UNIVERSITY OF CALIFORNIA, SAN DIEGO

Software Management of Memory Subsystem Contention on Multicore Systems

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

in

Computer Science

by

Alexander Dodd Breslow

Committee in charge:

Professor Dean M. Tullsen, Chair
Professor Scott B. Baden
Professor Michael L. Norman
Professor Alex C. Snoeren
Proessor Geoffrey M. Voelker

2016

The Dissertation of Alexander Dodd Breslow is approved and is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2016

DEDICATION

To my parents. It has been through your kindness, encouragement, and tireless support that I have been able to realize my potential as a person and professional. I dedicate this dissertation to you.

EPIGRAPH

We can know only that we know nothing.
And that is the highest degree of human wisdom...

Leo Tolstoy

All good things must come to an end.

Geoffrey Chaucer

The excellency of every art is its intensity,
capable of making all disagreeable evaporate.

John Keats

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First off, I would like to acknowledge the several advisors that I have had over the course of my PhD. They are in order Professors Allan Snavely, Lingjia Tang and Jason Mars, Scott Baden, and Dean Tullsen. To you I can offer only but the highest praise. I feel exceptionally fortunate to have had the opportunity to learn from so many great researchers whose unique talents each lifted me to new heights as a researcher. I am better off due to their tireless efforts as mentors, advocates, and friends.

In particular, I would like to thank my advisor Professor Dean Tullsen for looking out for my interests and for his mentorship. Dean is one of the kindest people that I have had the opportunity to know and also one of the very smartest. But that is not all; he is also a great listener, patient, and a consumate professional, often going far beyond what is expected of him. I feel incredibly privileged to have gotten the opportunity to work with him. Without his help, I know that my journey would have been considerably more difficult and its success uncertain.

In addition to my faculty mentors, I also would like to thank Dr. Laura Carrington and Dr. Ananta Tiwari of the Performance Modeling and Characterization Lab at San Diego Supercomputer Center. Laura funded the first 3 years of my PhD and constantly pushed me to improve the quality of my research through her healthy skepticism and critiques of my ideas. Her detail-oriented feedback made me a much better scientist and she enabled much of my early success. Both her and AT supported me during the tough moments of my PhD journey, and it was in large part through their help that I was able to produce ”Enabling Fair Pricing on HPC Systems with Node Sharing,” which garnered both best paper and best student paper award nominations at SC’13 and was ranked first in its track out of dozens of submissions. To AT, I cannot thank you enough for your support and help throughout the research process. You were awesome!

I also thank Dr. Michael Laurenzano and Professor Leo Porter. Working with you

both at PMaC was an absolute pleasure. Your shared enthusiasm for science, computer science education, and fitness gave me a strong sense of community, and I appreciated the many hours that you took to share your wisdom with me. Your passion for life and work inspired me each day to make the most of my opportunity.

In addition, I thank many of the excellent staff members at SDSC. When I first came to UCSD in the summer of 2011, they were incredibly patient with my questions. Thank-you to Dr. Pietro Cicotti, Dr. Bob Sinkovits, and Ken Yoshimoto for your help with using the computing resources at SDSC and your friendly disposition. In addition, I thank Nicole Wolter and Mahidar Tatineni. They humored many of my more unusual requests for configuring supercomputer software, and they in conjunction with Shawn Strande and Professor Michael Norman made the core experiments from my first two papers possible.

I also thank Dr. Martin Schulz at Lawrence Livermore National Laboratory. Your feedback, support, and kindness were deeply appreciated.

At Advanced Micro Devices, I would like to thank my primary mentors Drs. Dong Ping Zhang and Nuwan Jayasena. Your unwaivering belief, kind words, and friendship has made my work experience at AMD Research an absolute pleasure. Thank-you for your help in our joint efforts to pursue impactful research. In addition, I thank Dr. Joe Greathouse. Your patience explaining nuanced architectural and software concepts, help with tools, and paper critiques has made much of our recent research possible. I also want to thank Drs. Amin Farmahini-Farahani and Onur Kayıran for discussing research directions and humoring some of my stranger architectural ideas. At AMD, I have been privileged to work in an environment with great folks including Vinay Agarwala, Professor Hyesoon Kim, Dr. Mike Chu, Dr. Ashwin Aji, Dr. Shrikanth Ganapathy, Dr. Lifan Xu, Mateja Putic, Robert Searles, Hyojong Kim, Dr. Paula Aguilera, Clint Lestourgen, Ashutosh Pattnaik, and countless others.

At UCSD, I want to thank the following professors and administrators that in various moments throughout my graduate school career came to my aid: Professor Geoff Voelker, Professor Michael Norman, Professor Alex Snoeren, Professor Tajana Rosing, Julie Conner, and Lynne Keith-McMullin. For anyone else I forgot or who helped me, unbeknownst to me, you have my gratitude.

I also want to thank my friends and partners in crime Zak Murez and Matt Elkherj. Living with you provided constant entertainment and got me through some of the more mundane moments of grad school.

Other grad students also made my PhD experience memorable. I would like to thank Tan Nguyen, Mohammed Sourouri, Natalie Larson, Alden King, Ashish Venkat, Hung-Wei Tseng, Andreas Prodromou, Russell Reas, Brian Lunt, Grant Van Horn, Sam Kwak, Rick Strong, Prudhvi Tella, Zhaomo Yang, Tatenda Chipeperekwa, Jesper Lindstrøm Nielsen, Ankit Baid, Didem Unat and Hailong Yang. You made coming to work fun and enjoyable.

I also thank my professors at Swarthmore College. They prepared me for graduate school in a way few other programs would have been able. I give my sincere thanks to Professors Andrew Danner, Tia Newhall, Lisa Meeden, Doug Turnbull, Rich Wicentowski, Charles Kelemen, and Charlie Garrod for your devotion to the highest standards in pedagogy. Your example continues to inspire me.

To my committee, thank you for taking time out of your busy schedules to serve on my committee. It is an honor to have such a distinguished and versatile group.

To my parents, thank you for your support. You had a vision for me long before I was aware of it, and it has been through your gentle yet guiding hands, tireless support, and unwaivering love that I have achieved this and other successes in life.

Chapter 2, in full, is a reprint of the material as it appears in Darren Kerbyson, Georg Hager, Gerhard Wellein, Abhinav Vishnu, editors, Concurrency and Computation:

Practice and Experience 2013. Breslow, Alexander D.; Porter, Leo, Tiwari, Ananta, Laurenzano, Michael L., Carrington, Laura N., Tullsen, Dean M., Snavely, Allan E., Wiley Blackwell, December 2013. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in Proceedings The International Conference for High Performance Computing, Networking, Storage and Analysis 2013. Breslow, Alexander D.; Tiwari, Ananta, Schulz, Martin, Carrington, Laura N., Tang, Lingjia, Mars, Jason, November 2013. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part is currently being prepared for submission for publication of the material. Breslow, Alexander D.; Zhang, Dong Ping, Greathouse, Joseph L., Jayasena, Nuwan, Tullsen, Dean M. The dissertation author was the primary investigator and author of this material.

Chapter 5, in full, is a reprint of the material as it appears in Proceedings of the USENIX Annual Technical Conference 2016. Breslow, Alexander D.; Zhang, Dong Ping, Greathouse, Joseph L., Jayasena, Nuwan, Tullsen, Dean M., USENIX Organization, June 2016. The dissertation author was the primary investigator and author of this paper.

VITA

2011	Bachelor of Arts in Computer Science Swarthmore College
2011–2014	Research Assistant University of California, San Diego
2014	Teaching Assistant University of California, San Diego
2014	Master of Science in Computer Science University of California, San Diego
2015–2016	Postgrad Researcher Advanced Micro Devices Research
2016	Doctor of Philosophy in Computer Science University of California, San Diego

ABSTRACT OF THE DISSERTATION

Software Management of Memory Subsystem Contention on Multicore Systems

by

Alexander Dodd Breslow

Doctor of Philosophy in Computer Science

University of California, San Diego, 2016

Professor Dean M. Tullsen, Chair

The multicore era has initiated a move to ubiquitous parallelization of software. In the process, cores have scaled out but the memory subsystem resources have not kept up. Memory subsystem contention within and between applications makes it challenging to extract performance scaling that matches the increase in the number of cores. This dissertation explores the diagnosis of memory subsystem contention, identifies associated performance and energy efficiency opportunities, and suggests techniques and optimizations to both precisely measure and reduce the contention. The dissertation begins by exploring contention within a single and between multiple, large-

scale, distributed scientific applications and moves to exploring the impact of memory subsystem contention on graphics processing units, accelerators that are seeing increasing usage in both commercial data centers and scientific clusters. The findings of the studies demonstrate that memory subsystem contention is a serious impediment to achieving high performance and energy efficiency but also that relatively simple techniques that control job placement, resource sharing, tuning of parallelism, and algorithmic optimizations at the application level provide significant opportunities to improve performance and energy efficiency.

The dissertation comprises four distinct works. (1) It begins by quantifying the performance and energy efficiency opportunities afforded by co-scheduling large-scale distributed scientific applications within a supercomputer. (2) From there, it studies the design of a prototype system for dynamically quantifying inter-application interference between co-located supercomputer jobs and uses those estimates to reform the accounting system to more fairly reflect end-user utility. (3) Next, it explores performance and energy scaling of analytic database workloads on graphics processors and finds that disabling whole compute units can reduce both query execution time and energy by throttling back the number of threads at any instant that actively contend for the shared last level cache. (4) The dissertation ends by describing the Horton table, a hash table that accelerates in-memory data-intensive computing by more efficiently using hardware cache and memory bandwidth.

Chapter 1

Introduction

Recent trends in computing have seen a proliferation of multicore systems, from small embedded devices and smart phone processors up to the large multisocket multicore servers that are found in large-scale data centers. A key consequence of multicore systems is that to obtain high performance for a single application, serial legacy code has needed to undergo parallelization. In addition, even existing parallel programs from the era of multichip symmetric multiprocessor systems have undergone substantial revisions as the degree of hardware parallelism has scaled. While harnessing parallelism is one part of the story, that alone is often insufficient for crafting applications whose performance doubles with a doubling of cores and where per-thread performance is highly optimized.

There are many factors that determine the performance scaling of parallel programs on multicore hardware (e.g., synchronization primitives [127] and contention for IO peripherals [141, 161]). This dissertation focuses on a subset of problems that arise from interaction between parallel workloads and the memory subsystem of today's multicores. Research into software-memory interaction has seen renewed importance due to a surge in large-scale, distributed in-memory computing frameworks that have seen increasing adoption in both the commercial and scientific computing domains. In the enterprise domain, there is increasing emphasis on deriving real-time insights from large datasets [184]. These low-latency requirements have driven the creation of in-memory

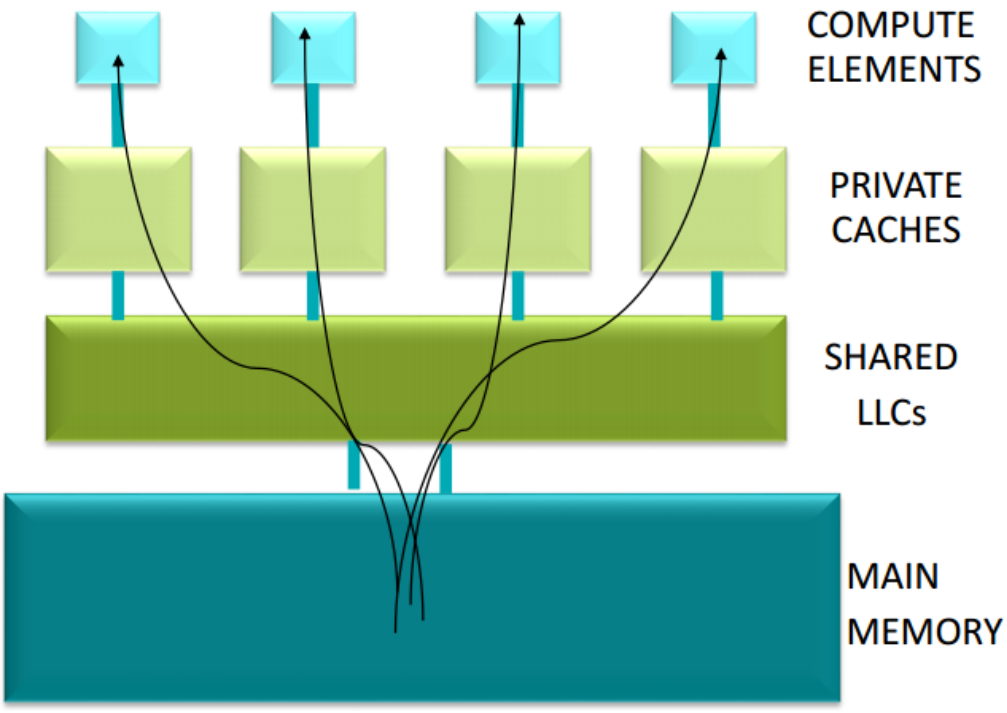


Figure 1.1. A high-level view of a chip multiprocessor. One or more narrow buses typically connect off-chip memory to the processor and often are a performance bottleneck when the last level cache suffers more misses per cycle than the memory buses have bandwidth to immediately service.

computing systems such as Spark [254], Storm [200], Tez [206], Dryad [120], and Naiad [178], and a move away from their predecessors that were designed and optimized with network and IO-bound hardware in mind [31, 78]. In the scientific domain, in-memory distributed computing remains ubiquitous. At the same time, continued scaling has become paramount to realizing the scientific potential afforded by exascale computing [47]. For both the commercial and scientific domains, understanding how software applications interact with the memory subsystem is crucial.

Figure 1.1 shows a high-level depiction of a multicore system. For the purposes of this discussion, we have simplified key elements to generalize between different architectures. At the top level are the processing elements that operate directly on fast registers. Subsequent levels in the memory hierarchy include one or more levels of caches

and then the main memory.

Often there are one or more levels of private cache (typically L1 and L2) that are exclusive to each processing element followed by a last level cache (LLC) that is shared among a plurality of processing elements. Caches that are higher in the memory hierarchy and closer to the processing elements most often have higher bandwidth, lower latency, but reduced capacity relative to caches closer to main memory like the LLC. In the ideal case, programs operate almost exclusively on registers and make loads that almost always hit in the private L1 caches. While there are a class of workloads that exhibit this behavior, there is also a growing important class of memory-intensive workloads that does not. Graph [43, 213, 256], in-memory data-stores [90, 96], and sparse linear algebra [104, 241] are examples of workloads that continue to proliferate in their application and utility to commercial and scientific computing, but which interact poorly with the cache hierarchies of today.

