4.2**: Parameters of the trace based simulation model.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active time</td>
<td>Variable</td>
<td>Captured using Xperf [XPe]</td>
</tr>
<tr>
<td>Idle time</td>
<td>Variable</td>
<td>Captured using Xperf [XPe]</td>
</tr>
<tr>
<td>P-State change time</td>
<td>125μsec</td>
<td>Measured in hardware</td>
</tr>
<tr>
<td>PreFlush time</td>
<td>Variable</td>
<td>Design parameter [BIO12]</td>
</tr>
<tr>
<td>Max CacheFlush time</td>
<td>1μsec</td>
<td>Analytically modeled</td>
</tr>
<tr>
<td>CacheFlush time</td>
<td>Variable</td>
<td>Modeled with mem traces [Sim]</td>
</tr>
<tr>
<td>StateSave time</td>
<td>100μsec</td>
<td>Measured in hardware</td>
</tr>
<tr>
<td>StateRestore time</td>
<td>80μsec</td>
<td>Measured in hardware</td>
</tr>
<tr>
<td>Cold miss penalty</td>
<td>25μsec</td>
<td>Derived using traces</td>
</tr>
<tr>
<td>Single core dyn power</td>
<td>10.9W</td>
<td>Measured in hardware</td>
</tr>
<tr>
<td>Second core dyn power</td>
<td>6.1W</td>
<td>Measured in hardware</td>
</tr>
<tr>
<td>P5 Leakage power</td>
<td>8.5W</td>
<td>Measured in hardware</td>
</tr>
<tr>
<td>P0 Leakage power</td>
<td>14.3W</td>
<td>Scaled P5 leakage [FAB+11]</td>
</tr>
<tr>
<td>Constant dyn power</td>
<td>4.2W</td>
<td>Global clock estimates [FAB+11]</td>
</tr>
<tr>
<td>PreFlush power</td>
<td>12.7W</td>
<td>P5 leakage + constant dyn</td>
</tr>
<tr>
<td>CacheFlush power</td>
<td>12.7W</td>
<td>Same as PreFlush</td>
</tr>
<tr>
<td>Avg. StateSave power</td>
<td>8.8W</td>
<td>Measured in hardware</td>
</tr>
<tr>
<td>Avg. StateRestore power</td>
<td>8.8W</td>
<td>Measured in hardware</td>
</tr>
<tr>
<td>Leakage at C6</td>
<td>2.1W</td>
<td>5% of peak power [FAB+11]</td>
</tr>
<tr>
<td>FinFET scaling factors</td>
<td>-</td>
<td>Derived using PDK/ITRS</td>
</tr>
</tbody>
</table>

**Model Verification.** In order to validate our trace based simulation infrastructure, we captured active time traces with power gating turned off and then with PreFlush timer set to 0μsec in hardware. We use the former trace (with no power gating) to drive our simulations of the latter configuration, for each of our benchmarks. We then validate the simulation with two different measurements of the real system with power gating. The first is the frequency of gating events (measured with counters on the system under test) and the second is the average event active time
gathered from the real trace. Our simulation of the latter, in particular, must account for cold start effects, idle events not power gated, etc. The results are shown in Fig. 4.6. For both the tests, we see a very good match between hardware based measurements and modeling. This confirms the validity of our timing and cold miss models.
4.4 Towards Oracle C6 Entry

As described in Section 4.3, whenever an entity idles there is an option to either stay in clock gated state or initiate entry into C6. This section first describes the state of the art algorithms from commercially available implementations today. Next, we propose algorithms that outperform state of the art implemented algorithms in energy efficiency.

4.4.1 Existing C6 Techniques

We examine three specific C6 entry algorithms, fixed preflush filtering, autodemotion, and recent value prediction.

**Fixed Preflush Filtering (FPF).** The FPF algorithm delays C6 entry for a fixed amount of time, referred to as the PreFlush latency, in order to filter out the high frequency, low duration idle events that can be detrimental to performance and energy. This scheme is used in production in the AMD 15h processor systems [BIO12].

**Autodemotion.** The Autodemotion algorithm also tries to filter out short duration idle events. Autodemotion assumes that short duration idle events are clustered and come in interrupt storms [Jia]. Therefore, the approach taken in Autodemotion is to either enable or disable C6 entry based on the frequency of idle events. If the number is greater than or equal to a threshold of events/sec for the current time interval, C6 is disabled for the next time interval. Otherwise, C6 is enabled and the system immediately enters C6 after P-State changes. The Autodemotion scheme is implemented in the Intel SandyBridge family of processors [Jia].

**Recent Value Prediction (RVP).** The RVP algorithm predicts the duration of the current idle event by maintaining a history and taking the moving average of N previous idle events. Based on the predicted duration of averaged values, a decision can be made to enter into C6 or remain clock gated. The RVP algorithm is used as the idle duration prediction algorithm in the Linux idle governor [lin] [MWT+12].
All of the above algorithms rely heavily on a threshold to choose good operating points – FPF on the preflush latency, Autodemotion on the number of events that constitute a storm, and RVP on the history length. We choose a value of 200\(\mu\)sec and 5,000 events per second as thresholds for the FPF and Autodemotion schemes respectively. The selected values were found experimentally to be near optimal for our hardware and workloads. For RVP, we found that using the last 64 values to predict the idle duration provides the best prediction and energy efficiency.

Using our simulation infrastructure we performed an evaluation of the relative energy efficiency of all the existing schemes as compared to Oracle gating. Oracle gating is the best possible power gating scheme – it uses precise information on the length of the idle events and cache dirtiness to power gate (immediately) only when it saves power. To summarize our findings, we found that each of the three schemes fell short of Oracle gating by at least 6% for planar transistors and at least 11% for 16nm FinFET transistors. We will present detailed results in Section 4.5. Out of the three schemes, RVP worked the best and Autodemotion worked slightly better than FPF. Next we identify the reasons for the relatively poor energy efficiency of existing schemes.

![Figure 4.7: Variation in energy break-even point.](image)
4.4.2 Key Shortcomings

Each of the three schemes suffers from one or more significant problems. Although simplistic and easy to implement in hardware, the FPF scheme divides short and long events with complete accuracy, but does so by taking a power hit on every long idle event (that should have been power-gated immediately). The Autodemotion scheme does a crude prediction (all events are short or all events are long), and suffers from limited accuracy as it performs an aggregate prediction to an unknown number of future events.

Amongst the three schemes, RVP performs the best as it attempts to use recent history to guess the current event duration. The relative success of RVP shows that idle events are correlated enough to allow us to do more accurate predictions. The success of autodemotion over FPF also implies some level of predictability. However, the fact the RVP falls short of the Oracle implies that there is potential for more accurate prediction than provided by this simple mechanism.

Another very significant contributor to the lack of efficiency in all of the three schemes is that all are based on the assumption of a static break-even point. As discussed in Section 4.2, cache dirtiness is a function of the benchmark and active time prior to the idle event. Fig. 4.7 shows the variation in the break-even point for one module of our modeled system, at different cache dirtiness levels, for both planar 28nm and FinFET 16nm transistors. With no dirtiness in the cache, the break-even point is about 200\(\mu\)sec for planar and about 425\(\mu\)sec for FinFET transistors. Since FinFET transistors offer significantly lower leakage, it makes sense to start in clock gated state longer – hence the longer break-even point for FinFET transistors.

Fig. 4.7 clearly demonstrates that the break-even point varies significantly with cache dirtiness. In fact, it varies over an extremely wide range of 5X to 7X. Hence, any scheme that only takes into account (predicted) event durations is still highly likely to make bad decisions even with perfect prediction.

Next we propose C6 entry algorithms that address the main shortcomings of existing techniques – lack of prediction accuracy and not incorporating cache dirtiness information.
4.4.3 Proposed Algorithm

We propose a scheme that predicts idle durations at a finer granularity (a new prediction for each event), using more careful analysis of past behavior. The proposed scheme is expected to be implemented in the system control unit and has to make sleep-state decisions once every few usec — hence the computational complexity must remain low.

**Linear Prediction (LP).** The LP algorithm predicts the duration of the current idle event by maintaining a history and making a prediction by building linear prediction models \[\text{Mak75}\] of previous idle event durations. Linear prediction estimates future events of a time series as linear functions of \(P\) prior observations as shown in equation (4.1):

\[
\hat{Y}(t) = \sum_{i=1}^{P} a(i) \times Y(t-i) \tag{4.1}
\]

In the above expression, \(a(i)\) represent the prediction coefficients and \(P\) dictates the number of prediction coefficients used. The error of the prediction, \(e(t)\), is:

\[
e(t) = Y(t) - \hat{Y}(t) = Y(t) - \sum_{i=1}^{P} a(i) \times Y(t-i) \tag{4.2}
\]

We use the Levinson-Durbin method \[\text{Lev47} \text{ Mak75}\] to solve the above equation for the coefficients which minimize the mean square error of the prediction. Idle event duration history may be retained for \(N\) events (where \(N \geq P\)) over which the prediction error is minimized for estimating the coefficients. This computation has a complexity of \(NP + P^3\) multiply-accumulate (MAC) operations.