Whereas a ”well-behaved” application might achieve almost perfect performance scaling as it expands to run across more cores, in contrast, these applications often do not. Instead, they may plateau or perhaps suffer performance collapse even before all of the cores have been assigned application threads. Often, the root of this poor scaling is that although the cores have scaled, many of the buses, controllers, and caches have failed to keep up with the increase in cores. Worse yet, parallel algorithms often need to replicate or allocate additional space over their serial variants, further reducing the effectiveness of hardware caches as parallelism scales. Thus cache- and bandwidth-limited workloads often achieve severely reduced performance as a result of this inter-thread contention.

In this dissertation, we consider the role of software to reduce memory subsystem contention with the goal of improving execution time, increasing throughput, reducing energy, or a combination of all of the above. Software’s role in this domain is essential because it can greatly offset hardware’s limitations when executing parallel programs.

While some degree of poor scaling or performance may be inherent to the semantics of a program's core algorithms and lead to memory contention that is practically unsolvable due to the limitations of today's hardware, there are also significant performance and energy efficiency opportunities that arise when software is properly tuned to reduce memory subsystem contention. Managing these detrimental inter-thread interactions is of high importance because it can improve performance by more than 2x [58, 59]. These effects are often amplified on large-scale systems, especially when they lead to or worsen the severity of straggler tasks, tasks that lag behind others with which they synchronize. Stragglers often increase program critical path length and lead to resource underutilization by forcing other tasks to wait for their completion long after the other tasks have finished [28, 57, 58, 78].

To address the various challenges associated with memory subsystem contention, this dissertation is divided into several parts, which we detail in the text that follows.

- Chapter 2 explores the performance and energy benefits associated with co-locating distributed scientific applications across a shared set of supercomputer nodes.
- Chapter 3 proposes and evaluates a system architecture for online determination of contention between co-located workloads to drive fairer accounting on supercomputers with node sharing between jobs.
- Chapter 4 explores the scaling of in-memory database workloads on graphics processors and examines how disabling compute units can reduce memory subsystem contention and also improve performance and energy efficiency.
- Chapter 5 presents a hash table design known as a Horton table that is optimized to more efficiently use cache and memory bandwidth.
- Chapter 6 concludes.

Below we give a more in-depth summary of each of the main projects that comprise this dissertation.

1.1 Job Co-locations on Supercomputers

Memory contention is often a serious impediment for achieving ideal performance scaling of large-scale parallel applications on supercomputers [69]. Parallel tasks within a single application contend with one another for scarce resources such as cache capacity, cache bandwidth, interconnection networks, and off-chip bandwidth to main memory. Severe performance degradation due to inter-thread interference is in large part due to the common application traits in scientific computing workloads: the widespread use of single program multiple data semantics (tasks execute the same program but on different data) [240], tight synchronization between tasks, and large working sets that can easily put large amounts of pressure on shared caches and off-chip memory bandwidth. Although these types of applications may achieve near-optimal performance scaling across compute nodes, there are a number of applications where either single-thread performance, intra-node performance scaling, or a combination thereof is severely hurt by similar threads from the same application contending for the shared memory subsystem in a similar fashion and in relative lock-step. Taken together these behaviors frequently lead to cache thrashing of the LLC, saturation of off-chip memory bandwidth, and ultimately performance degradation. Further, the high levels of fine-grain synchronization that are present in many of the distributed scientific applications that use the Message Passing Interface (MPI) [105] and similar communication libraries can cause a straggler task that has been slowed down by memory subsystem contention to also slow down all tasks with which it synchronizes, provided that they collectively wait at shared synchronization points. At large scales, this effect can be quite pronounced [141]. Worse yet, MPI typically allocates large memory buffers to send and receive messages [70, 159], which

can further exacerbate the problem by adding to the existing memory contention that the computational part of the code generates.

To reduce this memory subsystem contention, we evaluate thread placements that aim to decrease the number of threads that actively create pressure on memory resources at any one time. We first leverage a variant of undersubscription where the total number of distributed tasks is fixed but tasks are spread out over more nodes, which we term *job spreading*. Job spreading reduces memory subsystem contention and some forms of network contention by reducing the number of tasks that contend for resources within a node. However, job spreading underutilizes compute resources by leaving half of the cores idle. This deficit motivates the need to explore an alternative policy to improve throughput, known as *job striping*, where two or more jobs of approximately equal size are assigned disjoint sets of cores on each compute socket across a shared set of compute nodes. Because the co-located applications typically stress hardware resources in distinct ways, they often reduce contention over the de facto practice of assigning each application its own set of compute nodes. Further, contentious phases of one application can fall out of sync with contentious phases of another [141]. Contrast this with assigning a set of nodes to a single, distributed SPMD application. Its tasks are all likely to stress the system resources in the same way, and contentious phases are less likely to fall out of sync with one another due to shared synchronization points. Leveraging job striping achieves much of the same benefits as job spreading but also improves net throughput by 12% on real applications and reduces energy use by a similar amount when processing a workload consisting of real, large-scale scientific applications [58].

1.2 Fair Pricing for Supercomputers with Node Sharing

After the study of Chapter 2, we consider challenges associated with delivering a supercomputing system where node sharing is practical. One of the largest road blocks is that resource contention between jobs that share nodes is variable [49, 58, 124, 207, 253]. This variability poses a challenge because depending on an application's sensitivity to differing types of resource contention and both the severity and types of contention that other jobs generate, a job's execution time can vary wildly. When this variation manifests as a significant increase in run time over the status quo, it can reduce the utility of the job to the user. Prior work demonstrates that end-user utility is integrally tied to the time for a job to complete [148]. Even though queuing delays in a typical supercomputer batch scheduling environment are many hours, job execution times are also often quite long. Contention that stretches a job's execution from 20 hours to 40 hours is significant and could lead to the cluster manager prematurely killing the job for exceeding its allocated time. Such events are serious because they impact a user's productivity, with fair reaching consequences like missing grant fulfilment or conference deadlines. While this example is an extreme, it illustrates the need to precisely quantify inter-application contention in co-located environments to be able to detect and mitigate these types of scenarios.

Fair accounting is also an integral issue when considering job co-locations in high-performance computing environments. On supercomputers, scientists use a bank of credits known as service units to pay to execute jobs. When jobs run, they gradually deplete a user's service units at a rate proportional to the product of their execution time and the number of servers that are utilized. Thus if one co-located job dilates the run time of another, the user of the slowed down job would pay a greater price than if their job had experienced no contention at all. Worse yet, jobs that are particularly sensitive

to shared resource contention are likely to suffer both a slow down and receive the greatest surcharge. These traits mean that the current pricing scheme is unfair, and rather, a revised pricing scheme that discounts users for lost utility from run time dilation is necessary for co-location on supercomputers to be practical. However, achieving a fairer accounting mechanism requires accurately measuring the contention that jobs suffer from degradation. Further, this measurement must be done with little to no additional resource consumption, which is a hard problem.

To combat these detractors, Chapter 3 presents POPPA a prototype system that precisely quantifies performance degradation that results from destructive interference due to shared resource contention within the memory subsystem. The POPPA system works by gathering measurements from hardware performance counters that track forward progress of application threads at regular intervals. To increase the insight gained from these samples, POPPA also selectively deschedules all applications but one that share a group of compute nodes. By comparing the values of the performance counters of an application when it is coscheduled and when its corunners are briefly paused, POPPA determines the precise degradation that an application suffers from sharing its compute nodes with other jobs. We then propose incentivizing users to run on a supercomputer with node sharing by discounting them proportionally to their application's reduction in performance. As an example, an application that takes 5/4 times longer to run due to inter-job contention would cause the user to be billed at 4/5 of the baseline price. POPPA achieves a mean prediction error of 4% and prices jobs at a level of fairness that approaches what an oracle with complete information achieves. At the same time, careful performance tuning allows it to maintain an overhead of 1% or less [59].

1.3 Black Box Performance and Energy Efficiency Optimizations for GPU-Accelerated Databases

Chapter 4 shifts focus to lessening the effects of memory subsystem contention on graphics processors when running analytic database workloads. The motivation of studying these workloads on GPUs comes from the fact that GPUs are seeing increasing use in large scale systems [24], both commercial and otherwise, due to their improved energy efficiency and performance [107, 150]. We focus on database workloads because 40% of all the expenditure in the server hardware market goes to procuring systems to run them, making them the single largest segment of the market [140]. In contrast to Chapters 2 and 3, which largely focus on job co-location and inter-application contention, we instead consider the effect that tuning the compute-to-memory-bandwidth ratio of a GPU that runs a single application, which executes thousands of parallel threads with data-intensive code, has on performance and energy efficiency. Much as multiple applications can contend with one another for resources on a traditional chip multiprocessor and degrade one another's performance, the thousands of in-flight threads from a single application running on a GPU can face similar degradation from shared resource contention [202].

Chapter 4 begins by asking a simple question: can current analytic database workloads fully utilize all of the compute units of a modern GPU? Posing this question is important given the historical significance of the memory wall [247] in in-memory database systems [25, 54]. Scaling the memory wall prompted 20-plus years of research within the high-performance database community on columnar databases and memory-conscious database operators and optimizations to thwart its effects [23, 53]. In light of the relative maturity of these optimizations coupled with a GPU's superior memory bandwidth and both its high degree of hardware multithreading and memory-level par-

allelism for latency hiding [205], a careful study is required to precisely quantify the degree that the memory wall impedes the performance scaling of these applications on today's GPUs.

Our study begins by showing that like CPUs, the memory subsystem still does very much curtail the performance of online analytical processing workloads on GPUs. By using a tool that allows us to dynamically modify the GPU's firmware, we incrementally disable the compute units (akin to clusters of cores) on the GPU and measure the change in performance from the baseline configuration where all the compute units are enabled. We find that for most of the workloads, near-optimal performance can be achieved when only about half or less of the compute units are enabled. When all compute units are enabled, the additional in-flight threads cause excessive pressure on the last level cache and contention for off-chip bandwidth. Instead, it is often preferable to have fewer threads and fewer compute units enabled. To further exploit this property, we explore employing coarse-grain power-gating of disabled compute units to reduce the static power that is present even when transistors do not actively switch [88, 136]. We demonstrate that tuning the active number of compute units can reduce query run time and energy by as much as 24% and 42%, respectively.

1.4 Fast Hash Tables for Data-Intensive Computing

Chapter 5 follows from some of the insights derived from Chapter 4, where part of the study consisted of characterizing the workload at the level of computational motifs [33], routines, and database operators [74, 75]. In particular, we found that routines employing hash tables accounted for a majority of execution time in a large minority of the queries. Further, in-memory hash tables are frequently used in key-value stores for supporting fast point queries [96, 115], which illustrates their ubiquity throughout the data management space, where they are an essential, practical, and interesting

primitive to consider for optimization. Upon examination of the state-of-the-art hash table design known as a bucketized cuckoo hash table [87], we found inefficiencies in terms of how it moves data within the memory hierarchy when probing the table. When performing a lookup on a key associated with a potential key-value pair within the table, the operation typically issues loads to between 1.5 to 2.0 cache lines from the table on average [90, 153, 203]. Given that hash tables often have poor temporal and spatial locality, once a hash table is too large to fit in private caches or for that matter cache at all, the performance quickly degrades because the majority of lookups trigger accessing cache lines from the level in the memory hierarchy that can adequately store the table. The further down in the memory hierarchy that is, the more limited bandwidth that is available and the more imperative it is to reduce the number of unique cache lines that are accessed per queried key. Thus, reducing the number of table-containing cache lines that are accessed per lookup query has a large impact on performance, especially when the table is quite large because it reduces the amount of data that needs to pass over the comparatively low-bandwidth and high-latency off-chip memory buses per query. Given that the prevalence of these tables is growing with the proliferation of data, studying ways of optimizing these memory-bound tables is imperative.

For our approach, rather than completely abandoning the prior art, which has a number of attractive properties (lookups that require examining at most 2 buckets [87], the ability to consistently fill the table between 95% to 99.99% full [87], and high performance when implemented with SIMD instructions such as AVX [190]), we instead retrofit the table to reduce its average lookup cost per key-value pair to close to 1 cache line (assuming 4 or more key-value pairs can fit in a single cache line, which is typical of columnar database workloads). The revised table inserts most key-value pairs using a single hash function and employs one of several alternate functions to map the item to a free bucket when the first function maps the item to a bucket that is already full. To

track remapped items, one of the cells of each bucket that overflows is converted into a special structure known as a *remap entry array*. The remap entry array permits all remapped elements to be located by accessing at most 2 buckets, and hence, 2 cache lines. It also accelerates negative lookups (lookups where an element is not found in the table) by acting as an approximate set membership data structure that filters lookups to subsequent buckets when a key's entry is not found in the bucket to which it was initially mapped. On a 95% full table, these optimizations reduce the expected lookup cost per key-value pair from 1.5 to 2.0 cache lines down to 1.06 to 1.18 cache lines, respectively. These optimizations improve lookup throughput on a performance-optimized GPU implementation by 5% to 95% depending on workload composition and table size. Although these workloads typically remain profoundly bandwidth-limited, they make much better use of the bandwidth that is available by cutting out extraneous accesses to buckets that do not hold the queried key [60].

Chapter 2

Evaluating the Benefits of Job Co-locations on Supercomputers

Recent studies into the feasibility of exascale computing have shown that exascale will present a series of unique challenges [44, 47, 66, 211]. To conquer these challenges, a reexamination of every level of the supercomputing infrastructure will be necessary, from the facility, to hardware, and up through all levels of the software stack. In this study, we show that the de facto supercomputing scheduling mechanism can be inefficient, and that there is a significant performance and energy efficiency opportunity with an alternative approach.

Most data center supercomputers are a collection of individual nodes or server blades that are connected to one another over a high speed network. Nodes are parallel computers in their own right, often containing multiple sockets and dozens of cores. Such supercomputers are inherently hierarchical and exhibit multiple levels of shared resources including on-chip buses and cache structures, on-node shared memory bandwidth and network interfaces, and the system-wide network interconnect. While shared structures help facilitate the function of a massively parallel machine, their presence necessitates conscientious orchestration of the sharing of these resources.

The prevailing approach to this coordination of resource sharing is simple: sched-

ule jobs from different users on disjoint sets of compute nodes. While this limits the potential chaos of an ungoverned scheduling environment where any user can monopolize the system, it also eliminates the advantages that cooperative sharing can offer. In particular, prior work on multiprogrammed workloads shows that system performance is significantly increased when heterogeneous threads are intelligently coscheduled on the same computational resources [49, 71, 139, 191, 214, 248]. Across a suite of different architectures, these studies find benefit in enforcing policies that facilitate fairer sharing of resources, which avoids thread starvation and accordingly increases throughput.

The reason colocating heterogeneous threads usually improves system performance is because each thread executes distinct code and thus taxes the system in largely different ways. In contrast, homogeneous threads execute nearly identical code and consequently often make very similar demands on the system. As such, they are more likely to impede one another's progress by competing over shared resources such as last level caches (LLC) and memory bandwidth. Avoiding these collisions on shared resources is key to realizing high aggregate system performance.

For the purposes of this study, we examine the performance opportunity provided by harnessing HPC workload heterogeneity at scale. Most large distributed scientific applications fall under the Single Program Multiple Data (SPMD) programming model. SPMD parallel programs are comprised of multiple identical or nearly identical tasks that operate on distinct data. Since they are similar, the colocation of these tasks on the same computing resources produces contention and hurts performance. Unfortunately, this is exactly what happens on present day supercomputing systems. Each application is given a private set of compute nodes. While this approach makes it easier to guarantee quality of service, it causes homogeneous tasks to be placed together, consequently increasing contention and accordingly decreasing throughput and energy efficiency.

For these reasons, we propose job striping, a process-to-core mapping technique

for supercomputing clusters. Job striping reduces contention by capitalizing on the heterogeneity in workloads found on today's supercomputers. On a job-striped set of compute nodes, half the cores of each processor run one parallel application and the other half run another. By interleaving two distinct jobs with one another, system contention is reduced.

To validate the efficacy of job striping, we conduct large-scale experiments on two complete racks (2048 cores) of the state-of-the-art Gordon supercomputer at SDSC. For the 1024 process NAS parallel benchmarks [35], job striping improves single application throughput by as much as 81%, and more than 26% on average. In addition, energy efficiency increases by as much as 52%, and 22% on average.

We also evaluate job striping on three very common scientific applications, GTC [158], LAMMPS [11], and MILC [13] at 1024 processes. On these applications, job striping increases mean throughput by 12% and energy efficiency by 11%. MILC improves the most from striping, with throughput improving by 23% and 32% when it is paired with GTC and LAMMPS, respectively.

This chapter also identifies the critical resources that benefit most from heterogeneity – the memory subsystem and the communication network. All applications whose performance improves from striping exhibit reduced contention for one or both of these resources when the workload is heterogeneous.

The primary contribution of this work is to demonstrate that colocating large scale HPC applications yields a performance benefit. Prior work on heterogeneous scheduling focuses almost exclusively on uni- and multi-threaded benchmarks or synthetic workloads run on a single server. In addition, it largely glosses over energy efficiency and network contention while using benchmark suites that are not representative of HPC workloads. This study, in contrast, demonstrates both the necessity for and the mechanisms to enable heterogeneous workload scheduling for real-world HPC environments.

Additional contributions of this study are as follows:

(1) We motivate job striping for HPC and describe a particular software solution.

(2) We present a new energy efficiency metric *Scaled Energy Efficiency* (SEE), which approximates the ratio of before and after power-delay products.

(3) We demonstrate that job striping significantly increases energy efficiency and throughput on real applications.

(4) We show that job striping achieves its performance gains by reducing contention on shared resources, through analysis of application profiles.

(5) We provide a simple mechanism to predict low performing application pairs.

(6) We show that pairing GTC with MILC, two applications commonly run at NERSC [30], improves energy efficiency by 13%.

2.1 Motivation

In this study, we motivate job striping by exploring an application's behavior when mapped in two commonly found processor affinities, one which we refer to as *compact* and the other which we refer to as *spread*. These terms are consistent with what is found in the literature on NUMA affinity optimizations [64] and in software and system user guides [20, 194]. The best way to explain these configurations is by an illustrative example.

Suppose we are given a distributed application A that is comprised of p single-threaded processes and a set of chip multiprocessors (CMPs) with c cores each. The *compact* configuration would assign CMPs to A such that each process is mapped to a single core with no cores left unassigned. In contrast, the *spread* configuration instead opts to use twice as many CMPs but to leave each socket half full. Thus *spread* uses $\frac{c}{2}$ cores per CMP versus the c that the compact configuration uses.

The spread affinity specification often offers a significant performance advantage

over compact because each process experiences less cache contention from its neighbors. In the compact configuration, the shared last level cache (LLC) and memory bandwidth are split among c processes. However, in the spread configuration they are divided between $\frac{c}{2}$ processes. This means that the mean available memory bandwidth and LLC space per core are 2x higher in the spread configuration. Since both of these resources are critical for achieving high performance, the spread configuration almost always outperforms compact. While this is likely a win for raw performance, job spreading has its limitations.

In real supercomputing systems, users are charged for the combined number of CPU hours that their application uses. The price of execution is derived by a simple formula where cost is the product of the number of nodes assigned to a job (N), the number of cores per node (P), the total time the job ran (T), and a rate constant (k) (shown below).

$$C = k * P * N * T \tag{2.1}$$

Consequently, if a user spreads their job, they are charged for the cores that they leave unassigned. While this would not be a problem if spread offered a 2x increase in performance, in many cases it does not. As such, users have a monetary incentive to maximize their application's performance per CPU hour, not per core. Thus users commonly request only enough nodes such that every process has a core to run on (compact). While this makes sense from the user's economic perspective, this may not yield the best overall system performance. For many scientific computing applications, the memory bandwidth requirements per core are quite high, and when the processor is fully occupied with multiple identical threads or processes, memory bandwidth is insufficient for delivering maximal throughput per process.

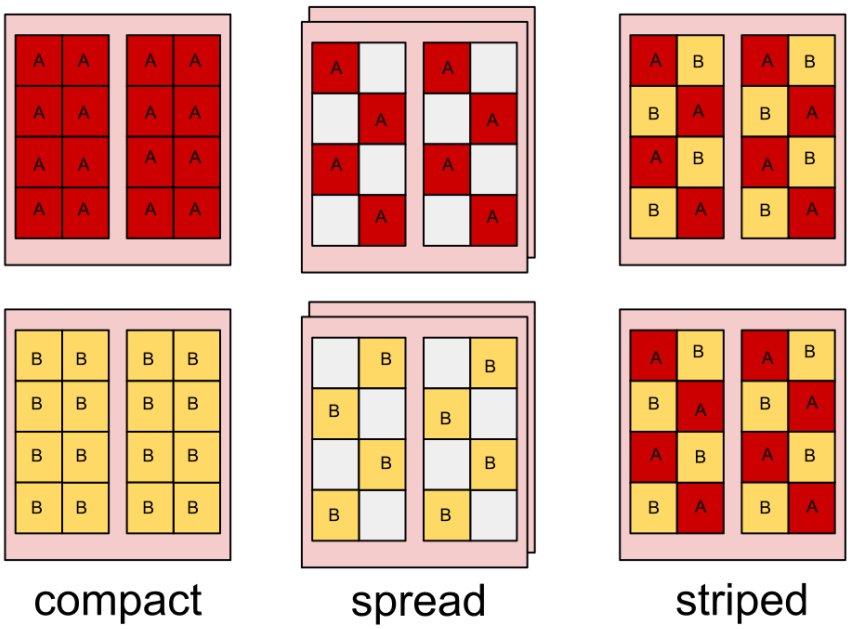


Figure 2.1. Methods for scheduling two 16 process distributed applications *A* and *B* on a supercomputer with two 8-core processor sockets (as in Gordon). In *compact*, *A* runs on one node and *B* on another. With *spread*, *A* and *B* are still isolated but run on two compute nodes each, with half the cores empty per CMP. Lastly in *striped*, A and B share all sockets of two compute nodes.

In fact, recent studies that examine single-process multi-threaded and distributed programs demonstrate that the optimal number of threads per CMP is highly application specific and rarely equal to the number of cores on the processor. In such cases, the optimal number of threads is either moderately lower or much higher than the number of cores available on a CMP [118, 195].

In such an environment, what we desire is a way to schedule that takes the best features of both *compact* and *spread*. We want the full occupancy that *compact* provides, but we also need the reduction in resource contention that *spread* offers. To do this, we develop and investigate *job striping*. Job striping takes two spread jobs and interleaves them such that every core is occupied by a uniquely identified single-threaded process.

In the *striped* configuration, even cores get assigned one application and the odd cores another. This process is repeated across all processor sockets. Figure 2.1 illustrates the *compact*, *spread*, and *striped* configurations.

Our intuition for why this can be effective is simple: distributed applications using MPI, or another mechanism such as UPC or SHMEM, often execute near identical tasks. As such, their constituent processes make very similar demands on the system. Suppose we are given a parallel program named *A* that largely follows the SPMD model. If each process comprising *A* requires that 6MB of its working set remain in L3 cache for high performance, then if one places 8 of these processes on the same 8-core processor, then at least 48MB of shared L3 cache will be required to avoid a reduction in throughput. However, on today's HPC systems, most L3 caches top out at 32MB. The requirement for a larger combined working set than the L3 offers will undoubtedly cause an increase in cache misses and, subsequently, a drop in performance.

However, job striping could remedy this problem. If *A* is striped with another job *B* whose processes each require 1MB in L3 cache, then the net performance is likely to be much better. While *A* scheduled as *compact* will almost certainly bottleneck on L3 capacity, when paired as *striped* with *B*, the sum of their working sets in L3 will be 28MB, small enough to fit within the 32MB L3 cache.

Communication over the inter-node network can also benefit from job striping. Many distributed scientific applications spend up to 35% of their total execution time sending and receiving messages [208]. Depending on the nature of this communication, similar processes can heavily contend for shared hardware such as the network interface card and switches. Suppose *A* exhibits short but intense bursts of communication. During these periods, it is quite likely that there will be a backlog of communication requests. If computation cannot be done to hide the latency of this communication, then performance will suffer. However, if there are half as many copies of *A* per node and *B*

does comparatively little communication, then A is likely to benefit from less waiting on the network and B will continue to make progress. In addition, even if A and B both are communication-heavy, their communication patterns are likely to be more out of phase than those of single parallel application [141].

In particular, a parallel program's tasks are limited in the distance they can run ahead of one another by the time length between global synchronization points. For highly synchronous codes, this means that all threads or processes execute the same set of instructions in relative lock step. For a communicating application A, this means that all of its processes very well might attempt to send messages over the network at roughly the same time. However, if A is paired with another code B, then the temporal communication patterns of A and B's processes can fall largely out of sync. This fact is corroborated by a closely related work by Koop et al. [141] where they show that pairs of symbiotic NAS parallel benchmarks often communicate over largely non-overlapping time intervals. For programs that make heavy use of synchronous communication such as MPI_Send and MPI_Receive, striping two applications in this way can significantly speed up program execution.

2.2 Methodology

2.2.1 Performance Metrics

To evaluate effective coschedules, we utilize a number of key metrics. To quantify system throughput, we use a variant of the weighted speedup metric from Snavely and Tullsen's work on symbiotic coscheduling for an SMT processor [214]. We call this metric scaled throughput (STP). STP is shown in Equation 2.2.

$$STP = \frac{1}{n}\sum_{i=1}^{n}\frac{S_i}{M_i} \tag{2.2}$$

In this equation, n is the number of parallel programs that are coscheduled together, S_i is the runtime of a program i when run alone in the compact configuration and M_i is the runtime of the program i when either the spread or striped affinity is applied to it. This variant of STP is the mean weighted speedup of the applications in a coschedule. For instance, a coschedule with an STP of 1.2 would exhibit a 20% average speedup over each job being scheduled using the compact policy.

For the purposes of determining energy efficiency, we have developed a new metric—Scaled Energy Efficiency (SEE). SEE is shown in Equation 2.3.

$$SEE = \frac{STP}{\bar{P_s}} \sum_{i=1}^{n} \overline{P_{c,i}} \tag{2.3}$$

$\bar{P_s}$ is the mean power of the striped or spread job schedule. $\bar{P_{c,i}}$ is the mean power of job i when run in the compact configuration. SEE is the product of scaled throughput and scaled power consumption. SEE is a reasonable energy efficiency metric because energy can be defined as the product of average power and time. Consequently, the metric approximates the ratio of the before (compact) and after (striped) power-delay products.

The $\bar{P_{c,i}}$'s terms are additive because the energy used by two jobs running isolated in the compact configuration is the sum of their individual energies. We can divide this by $\bar{P_s}$ because the number of compute nodes occupied by two jobs A and B scheduled compactly is the same as when the two are striped together. The $\bar{P_s}$ term is necessary because it is currently not possible to measure the contribution to net power from each job when they are striped. Only a combined measurement is realistically feasible.

2.2.2 Gordon Supercomputer

All of our experiments are conducted on San Diego Supercomputer Center's (SDSC) Gordon Supercomputer [183]. Gordon is a particularly innovative machine.

It is the first supercomputer to incorporate Intel's Sandy Bridge processors and has state-of-the-art Intel NAND Flash solid state drives on every node. The system features a 3D torus network topology with dual-rail QDR infiniband with 8GB/s of bandwidth in each direction between nodes. Each compute node is dual-socketed, each socket being occupied by an 8-core 2.6 GHz Xeon processor with 32KB of L1 and 256KB of L2 private cache per core and 20MB of shared L3 per socket for a total 40MB. Every compute node has 64GB of main memory with 32GB per NUMA node. We obtain this information by using the `lstopo` command from HWLOC [63] and from the Gordon user guide [20].

2.2.3 NAS Parallel Benchmarks

For one part of our study, we use the message passing interface (MPI) version of the NAS Parallel Benchmarks (NPBs) [36, 179]. As of version 3.3, the NPBs consist of 9 parallel benchmarks with input sizes ranging from W,S,A,B,C,D,E, and F where jumps between classes roughly scale runtime by 4 to 16x. NAS binaries are named as <benchmark name>.<class size>.<number of compiled processes>. For our experiments, we chose to use bt.D.1024, cg.E.1024, ep.E.1024, ft.D.1024, lu.E.1024, mg.E.1024, sp.D.1024. These input sizes ensure that each application runs for at least 20 seconds when scheduled in the compact mode.

The NAS Parallel Benchmarks are important for our study because they represent common computational kernels or motifs that comprise more complex real-world applications. In addition, they are open source, well studied, and exhibit strict versioning. Thus our experiments are repeatable.

2.2.4 Real Scientific Applications

We also conducted a study using three ubiquitous scientific applications: the 3D Gyrokinetic Toroidal Code (GTC) [158], LAMMPS [11], and the MIMD Lattice Computation (MILC) [13]. GTC is used for modeling microturbulence in plasma and is very heavily used at NERSC at the Lawrence Berkeley National Laboratory (LBL) as well as other Department of Energy (DOE) sites. LAMMPS is a classical molecular dynamics code that has been identified as one of the Department of Defense's key applications. In addition, it is currently being used by Lawrence Livermore National Laboratory (LLNL) as one of the applications to benchmark the new IBM BlueGene/Q Sequoia supercomputer [2]. Our third application MILC is a quantum chromodynamics program that enables the study of subatomic particle interactions. GTC and MILC were selected along with four other scientific applications to benchmark the petascale Hopper Supercomputer at LBL [30].