The selection of \(N\) and \(P\) values are design parameters. Larger values of these could potentially exploit long-term recurrence patterns at the expense of increased implementation costs. Larger values of \(N\) could also capture un-correlated data in the model and reduce performance. We performed a design space evaluation over the space of \(N\) and \(P\) and found that \(N=1024, P=1\) yielded the best prediction accuracy.
Although the LP algorithm requires $NP + P^3$ MAC operations, or $N + 1$ when $P = 1$, this only holds when we do a new calculation from scratch. If we reuse the prior calculation, we can do an incremental computation when we add one value to the history buffer, using two multiplications (one for the new value, one for the old value popped off the stack), one subtraction, and one addition. Hence for $N=1024,P=1$, every incremental calculation of coefficients can be calculated in less than 20 operations total. Since the prediction operations typically happen every few $\mu$sec, a load of 20 MAC operations represents a trivial load on system management controllers that run at speeds of hundreds of megahertz.

Mechanics. First, we use the LP scheme to predict the duration of each new idle event. Next, we obtain an estimate or exact calculation of cache dirtiness. The latter can be easily done with a counter that increments when a clean line transitions to dirty, and decrements when a line is written back. Then, the system management unit evaluates potential power savings by evaluating the dynamic break-even point and comparing it with the predicted duration. If gating is found to be beneficial based on power estimates, C6 is entered immediately, otherwise the module stays in clock gated state till the end of the idle event. The system management unit also tracks the accuracy of the prediction scheme at the end of the idle event once the actual idle duration is available. If there are large prediction errors consistently, the LP schemes default to the Autodemotion scheme. The modeled RVP scheme does the same.

4.5 Results

We implement Never power gating, Always power gating, the three baseline industry schemes, our proposed LP predictor using our modeling environment, and generate power and performance numbers for each application both for 28nm planar and 16nm FinFET technologies. The RVP and LP schemes were evaluated both in the context of a static break-even and a dynamic break-even calculation. For the former, they assume a break-even latency based on the C6 delays.
and costs associated with the fixed average measure (15%) of cache dirtiness. For the latter, a dynamic break-even is calculated based on counters that measure actual dirtiness. This could be a simple linear calculation based on Fig. 4.7.

Fig. 4.8 (a) and (b) summarize the energy and performance findings over the complete set of benchmarks. All results are shown relative to the 28nm planar and 16nm FinFET specific Oracle prediction schemes with perfect knowledge of durations and dirtiness.

As seen in Fig. 4.8, both Never and Always have significant penalties vs Oracle gating. Never has a slight performance advantage of less than 5% but an energy disadvantage exceeding 50%. Always loses about 15% performance. The impact of always gating is worse for FinFET
16nm as the ratio of dynamic energy vs. leakage is high for FinFETs vs. planar, hence, the cumulative power gating entry and exit overheads become more significant for FinFET transistors.

All of the baseline industry schemes (FPF, Autodemotion, and RVP) are more energy efficient than Never and Always gating. For planar transistors, FPF (13.2%) is much worse than Autodemotion (7.9%) as the leakage energy costs are higher in planar transistors. However, for FinFET transistors, FPF (12.6%) and Autodemotion (11.4%) are more similar.

The RVP+Avg Dirtiness and LP+Avg Dirtiness schemes only offer a very slight or no advantage over Autodemotion. This implies that any additional accuracy in the estimation of idle time is of little use given the inaccuracy of the break-even point. However, when combined with actual measures of dirtiness, RVP+Dirtiness and LP+Dirtiness schemes improve significantly. For FinFET transistors, RVP+Dirtiness improves to within 7.1% of Oracle, an improvement of more than 5% over RVP+Avg Dirtiness (12.7%).

LP+Dirtiness outperforms every other scheme as it combines a very good predictor together with measures of dirtiness. To summarize the performance of LP+Dirtiness, it reaches within 3% of Oracle performance for both planar and FinFET transistors. This is nearly an 8% energy advantage over each of FPF, Autodemotion, and RVP+Avg Dirtiness for FinFET transistors. The benefits for planar transistors are more than 4% over all three baseline schemes. Performance of the LP+Dirtiness scheme reaches within ±1% of the Oracle scheme on average. These results clearly demonstrate the benefit of having both a good predictor of idle duration and accurately computing the dynamic break-even point. Having one of the two is insufficient.

Fig. 4.9 (a) and (b) show the energy and performance comparisons between the FPF, Autodemotion, RVP+Dirtiness, and LP+Dirtiness schemes for the complete set of benchmarks. The benchmarks are shown in order of increasing processor utilization, with ColorFill being the lowest, and WebBrowser being the highest.

We notice some general trends in the energy results. FPF generally has higher energy than Autodemotion when the utilization is low, but the trend reverses as utilization increases, indicating
that Autodemote is not as effective at filtering short duration idle events with higher utilization benchmarks. As utilization increases, so does the average duration of active events and cache dirtiness; with long active events between idle events, even if the idle events are all short, they will never qualify as a “storm.” Hence, Autodemote tends to power gate more often and lose efficiency.

Another interesting trend is that RVP+Dirtiness is worse than FPF on several benchmarks with higher utilization (e.g., Pathfinder, Backprop, Gaussian etc.). As utilization increases, with increasing active times, the duration of idle events decreases. FPF filters all short duration idle events regardless of utilization, and is thus less likely to “miss badly” on shorter idle events. RVP+Dirtiness may suffer from a lack of prediction accuracy in such cases and hence FPF

Figure 4.9: (a) Energy for FinFET 16nm. (c) Performance for FinFET 16nm.
outperforms it. With better prediction, the LP+Dirtiness scheme avoids this and outperforms FPF, Autodemote and RVP+Dirtiness in almost all cases.

Fig. 4.9 (b) shows performance results for the different benchmarks. In general, Autodemote has the worst performance amongst all schemes. It suffers penalties exceeding 5% on several benchmarks (StartApps, DirectX9, TextEditor etc.). This is consistent with the reasoning described above; Autodemote biases towards gating more often. StartApps shows an anomaly and loses about 10% performance on FPF even when the FPF scheme is conservative. The reason for this behavior lies in the event distributions for the StartApps benchmark. As shown in Fig. 4.2, StartApps has about 30% of its idle events distributed between 200-400 μsec. Since the FPF scheme uses a threshold of 200 μsec it wastes energy and yet does not filter out these medium sized events. Overall, the LP+Dirtiness scheme matches closest to Oracle performance.

In many cases, we care not only about performance, but also the distribution (e.g., the outliers can create long tail distributions, or worse, delay the completion of an entire parallel computation). The goal of any management scheme is to maximize the benefit as well as minimize any outliers. Fig. 4.10 shows the power penalty relative to Oracle for all four schemes sorted by decreasing utilization. The figure clearly shows that the LP+Dirtiness scheme not only gives the best average gain, but also significantly decreases variation, resulting in a more robust power management algorithm.

### 4.6 Conclusion

This work provides a comprehensive analysis of idle behavior in emerging applications on CPU-GPU systems. It shows that there is no fixed break-even point in core power gating because of the effects of cache dirtiness. It evaluates existing state-of-the-art systems, and demonstrates new techniques to predict when to enter the C6 power gated state. These schemes are shown to provide average energy reduction exceeding 8% over existing schemes, and sacrifice almost no
Figure 4.10: Energy efficiency of C6 management schemes for FinFET 16nm. Benchmarks are sorted by decreasing utilization.

performance.

Acknowledgements

Chapter 4 in full, is a reprint of the material as in it appears in the proceedings of the HPCA 2015. Arora, Manish; Manne, Srilatha; Paul, Indrani; Jayasena; Nuwan; Tullsen, Dean, Understanding Idle Behavior and Power Gating Mechanisms in the Context of Modern Benchmarks on CPU-GPU Integrated Systems. IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015. The dissertation author was the primary investigator and author of this paper.
Chapter 5

Job Scheduling in Density Optimized Servers

The increasing demand for computational power has led to the creation and deployment of large-scale data centers. During the last few years, data centers have seen improvements aimed at increasing computational density — the amount of throughput that can be achieved within the allocated physical footprint. This need to pack more compute in the same physical space has led to density optimized server designs. Density optimized servers push compute density significantly beyond what can be achieved by blade servers by using innovative modular chassis based designs.