For GTC, we use an input file that is a modification of one provided by one of its lead developers. With LAMMPS, we use the embedded atom model (EAM) potential input provided with the Sequoia Benchmarks and scale the lattice by a factor of 32 in each of the x, y, and z directions. With MILC, we use the input provided in the NERSC-6 benchmarks for 1024 processes.

2.2.5 Compilation

We compiled both the NAS parallel benchmarks and the scientific codes using the Intel compilers (version 12.1). For our MPI library, we use MVAPICH2 (version 1.8) because it is optimized for multirail QDR infiniband. For GTC, we largely left the makefile untouched. LAMMPS requires an FFT library – we use the FFTW3 module (version 3.5) that defaults on Gordon when the Intel compilers and MVAPICH2 modules are loaded. In addition, we compile MILC with NETCDF (version 4.0.1).

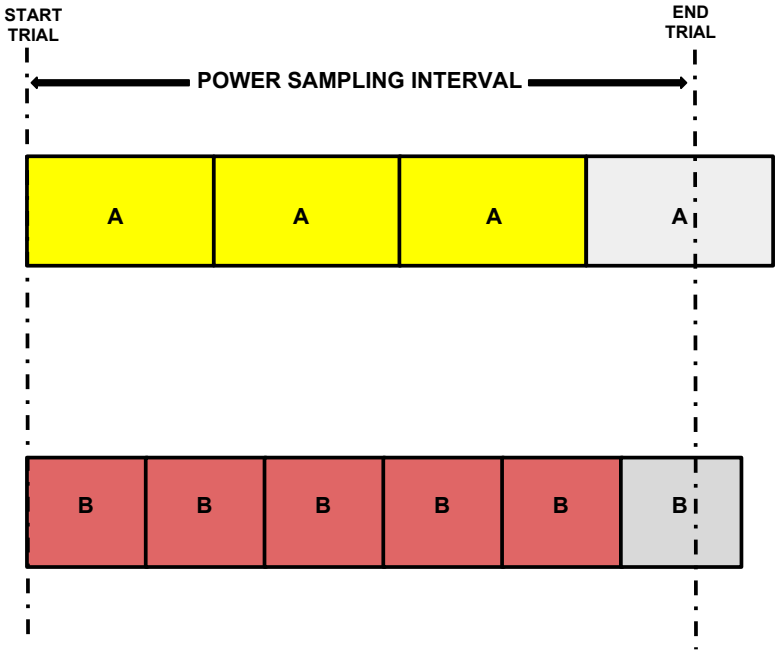


Figure 2.2. This figure represents a trial of running two striped applications A and B.

2.2.6 Experimental Setup

For each application, we spawn 1024 processes using mpirun_rsh. We run all of the experiments on two full racks of Gordon, totaling 128 compute nodes and 2048 processor cores. We evaluate NPBs and real applications separately. For each job, we run it in the compact, spread, and striped configurations. During trials using the compact scheduling method, we execute one copy of the application on one rack and another copy on the other. In the spread configuration, we bind four processes to each processor socket by setting **MV2_CPU_BIND=0:2:4:6:8:10:12:14** (see [194]). We do this to enforce job spacing in the hope that the absence of processes on adjacent cores will improve energy efficiency [77]. In both the compact and spread schemes, we restart a job as soon as it finishes.

For striped jobs, we set the affinity masks of the first and second job to **MV2_CPU_BIND=0:2:4:6:8:10:12:14** and **MV2_CPU_BIND=1:3:5:7:9:11:13:15**, re-

spectively. We start both jobs at the same time, and as soon as one job finishes, we restart it. Each striped coschedule is run for a fixed time interval, and the final trial from each job is discarded. For a visual representation, see Figure 2.2. This methodology allows us to (1) ignore unwanted tail effects and (2) sample the jobs co-executing at a variety of time staggers.

For the NPBs, we run each job in the compact and striped configurations for 12 minutes. After that, we run each possible pairing of NPBs for 12 minutes as well. This is adequate time so that each job completes execution at least three times. Between jobs, we *sleep* for 90 seconds to allow the system to go back down to steady state. For our real applications, we repeat the same steps but increase runtime per experiment to 30 minutes. With this time allowance, each application runs for at least 4 iterations per experimental trial. After running all trials, we repeated the same experiments several days later.

2.2.7 Power Measurement

We measure power consumption using a key feature of the Gordon supercomputer. For each rack, there are two power sources that provide the energy necessary for operation. On each of these sources is an embedded power monitoring unit (PMU). Each PMU makes a power measurement approximately every 3 to 5 seconds, however not in sync. The Performance Monitoring and Characterization (PMaC) laboratory at SDSC has developed a system whereby these data are sent over the network to a remote host and the readings are compiled into logs. To find average power, we take the discretized integral over the power readings of each PMU device during the runtime interval. As previously mentioned, this interval is 12 minutes for the NPBs and 30 minutes for the real applications. Once we have these values, we sum them together and divide by the runtime (12 or 30 minutes) to get average power. We do this rather than just taking the average over each power measurement because the time intervals between such measurements

are not uniform.

2.2.8 Performance Correlation

The first time we run the experiments, we dynamically instrument each executable at runtime with the IPM [243] profiler. When we rerun the experiments, we instead instrument with the profiling capabilities of PSiNS [227]. During each run of every experiment, we collect the L1, L2, and L3 cache misses, the total dynamic instructions, and the number of cycles without instruction issue for each MPI task. These data are aggregated at the process level through the use of PAPI [65] performance counters. In addition, both tools report the percent of time spent in each MPI routine for every process. The total penalty of instrumenting each application is at most 10%. This instrumentation is present in every run of the compact, spread and striped experiments. We do not believe that this instrumentation fundamentally alters application behavior. To make sure this is the case, we have repeated a large subset of our experiments without instrumentation and have observed no significant change in coschedule outcomes.

This instrumentation allows us to categorize the behavior of an application through the analysis of the traits of each of the parallel tasks that comprise it. From these data, we draw correlations between increase in striped performance and a reduction in system contention.

2.3 Performance Results

2.3.1 Compact Versus Spreading Versus Striping

Figure 2.3 shows the performance results for the first set of experiments with the NAS parallel benchmarks and the second set of experiments with GTC, LAMMPS and MILC. For the NAS parallel benchmarks, the mean performance increase from job spreading is 50%. If one examines striped coschedules of non-identical NPBs,

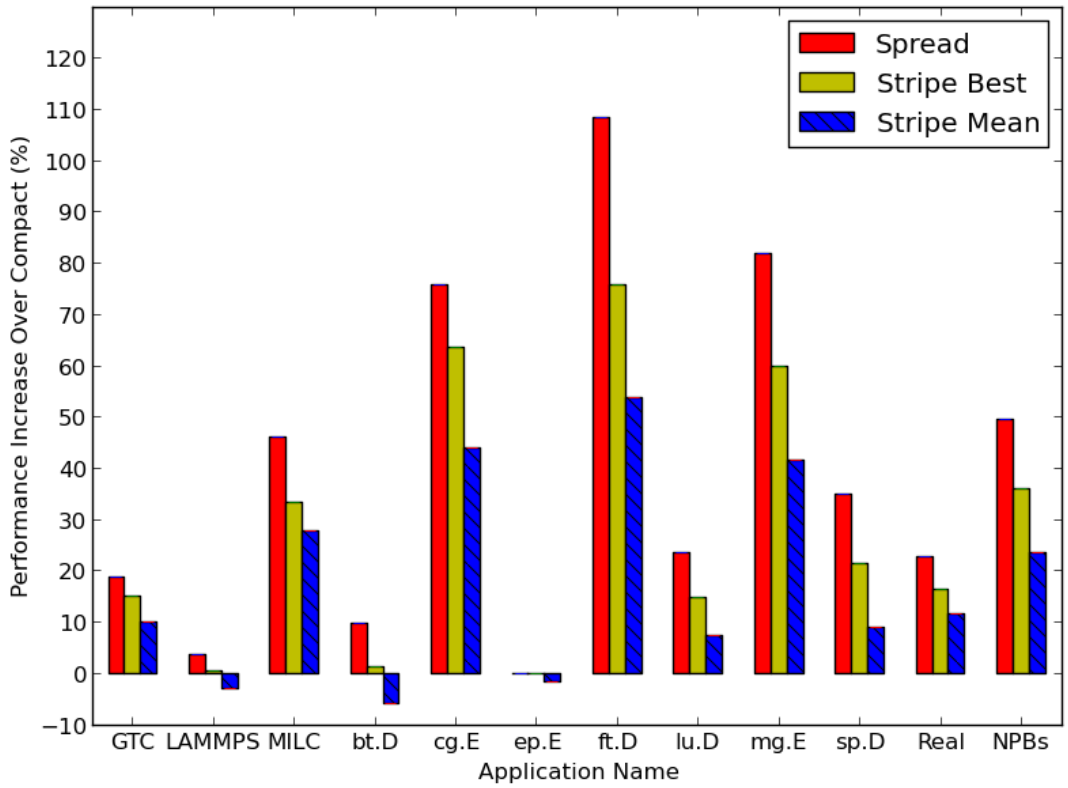


Figure 2.3. Increase in system throughput (STP) over compact when applying job spreading and striping to the NAS parallel benchmarks and GTC, LAMMPS and MILC

the average performance increase is 26%. If one selects the best running mate other than embarrassingly parallel (EP) for each benchmark, then the average increase in performance is 36%. We choose to exclude EP because EP is minimally contentious. Each EP task's working set fits entirely in the private levels of cache, and EP spends very little time in active communication. Because of these traits, EP universally causes every application that it stripes with to achieve its best striped performance. Thus for the sake of fairness and realism, we exclude these results from the "Best" average.

For the NAS parallel benchmarks, random striping yields about 50% of the performance benefit of job spreading and striping each job with its best running mate provides 70% of the performance benefit of spreading. This trend continues for real

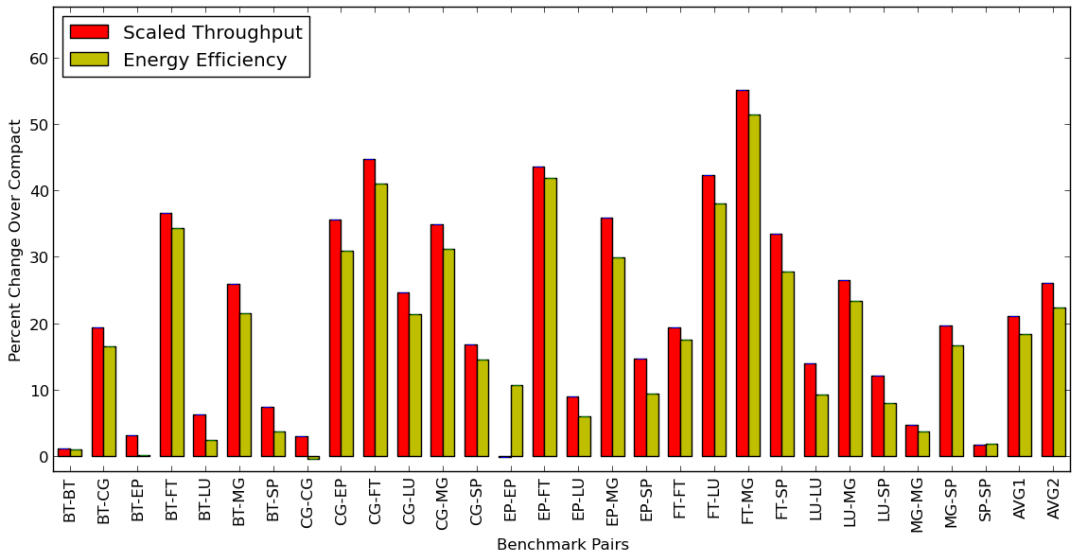


Figure 2.4. Increase in system throughput and energy efficiency when applying job striping to pairs of 1024 process NAS parallel benchmarks

applications as well. For the GTC, LAMMPS and MILC, job spreading increases throughput by 23%, and the mean heterogeneous striping and the mean best striping improve performance by 12% and 16% respectively.

2.3.2 NAS Parallel Benchmarks

In this section, we examine the increase in collective throughput and energy efficiency for pairs of striped NAS Parallel Benchmarks. The results are presented in Figure 2.4.

For completeness, we run all pairwise combinations. This includes both heterogeneous pairings (e.g., CG striped with LU) and homogenous pairings (e.g., CG striped with CG). However, homogeneous pairings typically combine the worst of *compact* (maximized pressure on bottleneck resource) and the worst of *spread* (some increased communication as communicating threads are farther apart). Thus, AVG1 refers to the average of all possible pairings and AVG2 to the mean performance of only heteroge-

neous pairs. It is reasonable to assume that a job striping runtime system would avoid pairing identical applications together.

Overall, striping provides improved performance. Even in the worst case where we are limited to pairs of identical benchmarks, job striping in aggregate still increases both STP and SEE by 6%. Although the kernels are identical, the two instances do not synchronize with one another, and as a result they compete less for the inter-node network. FT-FT is the best homogeneous pairing and exhibits a 19% increase in performance and a 17% increase in energy efficiency. For additional discussion of the FT-FT pairing, see [141].

Heterogeneous striping improves system throughput by 26% and energy efficiency by 22%. With every benchmark, striping improves scaled throughput (except EP striped with EP -0.2%) and energy efficiency (except for CG striped with CG -0.5%). The best coschedule is the FT-MG pairing. FT speeds up by 51% and MG by 60% for a total STP increase of 55%. Energy for the FT-MG pairing improves by 52%. The next best coschedule in terms of throughput and energy efficiency is CG paired with FT. CG speeds up by 51% and FT by 39% for a total combined performance improvement of 45%. The next two best pairings are EP-FT and FT-LU.