This chapter presents a comprehensive analysis of the impact of socket density on intra-server thermals and demonstrates that increased socket density inside the server leads to large temperature variations among sockets due to inter-socket thermal coupling. The chapter shows that traditional chip-level and data center-level temperature-aware scheduling techniques do not work well for thermally-coupled sockets. The research proposes new scheduling techniques that account for the thermals of the socket a task is scheduled on, as well as thermally coupled nearby sockets. The proposed mechanisms provide 2.5% to 6.5% performance improvements across various workloads and as much as 17% over traditional temperature-aware schedulers for
computation-heavy workloads.

5.1 Introduction

The last decade has seen an increase in data center deployments in the enterprise as well as by cloud service providers. With the continued push towards consolidated systems in the enterprise, and the emergence of new applications in domains such as real-time data analytics, industrial IOT and deep learning, this trend is expected to continue \[HPE\] \[Cis16\]. This has resulted in demand for Density Optimized Servers.

Density optimized servers consist of a chassis that provides power and cooling. Compute, memory, storage and connectivity are organized in the form of cartridges. Upgrades to servers can be performed by upgrading individual cartridges. Various combinations of cartridges can be used to create a modular server design where the server is heavily optimized for different workloads such as compute or storage.

Examples of density optimized servers include HPE Moonshot \[Moo\] M700 \[Prob\] based systems that are targeted at enterprise virtual desktop infrastructure (VDI) applications. The HPE Moonshot packs forty-five cartridges, each with up to four AMD Opteron X2150 \[Optb\] sockets in a 4U form factor. Another example is the Cisco UCS M-Series \[Cis\] modular servers. The UCS M-Series has eight compute modules, each with two Intel Xeon E3 sockets \[Xeob\] in a 2U form factor. Similarly, Dell has a lineup of density optimized servers with the PowerEdge FX2 \[Delb\] chassis-based solutions. More recently, Facebook and QCT have released the Rackgo X Yosemite Valley dense-server design \[Rac\] that packs 12 Intel Xeon D-1500 \[Xeoa\] sockets in a 2U form factor.

As server shipments continue to rise, the density optimized server market is growing at a much faster rate than the overall market \[IDC15\] \[IDC\]. According to IDC, in 2015 density optimized servers (also called modular hybrid-blade servers) represented 10% of the server market.
Figure 5.1: (a) Power per 1U, and (b) Sockets per 1U for 400 server designs released from 2007-2016.

and they are expected to rise to represent 30% of the server market share by 2020 [CRN].

The market for density optimized servers is growing rapidly because these designs offer distinct advantages. First, they offer the ability to customize server hardware for the application. Second, these servers optimize for the physical space and hence reduce data center build out costs [Dela]. Third, reduced physical footprint enables more efficient (e.g., smaller or cheaper) cooling systems [Dela]. Fourth, as the physical footprint shrinks, operational, maintenance, and IT administration costs are reduced [HPE].

Density optimized servers provide these advantages by packing more CPU sockets per unit volume than traditional server designs. Figure 5.1 shows socket and power density for 400 servers from published results of SPECpower_ssj2008 benchmark data [Spe16]. All published
data from 2007-2016 was considered except tower servers. Data for density optimized servers was estimated from manufacturer specifications.

Figure 5.1 shows that 1U servers consume more power per unit volume on average (208 Watt/U) than 2U (147 Watt/U) and “Other” rack server designs (114 Watt/U). This can be attributed to the higher socket density of 1U designs (1.79 Sockets/U) as compared to 2U (1.15 Sockets/U) or “Other” (0.78 Sockets/U). Blade servers exhibit power (421 Watt/U) and socket density (3.47 Sockets/U) that is nearly double that of 1U designs. This is because Blade servers are optimized to pack in more sockets with the use of a shared chassis. Density optimized servers are an evolution of Blade server designs and push the envelope even further. They exhibited power density of 588 Watt/U and socket density of about 25 Sockets/U based on a study of 10 server designs from 4 vendors. This is nearly a 50% increase in power density along with nearly a 6X increase in socket density over Blade server designs.

As density optimized servers gain popularity, it is important to consider the impact of higher power and socket densities on system performance, especially since the individual cartridges share resources. As previous work in balanced system design has demonstrated, performance is not just a matter of increasing the number of cores or size of the machine. Rather, any such increase needs to be accompanied with efficient management techniques to maximize performance [ST00] [MTH+11] [PMA+13].

In this research, we specifically focus on understanding intra-server thermals because of the use of a shared cooling system. With this understanding, we investigate new scheduling techniques that improve performance of density optimized servers. In particular, workload schedulers need to now account for thermal coupling. Since many sockets share the cooling system, the placement of a single job not only impacts the temperature profile of the scheduled socket, but potentially every socket downwind of that socket.

Thermal coupling is defined as the heat-transfer phenomenon of one heat source affecting the temperature of other heat sources in close physical proximity. Thermal coupling is pervasive
in computing systems and previous research has demonstrated the impact of such interactions on cores [SSS+04], in 3D stacked systems [PL07], in CPU-GPU systems [PMA+13] and between DRAM DIMMs and CPU sockets in a server system [ANR12]. With increasing socket counts and sharing of the cooling system, sockets within dense servers exhibit strong thermal coupling. This can impact scheduling algorithms.

This research makes the following contributions:

1. Using real-world server configurations including the HPE Moonshot system, this work first demonstrates how socket density creates a new type of temperature heterogeneity inside dense servers. This causes significant temperature difference inside the dense server even at reasonable single-socket power and cooling levels.

2. This work studies temperature-dependent scheduling schemes used at the chip-level as well as data-center system level, and demonstrates that existing algorithms are insufficient because they do not take into account the effects of inter-socket thermal coupling caused by directional air-flow.

3. Lastly, the research proposes a new scheduling scheme that shows gains in performance and energy efficiency. In particular, for computation-heavy workloads we see as much as a 17% performance gain over a traditional temperature-aware scheduler, and across all load levels, we average 2.5% to 6.5% gains for various workloads.

The rest of the chapter is organized as follows. Section 5.2 presents an analysis of socket density and temperature heterogeneity in dense servers. Section 5.3 describes our evaluation methodology. Section 5.4 describes existing scheduling algorithms, their key shortcomings, and describes our proposed enhancements. Section 5.5 presents results and Section 5.6 concludes the chapter.
5.2 The Impact of Socket Density

The HPE Moonshot system can pack in as many as 180 sockets in a 4U chassis. This level of density can create socket to socket interactions that impact performance and efficiency. Using the HPE Moonshot and other recently proposed density optimized servers as examples, in this section, we analyze the influence of socket density on intra-server thermals and application performance.

Figure 5.2 shows a computational fluid dynamics (CFD) model of an HPE Moonshot Proliant M700 cartridge constructed using Ansys IcePak. The system has four sockets, arranged in a 2 X 2 configuration. Each socket consumes 15 Watt of power. Air flows horizontally from left to right and passed over sockets in series. Figure 5.2 shows that, while cool air flows over the sockets on the left, the sockets on the right receive air that is higher in temperature. The measured average air temperature difference between the left and right sockets was 8°C. Consequently, the second set of downstream sockets have a higher ambient temperature and may reduce performance when operating under a temperature limit.

To mitigate chip temperature differences because of inter-socket thermal coupling, the HPE Moonshot Proliant M700 cartridge uses 2 types of heat sinks. The upstream sockets that
receive cooler air have 18 fins in the heat sink vs. 30 fins for the downstream sockets. The use of distinct heat sinks adds yet another dimension to the thermal heterogeneity in the system, increasing the difficulty of making optimal scheduling decisions. For example, we can choose to place a job on the socket with the coolest ambient air, but that choice heats up at least 1 more downstream socket.

Figure 5.3: (a) Organization, and (b) Relative performance of Coolest First (CF) and Hottest First (HF) scheduling for coupled and un-coupled designs.

Figure 5.3 (a) shows a 2-socket system with different heat sinks arranged in a coupled (similar to the M700 cartridge) and un-coupled manner (similar to traditional 2-socket 1U server). Part (b) shows the relative performance when using two scheduling schemes, Coolest First (CF)
and Hottest First (HF) at 50% utilization. The CF scheme schedules jobs on the coolest available core and keeps work away from the hotter cores. CF-style scheduling algorithms have been demonstrated to work well as scheduling algorithms [GPV04] [MCRS05] [WvLD+09]. The HF scheme does the exact opposite – it schedules work at the hottest possible idle socket, which would not generally be expected to be a good strategy.