The EP-FT pairing is particularly interesting because it is one of the instances where one application benefits at the expense of another. If we saw no benefit from heterogeneous scheduling, we would expect this to be the common case – that job striping would sacrifice one job for the benefit of another (e.g., a cache-intensive job would accelerate, but the co-scheduled job would suffer from the increased cache pressure). In fact, in over half the cases, job striping improves the throughput of *both* applications. However, EP-FT is one of the exceptions. EP suffers a 4.6% loss in performance, whereas FT's throughput increases by 92%. There are 13 such cases. In these pairings, the average performance decrease to the victim application is 4.1%, whereas the average performance

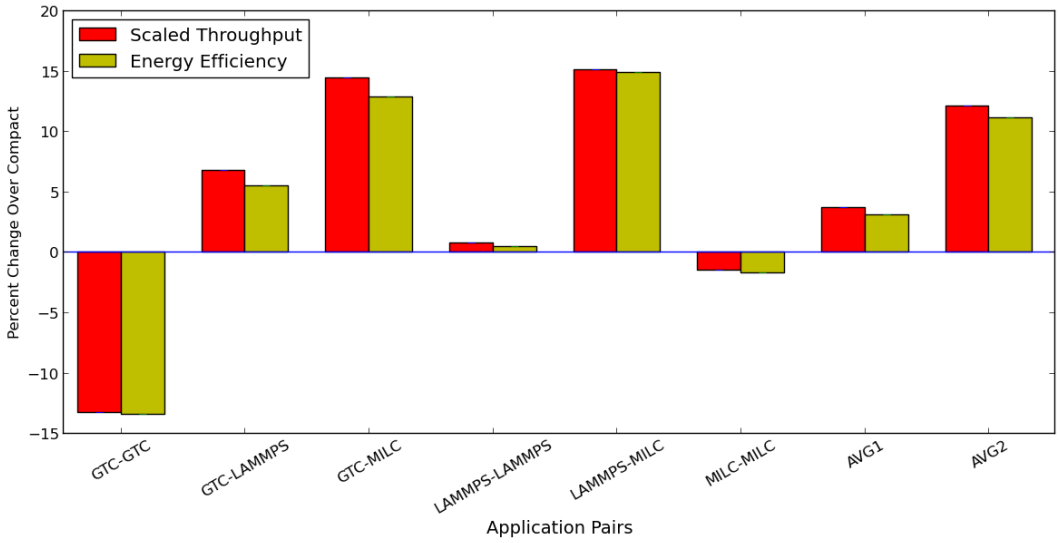


Figure 2.5. Increase in system throughput and energy efficiency when applying job striping to pairs of 1024 process scientific codes; AVG1 is the mean over all coschedules, and AVG2 is the mean over heterogeneous ones.

increase of the accelerated application is 48.9%. BT and EP are the most commonly victimized kernels, with BT suffering a performance hit from five out of seven pairings, and EP suffering a throughput hit in all seven. Besides these, only CG-SP and CG-LU exhibit victimization (SP and LU victims).

We can partition the pairings into the set of jobs where both improve, and those where one is slowed. Interestingly, the overall system performance gain of the two sets is similar. The former sees 19.4% gain from striping, while the latter gains 19.1%. The gap is only a bit larger if we examine energy efficiency. The former group gains an SEE improvement of 19.3%, while the latter gains 16.4%. These results do raise questions about fairness for the users who initiated the striped jobs. These concerns will be addressed further in Section 2.5 and Chapter 3.

2.3.3 Real Scientific Applications

For the production scientific codes, job striping again improves throughput and energy efficiency. Figure 2.5 illustrates the scaled throughput and energy efficiency per striping. As with the NAS parallel benchmarks, we observe that some applications benefit more than others from striping. The application that benefits most from striping is MILC (at 1024 processes on the NERSC input). When it is paired with GTC and LAMMPS, its throughput increases by 23% and 34% respectively. GTC also exhibits performance gains when striped. Striping GTC with LAMMPS and MILC improves GTC's throughput by 12% and 6%.

LAMMPS is the only application that does not benefit from heterogeneous striping and the only one whose performance improves from homogeneous striping. It suffers a mean 1.7% decrease in throughput when striped with GTC and 3.4% performance loss when paired with MILC.

In addition to improving application throughput, job striping also raises energy efficiency. Similar to the NAS parallel benchmarks, there is little change in power consumption relative to the increase in useful execution. For heterogeneous coschedules, mean energy efficiency increases by 11%, almost as much as the 12% increase in scaled throughput.

Stability of Results

Table 2.1 reports the minimum and maximum observed runtime per real application over eight or more trials. From the table, we observe that the results from the previous section are quite stable and reliable. Over the multiple runs of each application, the maximum variance in runtime for all applications in each of the three heterogeneous pairings is less than 2% and is no larger than running applications compactly. Consequently, the behavior of a striped coschedule is highly consistent between runs.

Table 2.1. Minimum and maximum runtime in seconds for each of the three applications scheduled compactly, spread, or striped.

		Compact	Spread	GTC	LAMMPS	MILC
GTC	MIN	198.9	167.6	227.9	172.7	187.9
	MAX	202.2	170.2	231.7	174.3	191.5
LAMMPS	MIN	289.9	280.0	295.2	288.3	300.8
	MAX	292.0	282.9	296.3	290.2	302.9
MILC	MIN	511.4	350.2	416.7	383.5	515.7
	MAX	514.7	354.7	424.6	388.3	527.4

2.4 Interaction Between Striped Pairs

Thus far we have only focused on the performance benefit of job striping. In this section, we investigate how it is achieved. From our point of view, the best way to predict how a job's performance will change when striped is to look at the change in its relative L3 misses and communication when spread. Since striping is the interleaving of two spread jobs, a job's change in behavior when spread should largely dictate how and when it will benefit from striping. Figure 2.6 presents the change in relative L3 misses, the time spent in computation and communication, and the increase in performance.

2.4.1 Spread Performance Relative to Compact

We examine L3 cache misses because previous studies have demonstrated that applications with high L3 cache miss ratios often have lower instruction throughput than ones with fewer L3 misses. In addition, the literature shows that pairing heterogeneous uniprocess single and multithreaded programs together can often reduce L3 misses, and boost performance [49]. On real machines, an L3 cache miss incurs a latency penalty that is often an order of magnitude greater than misses to the higher level private caches. On Intel Sandy Bridge processors, the quoted L3 access time is 26 cycles. However, the latency to access main memory is much greater. We use lmbench to measure the relative latencies and find the L3 access time to be 42 cycles and the time to access main

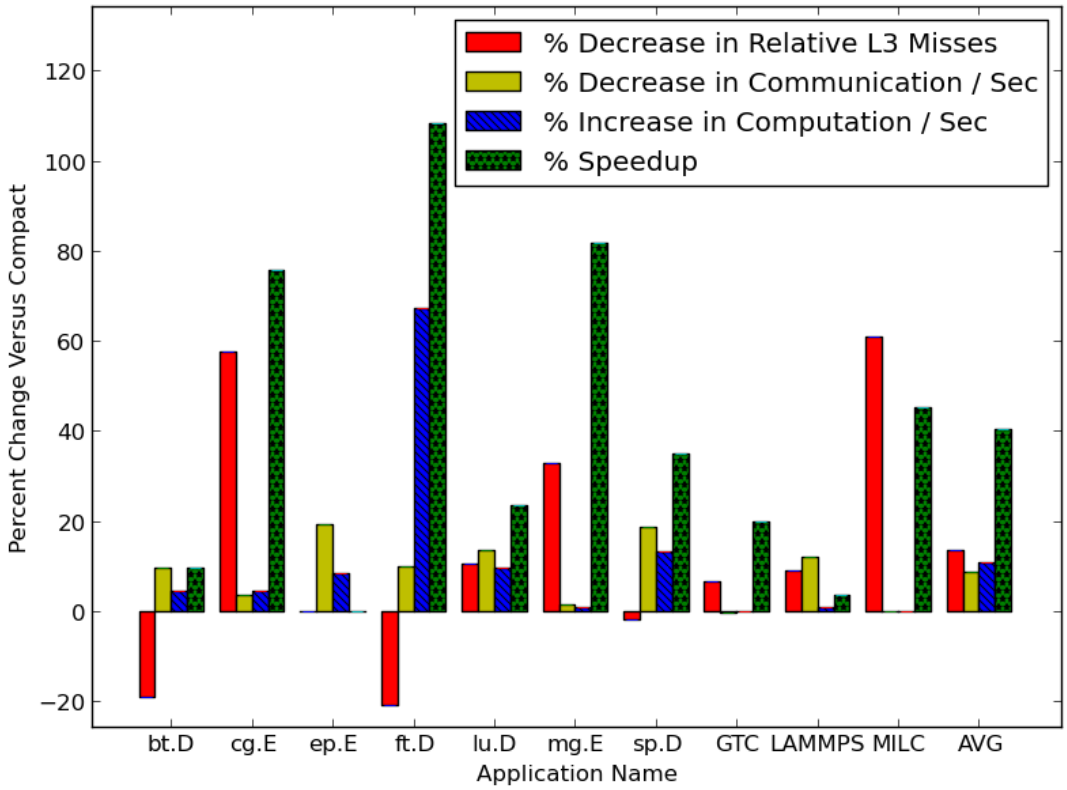


Figure 2.6. Shown are the change in application performance and performance counters when jobs are spread versus compact: L3 change is scaled relative to the mean L3 misses per 1000 instructions of all ten codes when individually scheduled (compact) and communication and computation changes are scaled by percentage of total runtime. The increase in computation is relative to the total percentage of time spent in computation versus communication.

memory to be 220 cycles [170, 204]. Thus, for each miss to the L3 cache, we incur an additional latency penalty of 180 cycles for that cache line. If this miss occurs on a read, it is unlikely that the processor can completely hide this latency. These latencies are of particular concern for CG and MG which exhibit 5.7 and 2.7 L3 misses per 1000 instructions.

In Figure 2.6, we observe that certain benchmarks show a significant reduction in L3 cache misses when spread. CG, MG and MILC realize substantial benefit when

spread. This is not surprising: CG, MG and MILC have the largest working sets, each allocating 873, 499, 716 MB per active process, and also the highest L3 miss ratios of their peers. Thus the spread configuration greatly reduces cache and memory pressure.

However, certain programs such as BT, FT, and SP actually incur more misses as a result. While an increase in performance despite more L3 misses may seem counter intuitive, if we examine the increase in useful computation and the decrease in communication, these numbers make sense. All three benchmarks exhibit substantial increases in computation per unit time. Thus, even though they incur penalty from more cache misses, this can be offset by an increase in useful work and less waiting on the network.

Most applications exhibit both a decrease in L3 misses and decrease in communication per unit time. To weigh the importance of these changes, one must first look at the applications' characteristics. EP for instance has almost no L3 cache misses per 1000 instructions, thus pairing it with other applications can only hurt. In addition, EP performs much less communication than the other applications. Thus when it is heterogeneously paired, its performance universally suffers. BT is also interesting because spreading significantly increases its relative L3 misses. Thus BT can only benefit from striping if it reduces BT's communication contention. If we look at BT's best pairings, they all occur when it is paired with applications that do as much or less communication than it does (itself and EP).

If we examine the highest throughput coschedules, almost all include FT. FT's L3 misses go up when it is paired with other applications, but this is expected given that spreading alone increases its relative L3 misses. This fact is however offset by the savings due to communication. In particular, FT.D.1024 spends more time in MPI routines than any other program, and it has the second lowest relative L3 cache misses of any program (0.48 per thousand instructions). In addition, reducing the time it spends in communication sharply increases its actual computation per unit time. Thus, it can afford

additional cache misses.

When FT is paired with CG and MG its communication per unit time decreases by 8.0% and 4.5% respectively. While this may not seem like much, remember that FT originally spends over 87% of its time communicating. Thus decreasing communication per cycle by 8.0% means FT spends 20% of its time executing meaningful computation instead of 13%. Thus, it executes 54% more computation per unit time. This figure almost exactly matches the 55% increase in throughput for FT, when it is paired with MG. The same is also true for FT paired with CG. FT now spends 30% more time doing computation. This number roughly matches the 39% increase in performance that we observe when it is paired with CG. In addition, we note that CG and MG also spend a large percentage of their total time in communication, 57% and 37% respectively. However, their communication patterns are distinct from FT's. FT spends approximately 80% of its total communication time executing MPI_Alltoall, and thus each process, and hence every compute node sends every other compute node its results.

CG on the other hand spends 53% of its total computation time in synchronous point to point MPI_Send communication and 43% waiting. We believe FT benefits from this pairing because CG's communication pattern is distinct and CG is largely inactive in MPI_Wait for large periods of time. Thus FT benefits from CG waiting on communication and also by the fact that the nature and phases of CG's communication are largely orthogonal to its own.

MG's behavior is also highly distinct from FT's. MPI_Wait and MPI_Init together account for 73% of its total time in MPI routines. Thus, FT can largely run without interference, as MG executes a brief burst of initial communication and then largely tapers off. For this reason, MG is particularly symbiotic with other benchmarks. Rather than having 16 MG processes all competing for network access during initialization, there are only eight. After initialization, it does very little active communication, and its

partners profit.

For CG, we believe that the majority of its performance improvement when striped is from reduced L3 cache misses. Only when CG is paired with MG does the percentage of execution spent in communication sharply drop (from 57% to 40%). However, even though the percentage of time CG spends communicating in most coschedules does not change, the communication composition does. Pairing CG with FT causes the time spent in MPI_Wait to decrease by 20 percent. This result is particularly interesting because it reiterates an important point about job striping: job striping can improve performance by reducing LLC contention, communication interference, or a combination thereof. When job striping pairs an application with a large working set and a high LLC cache miss ratio with another application with a small working set and low LLC cache miss ratio, the first application benefits because more of its working set now fits in cache. The second application consequently then suffers more cache misses because less of its working set now fits in cache. However, we observe that many applications with large working sets often spend more time in computation, and applications with smaller working sets usually spend more time doing communication. Applications that exhibit both high communication and high cache contention usually do not scale. Thus, even though the cache-benign application now has a higher LLC miss rate, it is now paired with an application that spends either less time in communication or with a communication pattern distinct from its own.

2.4.2 Real Applications

For real applications, we observe behavior that is similar to that observed with NPBs. MILC's L3 cache misses per 1000 instructions drop significantly when paired with both LAMMPS and MILC. Scheduled compactly, MILC has 1.7 misses per 1000 instructions. When scheduled with GTC or LAMMPS, this figure respectively drops to

1.03 and 0.78. Scheduling using the spreading method decreases this number further to 0.58. If we correlate these numbers to what we see with performance, a trend becomes clear. MILC runs for 513 seconds in compact, 413 seconds paired with GTC, 382 seconds when paired with LAMMPS, and 353 seconds when run in spread. If we fit a linear $y = mx + b$ function to these data (runtime vs. L3 misses per 1000 instructions), we get a coefficient of determination of $r^2 = 0.998$. A coefficient of determination of 1 corresponds to perfectly collinear data, and so the linear model is highly representative of the actual trend in the data. Thus, if we pair MILC on the NERSC input with another application and measure MILC's L3 misses per 1000 instructions, we expect to be able to accurately predict MILC's run time by using our linear model.