As expected, for an uncoupled system, CF outperforms HF by about 8%. However, for a coupled system, HF outperforms CF by about 5%. This happens because the HF scheme mitigates thermal coupling effects by scheduling less work on the upstream socket. This results in lower air temperature at the downstream socket. The better heat sink at the downstream socket also results in better performance. This simple motivational experiment shows that scheduling policies such as CF that have worked well for chip-level or data-center level scheduling may not work for thermally coupled systems.

### 5.2.1 Socket Organization in Dense Servers

![Diagram](image)

**Figure 5.4:** Thermal coupling in density optimized servers.

An important factor that differentiates density-optimized servers from traditional designs is socket organization. As demonstrated in the previous section, differences in socket organization
(e.g., coupled vs. un-coupled) can lead to different performance of scheduling schemes. In this section, we will look at how sockets are organized in current density optimized systems and using an analytical model of heat-transfer, we estimate temperature variations across sockets inside density optimized systems.

Figure 5.4 illustrates some choices around socket organization in dense systems. As shown in the figure, sockets can be organized in a coupled or un-coupled manner. More than two sockets can also be organized in a coupled manner to yield a tighter system fit but at a higher degree of coupling. We define degree of coupling as the maximum number of sockets that may have a significant thermal interaction caused by sharing of the cooling system.

Figure 5.4 also demonstrates socket entry temperature. We define socket entry temperature as the average temperature of air before it passes over a socket. As shown, systems with higher degree of thermal coupling and shared cooling show progressively higher entry temperatures if all sockets consume power.

<table>
<thead>
<tr>
<th>System</th>
<th>Details</th>
<th>Domain</th>
<th>Size (U)</th>
<th>System Organization</th>
<th>Total Sockets</th>
<th>Sockets per 1U</th>
<th>Socket TDP(W)</th>
<th>CPU</th>
<th>Coup -d(ing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rackgo X</td>
<td>OpenCompute</td>
<td>General</td>
<td>2</td>
<td>2T x 3B x 2S</td>
<td>12</td>
<td>6</td>
<td>45</td>
<td>Xeon D-1500</td>
<td>1</td>
</tr>
<tr>
<td>AMD IAMD</td>
<td>SM15000e</td>
<td>Scale-out</td>
<td>10</td>
<td>4R x 16CR x 1S</td>
<td>64</td>
<td>6.4</td>
<td>140</td>
<td>Opteron 6300</td>
<td>1</td>
</tr>
<tr>
<td>UCS M4308</td>
<td>M2x14</td>
<td>Scale-out</td>
<td>2</td>
<td>2R x 2CR x 2S</td>
<td>8</td>
<td>4</td>
<td>120</td>
<td>Xeon E5</td>
<td>1</td>
</tr>
<tr>
<td>Moonshot</td>
<td>M101p</td>
<td>Big data</td>
<td>4</td>
<td>15R x 3C x 1S</td>
<td>45</td>
<td>11.25</td>
<td>69</td>
<td>Xeon E3</td>
<td>2</td>
</tr>
<tr>
<td>Copper</td>
<td>Prototype</td>
<td>Scale-out</td>
<td>1</td>
<td>2R x 4S</td>
<td>8</td>
<td>8</td>
<td>50</td>
<td>X-Gene</td>
<td>3</td>
</tr>
<tr>
<td>Datan</td>
<td>Prototype</td>
<td>Scale-out</td>
<td>10</td>
<td>4R x 16CR x 4S</td>
<td>256</td>
<td>25.6</td>
<td>8.5</td>
<td>Atom N570</td>
<td>3</td>
</tr>
<tr>
<td>SeaMicro</td>
<td>SM15000</td>
<td>Scale-out</td>
<td>10</td>
<td>4R x 16CR x 4S</td>
<td>256</td>
<td>25.6</td>
<td>8.5</td>
<td>Atom C2750</td>
<td>5</td>
</tr>
<tr>
<td>Moonshot</td>
<td>M350</td>
<td>Hosting</td>
<td>4</td>
<td>15R x 3C x 4S</td>
<td>180</td>
<td>45</td>
<td>20</td>
<td>Atom C2750</td>
<td>5</td>
</tr>
<tr>
<td>Moonshot</td>
<td>M700</td>
<td>VDI</td>
<td>4</td>
<td>15R x 3C x 4S</td>
<td>180</td>
<td>45</td>
<td>22</td>
<td>Opteron X2150</td>
<td>5</td>
</tr>
<tr>
<td>Moonshot</td>
<td>M800</td>
<td>DSP</td>
<td>4</td>
<td>15R x 3C x 4S</td>
<td>180</td>
<td>45</td>
<td>14</td>
<td>Keystone I</td>
<td>5</td>
</tr>
<tr>
<td>Redstone</td>
<td>HPR12</td>
<td>Prototype</td>
<td>4</td>
<td>4T x 6R x 3C x 4S</td>
<td>288</td>
<td>72</td>
<td>5</td>
<td>EnergyCore</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 5.1 shows recently released density optimized server systems and compares them across various dimensions. Density for such systems varies from about 4 Sockets/U to as high as 72 Sockets/U. The systems with higher socket densities tend to use lower power sockets. A study of system organization shows that systems are organized in modular form as rows/trays.
of cartridges/boards, each with multiple sockets. Power for the sockets used in these systems varies from very low power cores at 5W per socket to as high as 140W per socket. The degree of thermal coupling varies from 1 to as high as 11.

Table 5.1 shows that the design space of density optimized systems is fairly large. To consider the impact on inter-socket thermals of various socket organization choices, we construct an analytical model of socket entry temperature for various power, degree of coupling, and airflow levels.

<table>
<thead>
<tr>
<th>Server Size</th>
<th>Power per 1U (W)</th>
<th>Air-flow (CFM) needs per 1U (DeltaT = 20°C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1U</td>
<td>208</td>
<td>18.30</td>
</tr>
<tr>
<td>2U</td>
<td>147</td>
<td>12.94</td>
</tr>
<tr>
<td>Other</td>
<td>114</td>
<td>10.03</td>
</tr>
<tr>
<td>Blade</td>
<td>421</td>
<td>37.05</td>
</tr>
<tr>
<td>DensityOpt</td>
<td>588</td>
<td>51.74</td>
</tr>
</tbody>
</table>

5.2.2 Analytical Model of Socket Entry Temperature

In order to see the impact of socket density on intra-server thermals, we build a simple analytical model based on heat transfer theory – this complements our more complex models later in the chapter. Electronic systems using forced air cooling rely on the transfer of heat between hot components and cold air being pushed through by fans in order to maintain the temperature. One of the critical limits imposed on server systems is the hot-aisle temperature limit. This limit sets the maximum temperature of air from the server outlet for human comfort. For example, Facebook data center hot aisle temperatures can be as high as 49C with inlets set to about 29C [Fac11]. This means that there must be enough air-flow provisioned to remove heat and maintain an outlet-inlet temperature difference of 20C. ASHRAE TC 9.9 guidelines [ASH11]
also mention a typical server temperature rise of 20C from inlet to outlet.

Figure 5.5: (a) Mean socket entry temperature, and (b) Socket entry temperature coefficient of variance, a measure of the variance in heat across the coupled sockets [COV].

Table 5.2 shows the air-flow requirements in order to maintain a 20C temperature difference between the inlet and outlet for various server systems. Air flow in the table is measured in cubic feet per minute (CFM) and power numbers are average server power levels as previously discussed in section 5.1. CFM levels were calculated using the standardized total cooling requirements formulation of the first law of thermodynamics [Sun]. Table 5.2 shows that to maintain a temperature difference of 20C between the inlet and the outlet, fans need to supply between
18.3CFM to 51.74CFM of air per 1U for different server power levels. High-end server fans, such as Activecool fans [Act08] used in HPE density optimized servers can meet such airflow requirements at reasonable power levels.

While the total CFM requirements are determined by the outlet temperature limits, the temperature inside the server is dependent on the socket power, degree of coupling, and the airflow on each socket. We use the values in Table 5.2 to calculate reasonable bounds on per-socket airflow. Next, we use socket power and degree of coupling combinations, with the standardized total cooling requirements formulation of the first law of thermodynamics [Sun] to derive the temperature of air arriving at points within the server. Figure 5.5 shows the results of our analysis.

Figure 5.5(a) demonstrates that as the degree of coupling increases, the mean socket entry temperatures also increases. As expected, the mean entry temperature for high-powered sockets is high, but even low-powered sockets can have high mean entry temperature at different airflow values. For example, a 15 Watt part with 6CFM of airflow can have about a 10C mean entry temperature difference for a system with degree of coupling 5, as compared to a system with degree of coupling 1. Figure 5.5(b) shows that not only is the mean socket entry temperature high, but inter-socket variations are also high and increase even further as the degree of coupling increases. The data demonstrates that socket organization can play a major role in intra-server thermals for systems with high degree of thermal coupling.

As table 5.1 shows, density optimized systems vary in organization, processor choices and application domains. In order to study optimization opportunities, we pick a single representative system used for its intended application domain. We pick the HPE Moonshot Proliant M700 system for this deeper analysis. The target application market for this system is Virtual Desktop Infrastructure (VDI) intended for enterprise desktop consolidation [Prob]. With 180-sockets packed in 4U space, the Moonshot system is highly representative of density optimized systems and is intended for a popular enterprise data-center use-case [HPE]. Next, we discuss infrastructure used to model our target system.
5.3 Methodology

This section describes details of our modeling infrastructure. Our primary exemplar system is the HPE Moonshot [Moo] M700 [Prob] density optimized server consisting of 180 sockets and targeted at enterprise virtual desktop infrastructure (VDI) applications. Since we are modeling a large scale system consisting of many sockets, and running workloads over a period of minutes, we develop a methodology that enables us to accurately simulate the behavior of such systems in reasonable simulation time. Each part of our infrastructure is either derived from hardware measurements or compared against other validated models.

5.3.1 Workloads

For the purposes of this study, we focus on one particular workload scenario, the Virtual Desktop Infrastructure, where the servers are running desktop applications in support of thin clients or terminals.

To study desktop application behavior, we used PCMark 7 [PCM11]. PCMark is a popular consumer application benchmarking suite, commonly used in the industry to characterize desktop performance. PCMark consists of more than 25 applications characterized into domains such as computation, storage, graphics, entertainment, creativity, and productivity. Out of these, we omitted applications that are not relevant to enterprise VDI (e.g., gaming) for a total of 19 PCMark 7 applications. To easily categorize benchmarks, we divided these applications into 3 sets; Computation intensive (Computation), Storage intensive (Storage), and General Purpose (GP) benchmarks consisting of all remaining applications.

Due to the unavailability of timing-accurate public domain simulation tools that can execute typical desktop applications on Windows, we use a trace based simulation methodology similar to that used in previous research using PCMark 7 [AMP+15]. We capture hardware traces of various PCMark benchmark runs using the Windows Xperf tool [XPe] at various processor
Figure 5.6: (a) Average job duration, and (b) Coefficient of variance of job durations within benchmark sets.

frequency states. Xperf captures idle and active transitions of the socket for the workload at a fine granularity. Using this information, we create a job arrival model for various PCMark benchmarks. We vary the job inter-arrival duration to simulate different loads on the system.

Figure 5.6(a) shows average job durations across all benchmarks for the three benchmark sets. The average job durations were found to be on the order of a few msec. The maximum job durations within each benchmark set were found to be almost 2 orders of magnitude higher. This is in line with previously published analysis of PCMark 7 [AMP+15]. Figure 5.6(b) shows the coefficient of variance across average job durations within the different benchmarks of each set.
The coefficient of variance ranges between 0.25 to 0.33. This shows that benchmarks within each set exhibit similar job durations on average and hence we choose to study benchmarks grouped in sets, rather than studying benchmarks individually.

![Figure 5.7: (a) Workload power (90C) with varying p-states. (b) Relative workload performance vs performance at 1900MHz.](image)

We measured power and performance in hardware at various frequency states to build a complete socket-level workload model. As power is influenced by temperature, we measure both temperature and power together, across varied loads. Using the measured power value, temperature value, and by estimating leakage to be 30% of TDP at the temperature limit (90C), we calculate power at different p-states and chip temperatures. Figure 5.7 shows the power and
performance levels for the different workloads.

The AMD Opteron X2150 used in the system has a TDP of 22 Watts and runs from 1900MHz to 1100MHz [Optb]. We see that the Computation workload uses the most power and Storage uses the least (18 Watt vs 10.5 Watt) at the highest frequency of operation. As frequency decreases, power decreases but more so for Computation than Storage. As expected, the Computation workload is also the most frequency sensitive with performance dropping about 35% for a 800MHz reduction in frequency. Storage is the least frequency sensitive workload. The General Purpose workload exhibits intermediate levels of both power and sensitivity to processor frequency.

5.3.2 Socket Ambient Temperature Model

Socket ambient temperature is a function of the physical design, power consumption of sockets, heat sink design and airflow inside the server box. Using available data for the ActiveCool [Act08] fans used in HPE Moonshot systems, physical dimensions published for the M700 server cartridge [Prob], and socket power model, we used an Ansys Icepak based modeling infrastructure to construct a model of the HPE MoonShot M700 based system. This computationally intensive model tracks heat sources, physical artifacts, fans, and turbulent airflow through the system. This commercial CFD based modeling infrastructure and methodology has been validated on real server designs to be within 1C of actual temperature within the server box [SAH+15a]. This modeling infrastructure yields socket ambient air flow levels and socket temperature based on the server physical design and different socket power consumptions.

Figure 5.8 shows the side view of a single M700 cartridge. The figure shows the model for a single cartridge as well as the second row acting as a lid on top of the first cartridge with its backside facing the first cartridge. The HPE Moonshot has 15 rows of cartridges spanning the width of the server, stacked on top of each other in this side view. We only examine thermal coupling along the direction of airflow in this work. There is some thermal coupling between
cartridges in the z direction across the width of the server; however, our CFD modeling confirms that these are small effects compared to the thermal coupling we model.

### 5.3.3 Chip Peak Temperature Model

To simulate system behavior with reasonable accuracy, it is important to reliably estimate peak chip temperature for a socket with known ambient temperature and chip power consumption. Architecture research has traditionally used compact thermal models like Hotspot [SSB+03] to estimate chip thermals as on-chip temperature differences between cold and hot spots can exceed as much as 50°C [PN] for large sized dies. However, recent research has also shown that socket level thermals in server systems can have time constants of the order of tens of seconds [AIR11] [PMA+13]. Hence, traditional detailed modeling approaches becomes prohibitively expensive when modeling a dense server system with many sockets and running workloads for periods of seconds to minutes.

In order to explore the development of simpler thermal models for peak temperature estimation, we studied on-chip temperature differences for the AMD Opteron X2150 [Optb]. Figure 5.9(a) shows data for temperature differences between the hottest and coldest spots on the die for 19 PCMark 7 benchmarks. The data was collected via a proprietary HotSpot like model.
Figure 5.9: (a) Temperature differences between hottest and coolest spots, and (b) Max temperature vs. power.

that has been validated with thermal camera measurements.

The data shows that different heat sinks do not have a major impact on chip lateral temperature differences, although, in general, the 18 fin heatsink had lower chip lateral temperature differences than the 30 fin heatsink. Interestingly, we see that the temperature differences on die are fairly low for the X2150 and range between 4C - 7C. We attribute this to the small size of the die at about 100mm$^2$ [Kab13a], about $3.5 \times - 6 \times$ smaller than server processor dies [Xeoc].

Figure 5.9(b) shows the maximum chip temperature and power for various PCMark 7 benchmarks using the validated temperature model. The data shows that the 30 fin heatsink provides better thermals than the 18 fin heatsink by about 6C - 7C for high power cases and 3C - 4C for lower power cases. Also, the peak temperature is well correlated with total power. Using
data and insights from the validated temperature model, we develop a simplified peak temperature prediction model for the X2150. We approximate the core peak temperature as:

\[ T_{\text{Peak}} = T_{\text{Amb}} + Power \times (R_{\text{Int}} + R_{\text{Ext}}) + \theta(Power, Sink) \]  

(5.1)

where \( T_{\text{Peak}} \) is the peak chip temperature, \( T_{\text{Amb}} \) is the socket ambient temperature, \( R_{\text{Int}} \) and \( R_{\text{Ext}} \) are the chip internal and heatsink external thermal resistance, and \( \theta(Power, Sink) \) is a linear function that is derived empirically. The model considers the thermal resistance from the die to the external environment but ignores any lateral thermal resistance. The developed model is similar conceptually to the simplified chip temperature model proposed in [AR10], but additionally models the path from the heatsink to the ambient.

Figure 5.10 shows the differences in temperature between our proposed simplified model and the validated proprietary thermal model. The figure shows that our proposed simplified model estimates temperature within 2°C of the validated model. This holds valid irrespective of the heatsink size.