The number of cycles without instruction issue also drops when MILC is striped. In the compact configuration, MILC has 22 cycles without instruction issue for every 1000 instructions. Spreading MILC reduces this number to 10, and pairing MILC with GTC and LAMMPS improves this figure to 17 and 13 respectively over the compact baseline. When plotting runtime versus cycles without instruction issue per instruction, $r^2 = 0.964$. While this correlation is slightly weaker, this is to be expected. Depending on where an L3 miss occurs in the code, it may or may not cause the pipeline to stall. When we conduct a linear correlation on L3 misses and cycles without instruction issue per 1000 instructions, $r^2 = 0.976$. Thus we conclude that MILC's speedup when striped is almost entirely tied to a decrease in contention over last level cache. These findings are similar to those found in the SPEC2006 MILC benchmark [49].

GTC benefits the second most from job striping. Like MILC, spreading significantly improves its L3 miss rate per 1000 instructions, while communication as a percentage of runtime remains essentially constant. We believe GTC runs well with MILC because their communication patterns and cache accesses are largely synergistic. GTC spends approximately equal time in MPI_Allreduce, MPI_Sendrecv, and MPI_Barrier.

MILC on the other hand spends 66% of its communication time in MPI_Wait and 24% in MPI_Allreduce. Thus for both codes, the majority of communication time is spent waiting for the data to propagate. This explains why spreading does not reduce the communication time percentage for either application, as waiting is not a contentious activity. In terms of cache behavior, GTC makes very few L3 cache accesses. Thus even though MILC's working set in L3 is large, GTC does not suffer.

LAMMPS exhibits very little change in performance when striped with itself and other applications. This is reasonable given that spreading LAMMPS only increases performance by 3%. While we expected the increase in LAMMPS' performance to be more significant due to the marked decrease in L3 misses for LAMMPS when spread, we conclude that these misses must be less detrimental than those incurred by GTC or MILC. In addition to little benefit from reduced cache contention, LAMMPS cannot profit much from improved communication either. In total, LAMMPS spends (7%) of total execution in MPI calls, so that even though pairing LAMMPS with GTC decreases its communication time by 15%, this has a largely inconsequential impact on runtime. We attribute minor benefit to the composition of LAMMPS's communication. LAMMPS spends over 65% of its communication time in bursts of synchronous MPI_Send. By having fewer processes executing MPI_Send at any one time, performance is improved.

Pairing LAMMPS with itself very slightly improves performance, and we attribute this to the communication of each LAMMPS parallel job being out of synchronization with one another. This is consistent with the fact that our communication-intensive or highly synchronous NAS parallel benchmarks also sped up when homogeneously striped.

2.4.3 Per Process Improvement from Striping

In the previous section, the overall benefit to the system from job striping was demonstrated in terms of both scaled throughput and energy. Figure 2.7 provides the im-

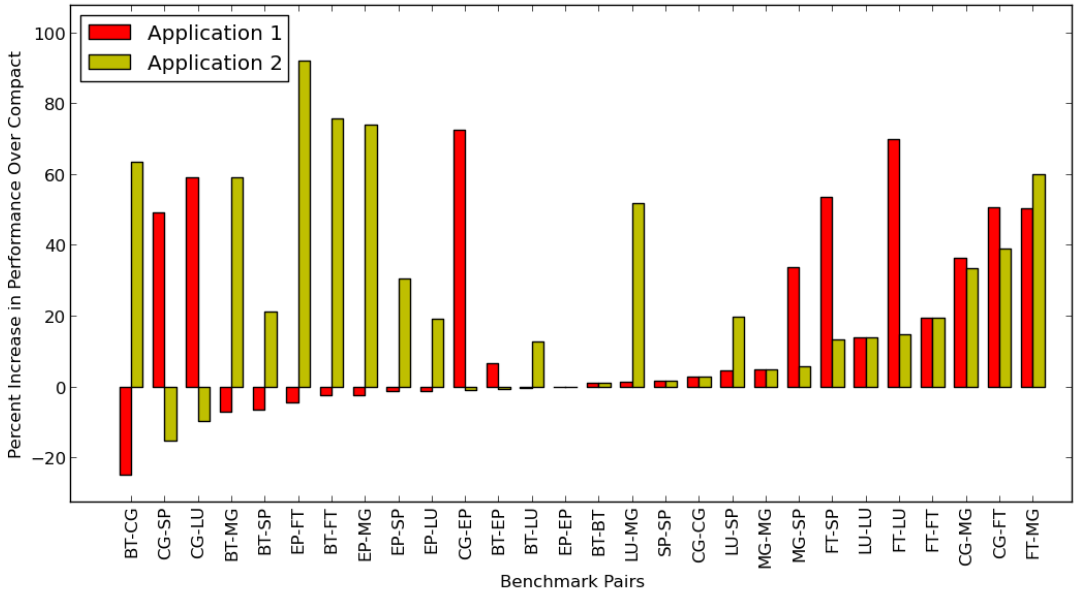


Figure 2.7. Improvement to scaled throughput per NAS benchmark in each pairing. Pairings are sorted by the worse performer of the pair.

Table 2.2. Benchmarks categorized by the reduction in execution time from running spread over running compact.

	Improvement of Spread over Compact		
	High $\geq 40\%$	Medium $\geq 20\%$ and $< 40\%$	Mild $< 20\%$
NAS	CG, FT, MG	LU, SP	BT, EP
Real	MILC		GTC, LAMMPS

provement to runtime per process in each pairing of the NAS benchmarks and Figure 2.8 provides the improvement per process for our real benchmarks. Although most of these pairings are an overall benefit to the system (see prior section), individual jobs may suffer from the pairing. Approximately half (16/34) of the striping pairs achieve symbiosis—each of the jobs in the pair benefit. 12 out of 34 of the striped pairs experience one of the jobs suffering less than a 5% degradation in performance. Lastly, in 6 out of the 34 pairs, one of the benchmarks suffers by greater than 5%.

Table 2.2 groups benchmarks by their gains from being run spread over compact.

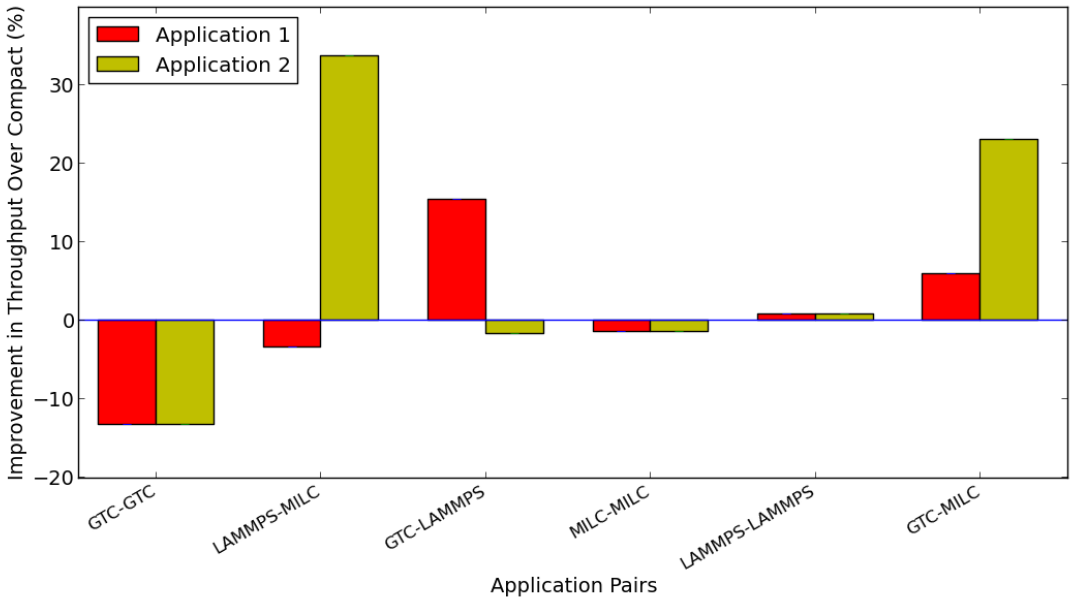


Figure 2.8. Improvement to scaled throughput per Real benchmark in each pairing. Pairings are sorted by the worse performer of the pair.

This produces groups of "High," "Medium," and "Low" spread performers. In general, striping with another job will provide worse execution time than if run alone spread. This allows us to use the spread behavior to infer the behavior of running striped. Table 2.3 presents the performance impact of pairing across different groups of benchmarks. It summarizes results from a combination of Figures 2.7 and 2.8 with Table 2.2.

When scheduling a High spread performer for striping, that benchmark always benefits from the striping with the one exception of MILC striped with MILC (where each suffers a minor slowdown). This is fairly intuitive: if the benchmark gains 40% or more running spread alone, it is unlikely (but possible) the interference from the coscheduled job will counteract the large spread gains.

Striping with Medium spread performers is more problematic as they always do well when scheduled with other Medium spread performers, but can suffer when scheduled with High spread performers. Four out of six such pairings of High-Medium

Table 2.3. Number of pairs which fall under the outcome of Symbiotic, Minor Interference (one benchmark loss of $< 5\%$), or Non Symbiotic (one benchmark loss of $\geq 5\%$) when pairing benchmarks from different categories of spread benefit (High, Medium, and Low).

Pairing	Symbiotic	Minor Interference ($< 5\%$ Loss)	Moderate Interference ($\geq 5\%$ Loss)
High-High	6	1	0
High-Medium	4	0	2
High-Low	1	5	2
Medium-Medium	3	0	0
Medium-Low	0	3	1
Low-Low	2	3	1

are Symbiotic, but two of those six exhibit Moderate Interference. As examples, CG-LU is a High-Medium pair where CG does very well at the expense of LU and the FT-LU pair is a High-Medium pair where both benchmarks benefit from the striping. Medium paired with Low always benefits the Medium spread performer. Although only one such pairing is non-Symbiotic, in three of the four cases the Medium benchmark benefits while the Low benchmark experiences a minor loss.

Striping of Low with Medium or High always benefits the Medium or High spread performer more than the Low. Although in many of these cases (8/11) the Low spread performer only suffers a minor loss in performance, in some cases (3/11) the Low spread performer suffers more substantially. Scheduling pairs of Low spread performers is generally of dubious benefit.

Although the interactions across the different categories of High, Medium, and Low spread performers can cause some complications, a simple trend remains true. When scheduling a High with Medium or Low, High ends up getting the better benefit. When scheduling Medium with Low, Medium receives the majority of the benefit of striping. We will address how one might use this information to address fairness concerns in the following section.

2.5 Fairness

As we have previously identified, striping can lead to colocations where one application speeds up relative to compact while the other slows down. Different systems have different models of charging users for use and each of these different models may choose to handle fairness concerns related to job striping differently. For example, in highly collaborative systems where all users value overall system throughput over individual job latencies, job striping could be enacted uniformly with everyone recognizing the value. In contrast, in systems where users are charged by the CPU hour, a fairer mechanism may be necessary (as is argued in Chapter 3). We offer the following suggestions that may be used individually or in concert with one another:

- Separate job queues could be provided. Users interested in running striped could submit to that queue.

- Submission to the striped job queue could be incentivized by the system administration in the form of additional running hours or reduced cost. In this study, reducing the cost to users by 23% would be enough to guarantee that no user paid more in the striped queue versus running their job in a traditional queue. This cost reduction would be equivalent to passing on the efficiency savings from running NAS-like codes on to the user. Chapter 3 builds on this idea by proposing awarding users discounts that scale in proportion to how much their job's spread performance degrades from being striped with another job.

- Users informed of the implications of spread performance on striped performance could use that information to aid in fairness decisions. For example, high spread-benefit jobs might be charged extra for submission to the striped job queue and low spread-benefit jobs might be charged less. When those jobs are paired, each user is

charged a more "fair" value and the overall system benefits.

- Future work may develop striped performance prediction which could be used to inform the accounting mechanism. Chapter 3 describes one such mechanism, but techniques that draw on prior work on diagnosing inter-thread contention [49, 164, 191] are also likely to be effective.

Fairness concerns vary by system and may conflict with concerns for overall system performance. Each system may need to re-address these concerns in light of job striping. This work does not aim to provide a panacea for these potential concerns but to instead inform the reader of the value of job striping and of potential mechanisms to aid in fairness decisions.

2.6 Related Work

This study is the only one to our knowledge that combines use of real scientific applications, power measurement and analysis, and coscheduling at thousands of cores into a single work. Previous studies on CMP-based systems have largely focused on single server coscheduling with multithreaded benchmarks or commercial applications.

The study that most resembles our own is by Koop et al [141]. They show that by colocating pairs of NAS parallel benchmarks across several machines, performance can be improved by reducing communication contention. They present data that shows that symbiotic applications communicate during largely disjoint time intervals.

Snavely and Tullsen proposed symbiotic scheduling for SMT processors [214]. Weinberg and Snavely evaluate the potential of symbiotic workload space sharing on an HPC platform [237] and the users' ability to accurately determine resource bottlenecks on that platform [238]. Paired gang scheduling proposes pairing I/O bound jobs with compute intensive jobs for better overall throughput in HPC [242].

Shared levels of cache have long been recognized as a point of contention. Yan and Zhang aim to predict worst case run-times by using profiled control flow information to predict i-cache contention between two threads [252]. Anderson et al. evaluated grouping processes by L2 miss ratios to avoid coscheduling those with high miss ratios, for real-time tasks on multicores [29]. Fedorova et al. recognize the potential of using heuristics for better scheduling on SMT processors. They use predicted L2 miss ratios based on reuse distance [46] to inform their scheduler on in-order, simulated, SMT processors [93]. StatCC was developed to predict shared cache miss ratios and subsequently coscheduled thread CPI [83]. StatCC evaluated 2-thread coschedule prediction against results from an in-order processor simulator. Blagodurov et al. recently evaluated various classification metrics for thread coscheduling [71, 139, 248]. Based on their evaluation, they used performance counters to derive metrics to aid in scheduling as to reduce LLC cache contention [49].

Cache partitioning for Quality of Service and fairness has been evaluated for SMT processors [67] as well as CMPs [106, 137, 157]. Hardware support for QoS or Fairness has limitations—namely higher expense, less flexibility, and longer time to market, and proposed software solutions often require dedicated time slicing to ensure disadvantaged threads make fair progress on existing systems [95].

More recent work by Iancu et al. in [118] looked at the benefit of over-subscription of multicore processors for various implementations of the NAS parallel benchmarks. They too found heterogeneity to be beneficial.