![Figure 5.10: Proposed max temperature model validation.](image)
5.3.4 Overall Model

In this section, we will describe the overall model. Table 5.3 provides values of different parameters used in our model along with information as to how they were derived.

In our simulation model, jobs for various benchmarks arrive as per a probabilistic job distribution model created to match the Xperf measurements previously described. Once the job arrives, it is put into a job queue, which is used by a centralized job controller to allocate jobs. The scheduler checks for job arrivals every 1 \textit{usec}. If there is a job to be scheduled and at least one idle socket available, the scheduler implements policies based on current and historical temperature, physical location, job power, and other parameters.

The scheduler works by first making a list of all sockets that are idle. It then makes a decision to allocate the job from the queue, based on the scheduling policy, on to one of the free sockets. If a free socket is not available, the scheduler does not make any allocation decision and waits to make decisions till a socket becomes available. It only allocates jobs when a socket becomes available and based on the conditions (e.g. temperature) of the time interval when the socket becomes available. In this study, we do not implement unbalanced schedulers that choose to schedule jobs to busy sockets even though there is an empty socket available.

Once the allocation of jobs is complete, the sockets execute work every time-step until the next job arrives. Throughput of every time interval and power are estimated based on the current p-state. Estimated power values are used to calculate the current peak temperature of each socket.

The peak temperature value is used to make p-state change decisions. P-states vary from 1.9GHz to 1.1GHz in 200MHz steps. The higher two p-states are boost states that are used opportunistically to improve performance where thermal headroom is available. A fully loaded socket under reasonable ambient temperatures is expected to only be able to sustain the P2 state (1500MHz) \textsuperscript{[kab13b]}. We implement a power management policy that emphasizes responsiveness and runs jobs at the highest possible PState within the temperature limit. This is a commonly
Table 5.3: Overall simulation model parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job length</td>
<td>Variable</td>
<td>Captured using Xperf [XPe]</td>
</tr>
<tr>
<td>Job arrival time</td>
<td>Variable with load</td>
<td>Captured using Xperf [XPe]</td>
</tr>
<tr>
<td>Job power</td>
<td>Variable</td>
<td>Measured in hardware</td>
</tr>
<tr>
<td>Job performance</td>
<td>Variable with PStates</td>
<td>Measured in hardware</td>
</tr>
<tr>
<td>Frequency/PStates</td>
<td>P0 (1900MHz) - P4 (1100MHz)</td>
<td>Product data sheet [Optb]</td>
</tr>
<tr>
<td>Temperature limit</td>
<td>95°C</td>
<td>Typical</td>
</tr>
<tr>
<td>Pstate change interval</td>
<td>1msec</td>
<td>From [PMA+13]</td>
</tr>
<tr>
<td>Power management</td>
<td>Highest PState allowed under 95°C</td>
<td>Typical for responsive systems [PMA+13] [Jia]</td>
</tr>
<tr>
<td>On-Chip thermal time constant</td>
<td>5msec</td>
<td>Typical</td>
</tr>
<tr>
<td>Socket thermal time constant</td>
<td>30 seconds</td>
<td>From [Jia]</td>
</tr>
<tr>
<td>Server inlet temperature</td>
<td>18°C</td>
<td>Typical</td>
</tr>
<tr>
<td>Server total airflow</td>
<td>400CFM</td>
<td>Calculated from fan data [Act08]</td>
</tr>
<tr>
<td>Airflow at sockets</td>
<td>6.35CFM</td>
<td>Estimated using Ansys Icepak model [SAH+15a]</td>
</tr>
<tr>
<td>Ambient temperature</td>
<td>Variable</td>
<td>Estimated using Ansys Icepak model [SAH+15a]</td>
</tr>
<tr>
<td>$R_{\text{int}}$</td>
<td>0.205 Celsius/Watt</td>
<td>Calculated using Hotspot [SSB-03]</td>
</tr>
<tr>
<td>$R_{Ea}$ 18-fin</td>
<td>1.578 Celsius/Watt</td>
<td>Calculated using Hotspot [SSB-03]</td>
</tr>
<tr>
<td>$R_{Ea}$ 30-fin</td>
<td>1.056 Celsius/Watt</td>
<td>Calculated using Hotspot [SSB-03]</td>
</tr>
<tr>
<td>$\theta_{(\text{Power}, 18 - f in)}$</td>
<td>4.41 - Power x 0.0896</td>
<td>Modeled</td>
</tr>
<tr>
<td>$\theta_{(\text{Power}, 30 - f in)}$</td>
<td>4.45 - Power x 0.0916</td>
<td>Modeled</td>
</tr>
<tr>
<td>Simulation time</td>
<td>30 minutes of server time</td>
<td>Simulate at least 10M jobs</td>
</tr>
</tbody>
</table>

used policy in consumer systems that emphasize responsiveness [PMA+13]. The power manager runs every 1 msec.

The power manager also implements power gating for idle sockets. At every run of the power manager, it checks for sockets that were idle completely during the last 1 msec and power gates them instantaneously. We assume that power gated sockets still consume 10% of TDP power. Once a job is complete, its performance is recorded by calculating the time it took to finish the job. Overall performance for each job allocation scheme is calculated by finding the cumulative time the system took to complete all jobs.

5.4 Scheduling Techniques

The uni-directional flow of air from one end of the server to the other is the primary factor leading to inter-socket temperature variance. In such systems, hot air that has already absorbed heat from one socket ends up flowing over a downstream socket causing different sockets to
Thermal coupling is fairly well-studied in existing computing systems. Thermal coupling exists among cores in a multi-core system \cite{HSS08}, between the cpu and the gpu in a heterogeneous multi-core system \cite{PMA13}, between cores across layers in a 3D stacked processor \cite{PL07}, and between sockets and DRAM in server systems \cite{ANR12}. At the data-center level, thermal coupling occurs vertically among servers in a rack \cite{CKS07} and across regions of a data-center \cite{MCRS05,SBP05}.

However, the problem of thermal coupling for density optimized servers has different properties. There are three main differences. First, thermal coupling in this context is primarily uni-directional because of the air-flow direction. Second, thermal coupling is unavoidable, at least amongst certain sockets, due to their physical proximity. Last, as discussed before, the degree of coupling is fairly severe.

Because of these differences, traditional scheduling techniques that have been used to manage thermal behavior may not be sufficient. In this section, we will describe state-of-the-art work in scheduling techniques. Next, we analyze existing techniques in context of the HPE Moonshot. Last, we propose a new scheduler that improves existing schemes.

### 5.4.1 Existing Scheduling Techniques

First, we examine existing scheduling techniques used to mitigate thermal effects at the data-center level.

**Coolest First (CF).** The CF policy \cite{MCRS05,TGSC06,WvLD09} assigns jobs to the coldest compute elements in the data-center in order to add heat to cool areas and remove it from warm areas. We use instantaneous socket temperatures within our dense server as a metric to implement this scheme. This temperature is maintained within our scheduler and updated every time step. We also implement the Hottest First (HF) scheme. HF is the exactly opposite policy to CF and schedules more work on the warmer areas of the system, amongst the idle sockets.
Minimize Heat Recirculation (MinHR). The MinHR \cite{MCRS05} policy assigns jobs so as to minimize the impact of heat recirculation in the data-center. The implementation of this scheme involves running (offline) reference workloads at different servers and measuring temperature throughout the data-center to calculate heat recirculation factors. Others have proposed better heat transfer modeling techniques \cite{TGV07} \cite{TGV08} to implement similar policies. We implement this scheme by assigning power to sockets and measuring thermal coupling throughout the server to make a fixed heat-transfer map of the dense server. At run-time, the scheduler uses this map and assigns jobs to the idle socket that causes the least thermal coupling in the system.

Random. The Random scheduling policy \cite{MCRS05} \cite{TGSC06} assigns jobs randomly across idle components in order to approach the behavior of uniform power consumption and thermals.

Next, we examine thermal scheduling strategies previously proposed at the chip level.

Coolest Neighbors (CN). The CN \cite{CRWG08} policy is also a variant of the CF policy. It considers temperature of each component as well as its neighbors’ temperatures to account for on-chip lateral heat-transfer. It assigns jobs to locations that have the coolest neighbors.

Balanced. The Balanced scheduling policy reduces temperature variance by scheduling work to maintain a uniform temperature profile at compute elements \cite{CRWG08} \cite{CSTSR09}. The policy works by scheduling work away from hot spot locations. We implement this policy by scheduling work furthest away from the hottest point in the server.

Balanced Locations (Balanced-L). The Balanced-L \cite{CSTSR09} scheduling policy assigns work to locations that are expected to be the coolest based on their location (e.g. cores on the edges). We implement this policy by giving preference to sockets that are closest to the air-inlets.