In the commercial domain, there has been some recent work on coscheduling. Tang et al. investigate how the performance of Google's workloads changes when using different thread-to-core mappings [225]. Mars et al. present Bubble-Up, a mechanism that can very precisely trade a small decrease in the QoS of certain applications for significant gains in data center utilization [164]. In addition to these works, the multi-tiered Déjà

Vu system provides a practical solution for intelligent colocation of virtual machines by combining a lookup table of virtual machine signatures, a clustering algorithm, and a small testbed of machines for periodic profiling when the runtime system's assumptions are violated [232].

2.7 Conclusion

This work has shown that there is tangible benefit in placing heterogeneous distributed applications from different users on the same set of shared resources. We validated the job striping approach at large scales on NAS parallel benchmarks and on three real applications and realized performance benefit both in terms of throughput and energy efficiency.

Random heterogeneous coschedules of 1024 process NAS parallel benchmarks improved throughput by 26% and energy efficiency by 22%. If we could select the best coschedule for each benchmark, throughput and energy efficiency improved further to 31% and 26% over the baseline.

For our real applications, we have shown that GTC and MILC benefited significantly from heterogeneous coscheduling and that LAMMPS boosts the throughput of any coschedule. MILC's throughput when striped with different applications increases by an average of 26%; in addition, we have illustrated that MILC's improvement from job striping is highly correlated to reductions in L3 misses and cycle stalls.

Furthermore, we have characterized how job striping works. Job striping improves the throughput of some applications by reducing communication contention (BT and FT). On others such as CG, MG and MILC, it reduces LLC misses. A combination of communication and LLC contention is also observed for some applications (LU and SP).

Job striping's performance can be predicted by examining how an application's traits change when spread. Applications that exhibit marginal benefit from spreading do

not exhibit performance benefit from striping.

We have also shown that job striping yields reliable performance on real applications. Striped jobs that run over similar length time intervals exhibit performance variation that is no higher than compactly scheduling jobs in the traditional manner.

Going forward, we propose extensive reexamination, validation, and testing of the existing software infrastructure. Our work shows that a simple change in the scheduling framework can produce a significant benefit. We are certain that further examination of other established practices will reveal additional opportunities for optimization.

2.8 Acknowledgements

The authors dedicate this work to the loving memory of Dr. Allan Snavely. In addition, the authors thank the anonymous reviewers for their constructive feedback. We also thank the following outstanding individuals at SDSC and Calit2: Jeff Bennett, Pietro Cicotti, Jerry Greenberg, Adam Jundt, Mitesh Meswani, Wayne Pfeiffer, Bob Sinkovits, Shawn Strande, Mahidhar Tatineni, Nicole Wolter and Kenneth Yoshimoto. Your advice and support has been invaluable.

This research has been supported in part by the DOE Office of Science through the Advanced Scientific Computing Research (ASCR) award titled "Thrifty: An Exascale Architecture for Energy-Proportional Computing," National Science Foundation grant: OCI-0910847, Gordon: A Data Intensive Supercomputer, NSF Grant CCF-0702349, NSF grant CCF-1018356, the UCSD Computer Science and Engineering Department, the Reserve Officers Association Henry J. Reilly Memorial Scholarship, and Semiconductor Research Corporation Grant 2005-HJ-1313. The authors also acknowledge the support of the Multiscale Systems Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program.

Chapter 2, in full, is a reprint of the material as it appears in Darren Kerbyson,

Georg Hager, Gerhard Wellein, Abhinav Vishnu, editors, Concurrency and Computation: Practice and Experience 2013. Breslow, Alexander D.; Porter, Leo, Tiwari, Ananta, Laurenzano, Michael L., Carrington, Laura N., Tullsen, Dean M., Snavely, Allan E., Wiley Blackwell, December 2013. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Fair Job Pricing for Supercomputers with Node Sharing

Supercomputers typically have hundreds to thousands of users and consist of tens to thousands of individual servers connected over a high-speed optical interconnect. At any one time, many users concurrently utilize the system. The current approach has been to give each user a non-overlapping set of compute nodes on which to run his or her application. While this approach prevents jobs from different users from clobbering one another, it leads to a missed performance opportunity. Chapter 2 and recent work show that co-location, where a set of jobs from different users runs on a shared set of compute nodes, can increase mean application performance and system energy efficiency by 20% by reducing contention for shared resources in the memory subsystem and inter-node network [58, 118, 141]. In addition, current architectural trends and exascale computing studies suggest that the benefit of co-location is likely to increase. The studies project that compute nodes will have hundreds to thousands of cores [47]. For some applications, it may not be possible to use all of these cores efficiently. In particular, 80% of all XSEDE jobs use less than 512 cores [21, 176], which means co-location will likely be necessary to utilize all of a node's cores.

Co-location seems inevitable for larger jobs as well. Projected scaling trends

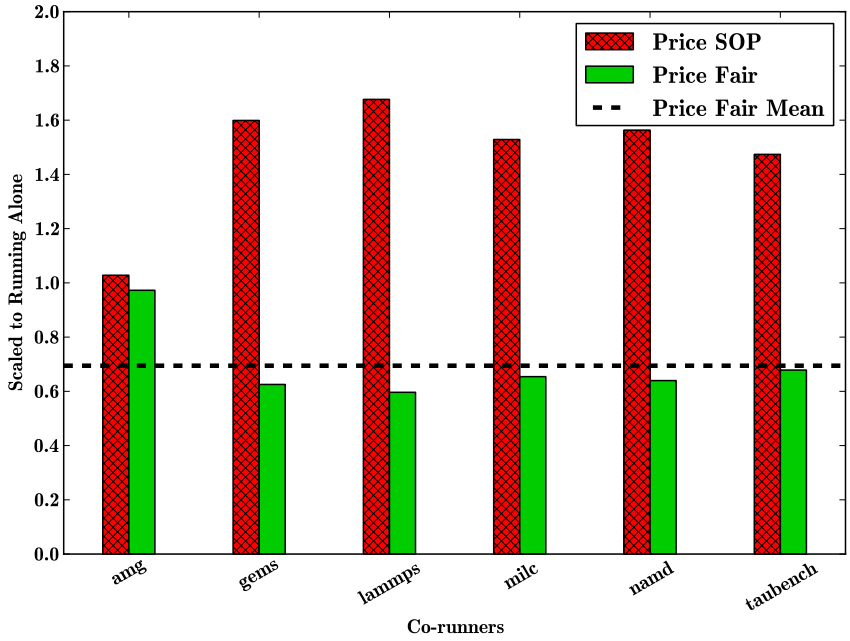


Figure 3.1. Performance of GTC, a plasma physics code, when co-located with the applications on the x-axis. The current pricing mechanism penalizes the user for co-locating their job by charging them more when their job degrades more.

suggest an increase in the number of cores per node that outpaces increases in memory bandwidth and cache capacity, which will reduce the resources available per core [47]. To mitigate contention, resource-hungry jobs will have to be spread out over more compute nodes and paired with resource-light jobs to maintain high system utilization [58].

Although co-location is beneficial to performance and energy efficiency, it also creates a new set of challenges, one of which is *fair pricing*. Fair pricing is a concern because although there is a net benefit from co-location, some pairings can cause one of the applications to slow down.[1] When this happens, we argue that the user should be discounted. However, if we apply the current state-of-practice (SOP) in HPC infrastructures, where users are billed proportionally to the time to execute their job, we find there is gross inequity – users whose jobs benefit from co-location pay comparatively less while users whose jobs do not benefit pay more.

[1] Chapter 2 presents and discusses corroborating findings on real, large-scale distributed applications in Sections 2.4 and 2.5.

Figure 3.1 illustrates the challenge. Under the current state-of-practice, a user running GTC [158], a plasma physics code, pays 60% more when co-located with LAMMPS [11], a molecular dynamics code, versus AMG [2, 37], a parallel algebraic multigrid solver. To remedy this problem, we suggest discounting a user based on the interference caused by the other co-running applications. The greater the interference, the greater the discount. The green bars show one such scheme. Because co-location increases machine throughput per unit time, these discounts can be viewed as passing the efficiency savings from co-location back to the end user when their expectation of service is violated.

Although the concept of progressive discounts is simple, the realization of such a policy on real systems poses a number of practical challenges. In particular, a *fair* pricing model of this nature requires precisely quantifying the interference due to shared resource contention. While there has been significant research into predicting cross-core interference, many of the techniques make heavy use of static profiling or have been tailored to specific machines or applications [84, 85, 164]. Even though this work has yielded considerable insight into the problem of shared resource contention, we argue that in practice, it is not practical for precise pricing on a real HPC cluster. In this domain, static profiling and machine- or application-specific approaches are not suitable as jobs may run very shortly after submission and their characterizations may not be known a priori. Although application profiling may enrich the solution space, we note that altering even a single input parameter for an application can vastly change its characteristics. For example, doubling a single array dimension can often radically transform an application's sensitivity to and aggressiveness on the memory subsystem. Thus, an instantaneous and dynamic mechanism is needed to continuously monitor and quantify the interference jobs suffer to drive precise pricing.

In addition to being dynamic and precise, the fundamental pricing mechanism

must also be lightweight. The underlying pricing agent has to be mostly invisible to the application and therefore must have a negligible overhead, below the system noise threshold. These objectives lead us to the two key insights of the work – only a software system that uses empirical, online tests is suitable for this problem domain, and such an approach must be agnostic to the underlying software and hardware.

In this work, we present such a solution: the Persistent Online Precise Pricing Agent (POPPA). POPPA is a lightweight runtime system that utilizes a cyclic, fine-grain, interference sampling mechanism to accurately deduce the interference between co-runners. The key design feature of POPPA is a dynamic contention detection technique we call **shuttering**. For brief periods of execution, POPPA pauses all applications but one and measures how the selected application's performance changes versus running co-located. From the disparity between the application's rate of forward progress made while running co-located versus shuttered, POPPA is able to precisely determine the impact of interference resulting from co-location and use these measurements to drive fair pricing for all users' jobs.

The contributions of this work are as follows:

- We introduce POPPA, a lightweight, workload and machine agnostic runtime system that enables fair pricing for HPC clusters. POPPA functions entirely in software, requires no changes to the system stack in current HPC clusters, and is readily deployable.
- We present the design of *precise shuttering*, a mechanism for the precise online measurement of the performance impact of cross-core interference. Our precise shuttering approach functions dynamically and requires no a priori knowledge or profiling of the applications.
- We present a new pricing model for HPC clusters based on POPPA to provide fair pricing to users.

- We provide a thorough evaluation of POPPA's efficacy and robustness as the central accounting mechanism on HPC clusters with a mix of MPI benchmarks and real workloads.

POPPA predicts co-located application run time with 4% mean absolute error and incurs less than 1% overhead. Using POPPA, we are able to discount the average user by 7.4% and deliver a pricing distribution that closely resembles that of an omniscient oracle.

3.1 Background and Motivation

In order to better understand why fair pricing is of such importance, we must first explore the current state-of-practice in accounting on supercomputers. We start by examining the accounting and allocation model found in the United States Department of Energy Office of Science INCITE program [22] and the National Science Foundation XSEDE program [21], two of the largest U.S. programs that provide resources to the general HPC research community. Each of these programs facilitates access to a number of large scale computing infrastructures. To successfully obtain an allocation, researchers submit grant proposals and, after reviews, are awarded time on those systems as a finite number of **service units (SUs)**. When a user runs a job on a system, they deplete their bank of SUs at a rate proportional to the length of their programs' execution and the number of compute nodes that they request.

In this model, users need strong guarantees that the value of an SU will not be negatively affected by other users' jobs running on the same computing resources. Similarly, supercomputer administrators care about user satisfaction and are incentivized to provide users with the best possible experience because individual supercomputing centers are awarded funds largely based on the success and popularity of their facilities.

Consequently, we observe that **throughout all levels of the funding ladder, fair pricing and accounting are crucial concerns**. Regardless of what mechanisms are implemented to improve supercomputer performance, energy efficiency or fault tolerance, they must not pervert the fairness of the pricing scheme.

3.1.1 MPI Programming Model

Most large scale scientific applications utilize the Message Passing Interface (MPI) as the core abstraction to facilitate workload distribution across a cluster. Two main characteristics of MPI programs are as follows:

1) **Single Program Multiple Data (SPMD)**: MPI processes execute the same static program binary and use unique identifiers called ranks to dictate communication patterns as well as which blocks of code get executed by different processes. While this allows for a large amount of potential diversity between processes, in practice most MPI programs are Single Program Multiple Data (SPMD): all processes execute the same core algorithm on different data. Thus within an MPI program, all the processes have high similarity, e.g., they all compete for the same resources.

2) **Tightly coupled communication synchronization**: The vast majority of MPI programs exhibit tightly coupled communication synchronization. Because of this tight synchronization, processes must execute in relative lock-step. If a process reaches an explicit or implicit barrier before the other necessary parties, it must wait until all others make similar progress before proceeding.

3.1.2 Co-location of MPI programs

When we reason about the nature of MPI programs, it quickly becomes evident that executing a single MPI program across a private set of compute nodes is an inefficient use of system resources. The *homogeneity between MPI processes* and the fact that *they*

are tightly coupled mean that many processes will execute the same program regions with high concurrency. When this happens, there is high risk for resource contention and performance degradation – homogeneous processes have high propensity to evict one another's data in the shared last level cache (LLC), contend for the memory controller, saturate off-chip bandwidth to main memory, and cause a backlog of messages for internode communication.

Previous research shows that homogeneous MPI processes can degrade one another's performance by more than 2x [58, 141]. In addition, these works show that introducing heterogeneity in workloads by co-locating multiple MPI programs on disjoint cores can drastically improve performance and energy efficiency. In fact, both studies find that aggregate throughput increases by 12 to 23% on average over the current state of practice, and [58] shows that system energy efficiency increases by 11 to 22%.

In conclusion, given the high cost of large supercomputers and the great performance and efficiency benefit of co-location, it is essential that we provide fair pricing mechanisms to make co-location practical.

3.2 POPPA Overview

In this section, we present the overview of the Persistent Online Precise Pricing Agent (POPPA) framework. Our primary design objective for POPPA is to provide accurate performance interference estimates for parallel applications with negligible overhead. As shown in Figure 3.2, POPPA consists of a main monitoring agent called the Controller and a series of Execution Managers.

Execution Manager: Each Execution Manager is responsible for launching and overseeing the entire execution of a parallel application on a given machine. The Execution Managers read from the central job queue and select the next job to run according to the job priority and its resource needs. An Execution Manager launches

the selected job and attaches a performance monitoring context (PMC) to the job. The PMC monitors the job performance by reading and evaluating appropriate hardware performance counters. During execution, the Execution Manager updates and reports the current status and performance data of the job to the Controller.