Adaptive-Random (A-Random). The A-Random \cite{CRWG08} policy is a variant of the CF policy and considers temperature history along with the current temperature. Amongst the
components with lowest temperatures, the policy chooses randomly from the ones with lowest historical temperature in order to weed out locations that are consistently hot.

**Predictive.** The Predictive [YLK08] [AR09] scheduling policy first calculates the future temperature of a socket if the job were to be scheduled on it. Based on the temperature it predicts the frequency at which the socket can run the job and picks the location that can run the job the fastest.

![Figure 5.11: Average runtime expansion vs. CF for existing thermal aware scheduling mechanisms at 30% and 70% load for the Computation workload (lower is better).](image)

**5.4.2 Analysis**

Figure 5.11 shows a performance comparison across all existing schemes for the Computation workload at two different load levels. Computation is the highest powered workload and also the most sensitive to frequency changes. We measure average run-time expansion across all job completions and baseline the results versus the CF scheme.

We see that at low load, the CF scheme works fairly well and most of the other schemes have similar or worse performance than CF. Only the Predictive scheme is able to significantly improve performance. However, at the higher load, different schemes produce different results. HF and MinHR are the worst performing schemes at the lower load level, but at the higher load
Figure 5.12: Organization of zones in the HPE Moonshot. Each row consists of 3 cartridges (12 sockets) and 6 zones. There are a total of 15 rows in the system.

MinHR and HF become the best schemes, while Predictive loses its advantages.

Before we explain the reasons for these results, we will define zones in relation to the HPE Moonshot system. As shown in figure 5.12, the Moonshot system has three cartridges in series, each like the cartridge in figure 5.2. We divide the HPE Moonshot into 6 zones with each cartridge consisting of two thermally coupled zones. We label the zones 1 - 6. The odd zones have a 18-fin heat sink and the even zones have a 30-fin heat sink. Air flows through the system from zone 1 to zone 6. This creates asymmetric thermal coupling, not just because of the different heat sinks, but because sockets within the same cartridge are only 1.6 inches apart, while adjacent sockets between cartridges (e.g., zones 2 and 3) are about 3 inches apart.

Figure 5.13 present insights into the behavior of existing schemes for the HPE Moonshot system. It shows two metrics for three different locations of the server. Frequency is the average frequency at which the sockets operate relative to the highest possible frequency of operation (1900MHz). Workdone is the relative amount of work performed within a specific location as a proportion of overall work. Both these metrics are calculated for the front zones 1-3, back half
zones 4-6, and for even zones with the better heat sink.

At 30% load, less than half of the system is sufficient to complete all of the work. As seen in figure 5.13 (a), all schemes except Random, HF and MinHR allocate most of the work on the front half of the server and are able to sustain high frequency of operation including being in boost states for a considerable amount of time. In this case, using the back half of the server hardly provides any advantage – we can clump jobs near the inlet and leave the downwind sockets idle, minimizing the effect of coupling. The result is that Random, HF, and MinHR, which don’t do that, suffer.

CN schedules jobs primarily at zones 1, 3 and 5 in order to choose sockets that are cool as well as have cool neighbors and hence ends up with lower performance vs. the other schemes
that schedule at the front part of the server. Predictive is the best scheme at 30% load as it is able to choose the fastest socket to run the current job dynamically. As Figure 5.13 (a) shows, Predictive performs almost 80% of its work in the front half of the server but almost 50% of the work on even zones. Since there is only one even zone in the front half of the server, Predictive is performing most of its work on zone 2, an even zone with the better heat sink. Hence by combining the best from both scenarios (choosing the front part of the server and also the part of the server with the best heat sink), Predictive can outperform all other schemes.

At higher loads, however, we can no longer assume downwind sockets will be idle. At 70% load, we are using the back half more heavily, for most schemes, and the frequency of the back half is more impacted. Schemes like HF and MinHR perform more work at the back of the server and also end up performing more work at even zones, as the back of the server has two out of the three even zones.

An added advantage of packing more jobs in the back is that the front sockets can run at higher frequency as they have less work and hence can sustain boost longer for whatever work they perform. As a result, at 70% load, schemes like MinHR and HF outperform schemes that over-schedule in the front part of the server. Predictive offers no advantage as it schedules work to the coolest areas without regard for thermal coupling and ends up heating both the front and back of the server. In-fact, Random performs better than most other schemes (except MinHR and HF) as it distributes more jobs towards the back of the server than front loading schemes.

This analysis shows all existing scheduling policies have considerable drawbacks. CN, Random, MinHR and HF leave performance on the table at lower loads by not running at the fastest locations. At the higher load levels, CF, Balanced-L, A-Random and Predictive do not account for thermal coupling or the heterogeneous structure of heat sinks in the server. In the following section, we will propose and evaluate a new scheme combines the best features of existing schemes to provide better performance.
5.4.3 Proposed Scheme

As seen from the analysis in the previous section, a good scheduling strategy for dense servers balances two factors. First, it should consider the speed at which a job can run at different locations in the server by accounting for ambient temperatures and heat sink properties. Second, it should consider if scheduling at current location degrades performance of other down wind locations. The performance degradation of other sockets should be balanced with running the job faster at the socket where the job is to be scheduled. That is, the scheduler should consider the system holistically in making decisions and not just optimize point by point.

Based on these factors, we propose a new scheduling policy named **Coupling Predictor** (CP). The CP algorithm extends the Predictive [YLK08] [AR09] algorithm by taking into account thermal coupling effects. CP predicts the performance of not only the socket where the job is scheduled but also the performance of all other sockets that are downstream to this socket. It chooses the socket to schedule that results in overall benefit. Consider two cases, first, a socket that can run the job at 1700MHz but slows down two other downstream sockets total by 300 MHz, and second, a socket that can run the current job at 1600MHz but does not slow down any other sockets. Given such a choice, CP picks the second socket to schedule the job on since it results in overall benefit.

**Mechanics.** At every time interval, the scheduler checks if any jobs have arrived and scheduling decisions are pending. If there are jobs that need scheduling, the scheduler first picks a row of cartridges with idle sockets at random and then evaluates candidates within that row.

Within the selected row of cartridges, the scheduler first finds a list of idle sockets. Next, for each idle socket, it assumes that the job is scheduled on to it, and estimates an initial chip temperature using equation 5.1. It then updates power by compensating for temperature dependent leakage and predicts final chip temperature, again using equation 5.1. The scheduler then estimates the highest frequency of operation that keeps the estimated chip temperature less than the temperature limit and saves this value. In addition, based on the ambient temperature
model of the system, it uses a table lookup to estimate downwind socket ambient temperature. Then again using equation 5.1 and assuming the downwind sockets continue to run the same jobs they are running, it predicts frequencies of each downwind socket.

Note that we have sought to keep the scheduler very simple. We use a simple linear coupling model, rather than the complex models we use to evaluate this research. We do not account for varying application slowdowns wrt frequency, etc. Thus, the gains we show here could be improved with more accurate accounting of these effects, but at a cost of a more computationally expensive scheduler.

5.5 Results

In this section we will evaluate the proposed CP scheme against other schedulers. Figure 5.14 shows performance results at different load levels for all of our benchmarks.

As previously discussed, Predictive performed the best amongst existing schemes at low load values. For load values in the 10% – 30% range, CP outperforms or matches Predictive across all benchmarks. For Computation, at 30% load, CP is better than Predictive by about 2%. As compared to other existing schemes, CP provides a 5% – 15% gain over MinHR and about 3% – 7% over other schemes. As load increases to the 50% – 60% range, Predictive loses its advantage and at higher loads CP is consistently better for the Computation workload.

At low load values, Storage and GP benchmarks both show good performance across schemes vs. CF (note that these graphs are normalized to CF – so CF is represented by the 1.0 line). CF packs almost all the work in the first zone. This may lead to throttling of the first zone while scheduling at better heat sinks downwind proves to be advantageous for other schemes. This benefit disappears as load and consequently heat in the system increases. As the load approaches 30% – 40% for the GP workload, we begin to see similar behavior as we saw for 10% – 20% load in the Computation workload. Storage is insensitive to frequency as well as low power, and hence
we see rather muted behavior across all schemes for Storage, except in a few cases such as 10% load.

The GP workload shows the most gain at low loads for the CP scheme. GP has lower benchmark power and almost comparable frequency sensitivity as compared to the Computation workload. Hence it exhibits slightly more opportunity to optimize at lower loads where it sees less throttling than Computation. Both Predictive and CP capitalize on GP and provide gains of about 8% on average vs. CF at low loads. Overall, at low loads, CP out performs CF by 3% to 8% and matches the performance of Predictive.

Mid load values (40% load – 60% load) exhibit some interesting behavior. At these load values CP has to continuously choose between optimizing for a single socket vs. account for thermal coupling. CP is able to make the right decisions in almost all cases except for 50% and 60% load in the GP benchmark where Predictive is about 2% better.

At high load values (70% load to 100% load), HF and MinHR perform well. As seen from figure 5.14, at high load levels, the CP scheme is able to match the performance of the HF and MinHR schemes in most cases. This is because CP accounts for the effects of job scheduling on downstream thermally coupled sockets. Computation workload has the highest power, most throttling at high loads, and highest frequency sensitivity. Hence, it presents the highest opportunity to optimize. The CP scheme out-performs CF by 8.5% for Computation on average across high loads and by 6.5% across all load levels. Benefits can be as high as 17% (Computation 80% load). CP also improves the performance of GP by 6% over CF. Storage again sees muted gain of about 2.5%. CP performs significantly better than Predictive at high load values with gains between 2% to 9% across different workloads and as high as 17% (Computation 80% load).

While HF and MinHR exhibit poor performance at low loads for Computation, their relative behavior vs. CP improves for GP and Storage workloads. CP continues to beat HF and MinHR but with decreasing margins. At high load values, the relative differences between
MinHR and CP reduce where power and frequency sensitivity drops. However, CP is able to compete or beat other schemes that improve performance over certain localized load ranges. Such adaptive and load agnostic behavior is important for server systems where system load can change constantly based on user demand.

If we are to consider averaged performance across all load values, CP outperforms all other schemes by at least 5.5% for Computation, 3% for GP and 1.5% for Storage. CP outperforms CF by 6.5% for Computation and GP and about 2.5% for Storage. If we consider individual load values, CP may outperform CF as much as 17% for Computation, 10% for GP and 5% for Storage. We observe that no existing scheme provide consistent performance across all load levels. Existing work tends to optimize at single points such as socket level frequency, or only minimizing heat recirculation which under-performs at certain loads in dense servers. The proposed CP algorithm not only improves performance but also provides consistent performance.

CP works better than existing schemes because it considers inter-socket thermal coupling and carefully weighs scheduling effects on other sockets. Also, decisions are made by evaluating the potential for throttling at each job arrival, allowing it to make decisions at a finer granularity than just considering load as a proxy for thermals.

Figure 5.15 shows normalized ED² product values across different loads and schemes for all of our workloads. For Computation, the ED² product drops to as low as 0.7× at 80% load. For GP and Storage, the ED² product drops to as low as 0.8× and 0.85× respectively. In general, the ED² product of CP tracks that of Predictive at low load values and MinHR at high load values. These results show that CP also matches the energy behavior of Predictive and MinHR at different load values. The CP scheme buys us the performance of these schemes but imposes no extra energy penalties.
5.6 Conclusion

This research provides a comprehensive analysis of intra-server thermals for emerging density optimized server systems. It shows that existing scheduling algorithms perform sub-optimally across the spectrum of load levels as they do not account for inter-socket thermal coupling effects. We demonstrate new scheduling techniques that account for heat transfer amongst sockets and resulting thermal throttling. The proposed mechanisms provide 2.5% to 6.5% performance improvements across various workloads and as much as 17% over traditional temperature-aware schedulers for computation-heavy workloads.

Acknowledgements

Chapter [5] in part, is currently being prepared for submission for publication of the material. Arora, Manish; Skach, Matt; An, Xudong; Hung, Wei; Mars, Jason; Tang, Lingjia; Tullsen, Dean, Understanding the Impact of Socket Density In Density Optimized Servers. The dissertation author was the primary investigator and author of this material.
Figure 5.14: Performance vs. CF baseline for various schemes at various load values for different benchmarks (a) Computation, (b) GP, and (c) Storage.
Figure 5.15: ED2 vs. CF baseline for various schemes at various load values for different benchmarks (a) Computation, (b) GP, and (c) Storage.
Chapter 6

Concluding Remarks

The number of programmable cores that are available in systems continue to increase with advances in device scaling, system integration, and iterative design improvements. Today, systems are not just integrating more cores but also integrating a variety of different types of processing cores. This has lead to mainstream adoption of dense heterogeneous computing systems. One typical methodology for heterogeneous system design is to comprise systems by using parts of homogeneous systems, and then performing density optimizations with replication.

This dissertation shows that such a design methodology is “heterogeneous system oblivious”, as well as “density oblivious”. The components of the system are neither optimized for the heterogeneous system they would become a part of, nor are they cognizant that many other similar components may exist in the system. This dissertation proposes two improvements to the design methodology of dense heterogeneous systems.

First, it proposes a heterogeneity aware approach to designing heterogeneous architectures. A heterogeneity aware methodology designs components that are cognizant of their role in the target system and are specifically optimized for that role. This enables efficiency and performance as components are better optimized for workloads that would run on them. Second, it proposes a density aware approach to design. In the proposed methodology, components of the system are
cognizant that other components exists, and understand how they might be affected or interact with other components. This enables efficiency as components can better account for interactions and mitigate potential sources of performance loss.

Specifically, this thesis addresses three specific aspects that introduce inefficiency in “heterogeneous system oblivious” and “density oblivious” designs. This thesis proposes improvements to CPU architecture, new CPU idle power management techniques for heterogeneous systems, and improvements to job scheduling in dense server systems.

As GPUs become more heavily integrated into the processor, they inherit computations that have been traditionally executed on the CPU. This changes the nature of the computation that remains on the CPU. We demonstrate that even when significant portions of the original code are offloaded to the GPU, the CPU is still frequently performance critical. We show that the code the CPU is running is different than before GPU offloading along several dimensions. ILP becomes harder to find. Loads become significantly more difficult to prefetch. Store addresses become more difficult as well. Post-GPU code places significantly higher pressure on the branch predictor. We also see a decrease in the importance of vector instructions and the ability to exploit multiple cores. Hence we show that CPU-GPU integration requires us to perform a heterogeneity aware redesign of CPU architecture.

In this thesis, we show that a large number of important consumer computing applications have significant idle time while the application is “running.” The inclusion of general purpose programmable GPUs is expected to further increase the appearance of rapid idleness even in traditional compute intensive application domains. Our research shows that multi-core idle power management schemes are not suited to short idle events and work poorly for heterogeneous CPU-GPU systems. As overall energy consumption is heavily impacted by idle power, we show the need to develop better idle power management mechanisms for heterogeneous systems.

We show that there is no fixed break-even point in core power gating because of the effects of cache dirtiness. We evaluate existing state-of-the-art systems, and demonstrates new techniques
to predict when to enter the C6 power gated state. Proposed heterogeneity aware schemes that are tuned better for an era of short idleness, provide average energy reduction exceeding 8% over existing schemes, and sacrifice almost no performance.

In this dissertation, we investigated the degree of thermal interaction that occurs between sockets in a dense server caused by the sharing of the cooling system. We demonstrate that thermal interactions in these designs are primarily uni-directional and fairly severe. At the same time, these interactions are unavoidable, at least amongst certain sockets because of how the cooling system works. We show that existing “density oblivious” scheduling algorithms perform sub-optimally as they do not account for inter-socket thermal coupling effects. This dissertation proposes better scheduling job algorithms for dense servers that suffer from such thermal interactions.

Using a computational fluid dynamics model, this research quantifies temperature heterogeneity among sockets due to inter-socket thermal coupling. Based on our thermal analysis, we propose a new scheduling policy named CouplingPredictor (CP). The CP algorithm extends the predictive job scheduling algorithms by taking into account thermal coupling effects. CP predicts the performance of not only the socket where the job is scheduled but also the performance of all other sockets that are downstream to this socket. CP chooses sockets for scheduling that result in overall benefit and significantly improves system performance. The proposed mechanisms provide 2.5% to 6.5% performance improvements across various workloads and as much as 17% over traditional temperature-aware schedulers for computation-heavy workloads.
Bibliography


[BIO12] BIOS and Kernel Developer’s Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, Jan 2012.


Dell-Blogs: The Time is Right - Introducing 'Copper'. [Cop]

Coefficient of Variation. [COV]


Dell, HP, Cisco Look Beyond The Traditional Blade/Rack Server. [CRN]


High Density and Performance Servers. [Dat]

Dell and the Value of Density Whitepaper. [Dela]

Dell FX converged architecture components. [Delb]


113


[LKC+10] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU
vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In ISCA, 2010.


http://nvidia.com/content/pdf/fermi_white_papers/.