Controller: The Controller is the main component of POPPA. Its principle responsibility is to conduct **shuttering**, a mechanism to measure and quantify the performance interference among the co-running applications. In essence, the Controller periodically pauses each application but one for a very short period and monitors the performance impact on the lone running application. To measure this impact, the Controller probes the PMCs of each active job to acquire the performance data and logs it. We present more details of the shuttering mechanism including our algorithms and policies in Section 3.4 and evaluate its accuracy and overhead in Section 3.7.

Figure 3.2 presents how POPPA can be used for pricing. After execution of a job has completed, the Pricer thread analyzes the raw performance data logged by the Controller and quantifies the performance interference and degradation. More details of the analysis and pricing are presented in Sections 3.3 and 3.5. Based on the quantification, the Pricer produces the price to be charged and propagates it to the Account Manager, which then deducts the price from the user's bank of SUs.

3.3 Pricing Model

In this section, we discuss the key issues related to pricing and accounting on current supercomputers and extend those notions to a supercomputer with job co-locations.

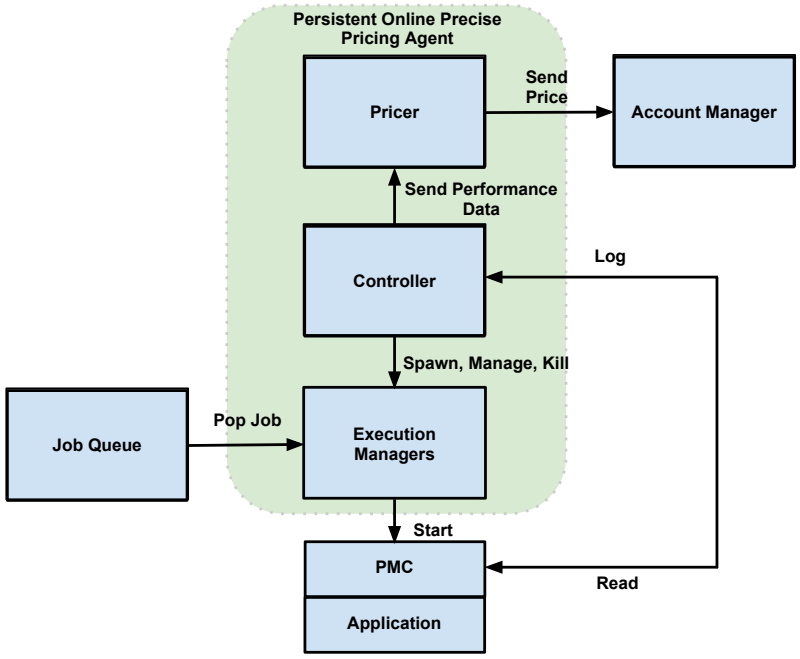


Figure 3.2. Interaction between POPPA components and other entities

3.3.1 Pricing Without Co-location

For purposes of this discussion, assume that a user wants to run a job i on a supercomputer and that P_i denotes the price that the user is charged for running i.

In present day systems, P_i is given by Equation 3.1, where L is a rate constant in terms of service units per core per time quanta, C_i is the number of cores that a job uses in whole compute node increments, and T_i is the run time of the program.

$$P_i = L * C_i * T_i \tag{3.1}$$

From this equation, we can see that the price variable P_i is linearly proportional to both the cores variable C_i and the time variable T_i.

3.3.2 Pricing With Co-location

In this section, we propose how one could modify the existing pricing model to more fairly price applications when co-locations are present. In particular, if we have a job i that is co-located with a set of jobs J, we want a formula that will produce a reasonable price $P_i^{co(J)}$, which takes into account the net interference from all applications in J. To this end, we replace L with a rate function F, yielding Equation 3.2, where $F : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. T_i^{solo} is the run time when the job i gets all compute nodes to itself and $T_i^{co(J)}$ is the run time of the job i when i is co-located with the set J of other jobs.

$$P_i^{co(J)} = F(T_i^{solo}, T_i^{co(J)}) * C_i * T_i^{solo} \tag{3.2}$$

Ideally, F is monotonically non-increasing so that the more degradation an application suffers from co-location, the more the user is discounted. For the purposes of this work, we assume utility is proportional to 1 minus the rational degradation. Therefore if we equate utility to fairness, then we select F such that users are discounted at a rate proportional to the degradation that each of their jobs experiences due to contention from co-runners. Thus if $D_i^{co(J)}$ is the degradation, then we want $P_i^{co(J)} = (1 - D_i^{co(J)}) * P_i^{solo}$. Consequently we define F as follows:

$$F(T_i^{solo}, T_i^{co(J)}) = L * \frac{T_i^{solo}}{T_i^{co(J)}} = L * (1 - D_i^{co(J)}) \tag{3.3}$$

By substituting Equation 3.3 into Equation 3.2 we see that we achieve the specific pricing model shown in Equation 3.4.

$$P_i^{co(J)} = L * \frac{T_i^{solo}}{T_i^{co(J)}} * C_i * T_i^{solo} \tag{3.4}$$

While Equation 3.4 is good for the user, we acknowledge that it is an idealistic

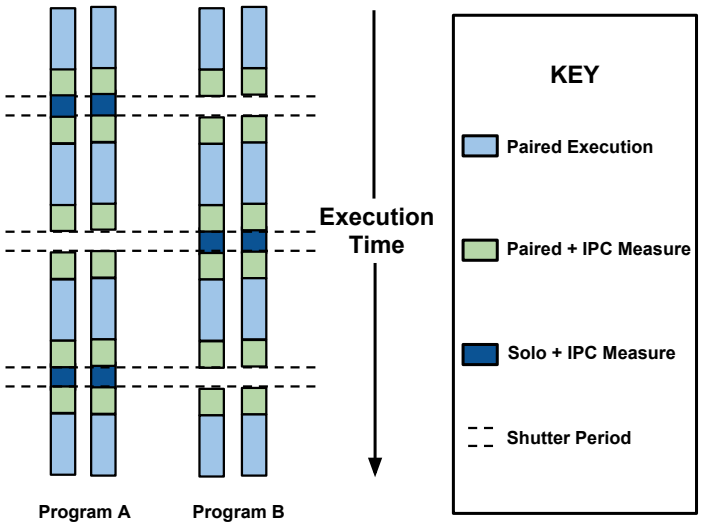


Figure 3.3. Shown here is shuttering in action on two separate jobs. During a shutter, one job executes while all others sleep.

model. Its simplicity makes it easy for end users to understand; however, we note other factors such as resource manager queue wait times, job priority, workload composition, the ratio of each shared resource a job consumes, machine architecture, and scheduling policy, i.e. capability versus capacity are also important factors when determining a fair price. Thus supercomputing facilities will have to decide what F makes sense for each of their systems.

3.4 Precise Shutter Mechanism

As previously mentioned, POPPA's chief design objective is to produce fair prices with high precision, low overhead, and without the need for a priori knowledge. To achieve these goals we have designed *precise shuttering*, an online co-runner interference masking approach. Essentially, the precise shuttering mechanism functions by alternating an application's execution environment between one where co-runners are executing and another where they are effectively absent.

Figure 3.3 shows shuttering in action on two applications A and B that are co-

located. The shuttering algorithm alternates between execution regions where A and B co-execute, A executes while B sleeps, A and B co-execute, and B executes while A sleeps. We repeat this pattern throughout the execution of the programs.

To gain insight from shuttering, we must measure the performance of each application before, during, and after shutter regions. During each shutter of duration $\mathcal{S}$, we leverage hardware performance monitors via `libpfm4` [15, 86] to measure the instructions per cycle of the sole non-sleeping application. To infer the degradation due to co-runners, we also measure the instructions per cycle (IPC) of all active applications $\mathcal{S}$ microseconds before the shutter and $\mathcal{S}$ microseconds directly after it.

Since we are primarily concerned by how performance changes with the presence or absence of contention, we only need to monitor the performance during small windows around shutters. We also perform each shutter infrequently to minimize the perturbation of application execution and parameterize the rate of shutter samples to control POPPA's overhead. As we show in this work, frequent shutters are not required to produce an accurate predictive model.

Algorithm 1. *Measure(i, S, K)*

```
1: Initialize array perfValue of length |A[i]|
2: for k = 0 to K − 1 do
3:     for each thread t that is part of A[i] do
4:         perfValue[t] = ReadCounters(t)
5:     end for
6:     Sleep for S μs
7:     for each thread t that is part of A[i] do
8:         perfDict[t].append(ReadCounters(t)-perfValue[t])
9:     end for
10: end for
```

Algorithm 2. *Shutter_Core(j, S, K)*

```
1: for i = 0 to |A| − 1, where i ≠ j do
2:     for each thread t that is part of A[i] do
3:         Pause t
4:     end for
5: end for
6: Measure(j, S, K)
7: for i = 0 to |A| − 1, where i ≠ j do
8:     for each thread t that is part of A[i] do
9:         Resume t
10:        perfDict[t].append(THREAD_ASLEEP)
11:    end for
12: end for
```

Algorithm 3. *POPPA_Core*

```
1: j = 0
2: while true do
3:     for i = 0 to |A| − 1 in parallel do
4:         Measure(i, S, K)
5:     end for
6:     Shutter_Core( j, S, K)
7:     for i = 0 to |A| − 1 in parallel do
8:         Measure(i, S, K)
9:     end for
10:    j = (j + 1)  mod |A|
11:    Sleep  Pμs
12: end while
```

3.4.1 Algorithms

In this section, we present the logic of the shutter mechanism, whose core parts are shown in Algorithms 1, 2 and 3. Below we define a list of common data structures and constants used by the algorithms:

- A, an array of co-located applications
- *perfDict*, a lookup table that stores the measured IPC values of each application
- K, the number of IPC measurements to make in a row in a specific region[2]

[2]We fix $K = 1$ for experiments and analyses in Section 3.7.

- S, the length of the each measurement in μs
- P, the length of time between groups of measurements, i.e. the normal execution period, in μs
- $\mathcal{S}$, the length of a shutter, approximately $K * S$

The core routine is Algorithm 3. At each iteration, we first measure the IPC of each application while co-located (lines 3-5). We then shutter application j by calling *Shutter_Core* (line 6), which subsequently calls *Measure* to measure the IPC while j is running alone. After that, we measure the IPC of all applications and increment j (lines 7-10). Then the shutter component of POPPA goes to sleep for $P\mu s$ of normal execution (line 11). Since POPPA is persistent, this process repeats continually as applications end and new applications enter the application pool.

3.4.2 Tuning the Shutter Mechanism

The shutter implementation presents a number of challenges. In particular, selecting the correct granularity to shutter at is key to accurately quantifying interference without noticeably adding to it. The first parameter is the gap between shutters P. As P is decreased, the amount of time that POPPA is active increases, consequently also increasing overhead. Since utilization in supercomputers is often above 95%, we assume that each core has an application thread assigned to it. Due to this fact, POPPA must time slice with application threads. If POPPA is active for x% of a single core's execution time, then assuming POPPA threads do not migrate, one of the co-running applications is likely to suffer at least an x% hit to performance due to synchronization between processes.

Since the POPPA runtime inevitably has overhead, we experimented with conducting round-robin migration of the POPPA threads to distribute the performance impact of time slicing across all application threads; however, we determined that a better solution was to select values for K, P and S that make POPPA's CPU utilization very low, as

migration is not guaranteed to be fine-grain enough to mitigate the effect of time slicing.

Another important parameter is $\mathcal{S}$ the duration of a shutter. In our implementation, this quantity is equal to the base cost of doing a shutter on 8 MPI processes, approximately 120 to 200μs (see Figure 3.4 in Section 3.7.1), plus $K * S$, where $K * S$ is the product of the number of consecutive measurements and the length of each such measurement. During a shutter, the paused application makes no progress, thus keeping shutter duration very short relative to P is a primary concern.

An unexpected find relating to the shutter mechanism is that in certain cases, POPPA actually slightly improves the performance of co-located applications. During shutters, applications that sleep sacrifice a small amount of forward progress and the lone runner receives a performance boost from reduced contention. When the net performance boost from running in isolation offsets the net performance loss from sleeping, applications speed up relative to the baseline co-schedule performance. For pairs of two applications, speedup occurs when a co-schedule increases one application's run time by more than 2x relative to running with half the cores idle per socket. This phenomenon is demonstrated empirically in Section 3.7.2.

3.5 Estimating Degradation

In this section, we present our method for linking the raw data that POPPA produces to the actual prices we charge.

3.5.1 Idealized Model for Degradation

Our pricing model assumes that for an application i, we know the degradation $D_i^{co(J)}$ that i suffers as a result of co-location with a set J of applications. In our pricing model discussion, we formulated $1 - D_i^{co(J)}$ as $\frac{T_i^{solo}}{T_i^{co(J)}}$. While this gives us a precise way to calculate degradation, POPPA cannot directly measure T_i^{solo}. Thus, we modify the

formulation such that it is amenable to the IPC data that POPPA produces.

On modern chip multiprocessors, if we are given an execution time in seconds, we can convert this to a value in clock cycles. Thus if we know the clock ticks per second, we can write the performance of i normalized to running alone as the ratio of clock cycles C_i^{solo} and $C_i^{co(J)}$(see below).

$$Perf_i^{norm} = 1 - D_i^{co(J)} = \frac{C_i^{solo}}{C_i^{co(J)}} \tag{3.5}$$

Additionally, if we assume i to be a truly serial program, then it is the case that i's dynamic instructions I_i do not change. Thus $I_i^{solo} = I_i^{co(J)}$, and consequently we can transform Equation 3.5 into a ratio of IPCs by multiplying by $\frac{I_i^{co(J)}}{I_i^{solo}}$, yielding the following:

$$Perf_i^{norm} = \frac{IPC_i^{co(J)}}{IPC_i^{solo}} \tag{3.6}$$

3.5.2 Known Challenges with Parallel Programs

For parallel programs, however, it turns out that Equation 3.6 is often imprecise. Many parallel programs contain mutexes, semaphores, and other locking mechanisms to enforce program correctness by preventing data races. When a load imbalance occurs, that is, one parallel process advances faster than its siblings, these locking mechanisms can distort both dynamic instruction count and CPU clock cycles.

With MPI, this issue is quite prevalent. If a communication routine is implemented as blocking, then it is common practice to have the thread that initiated the routine to poll for a certain number of cycles and then sleep. During this polling period, the thread executes a while loop where it continually tests whether the communication operation has completed. If the thread fails to finish the communication operation within a certain interval, it is put to sleep and signaled to wake up when the operation has

completed. Because contention and background noise on the system can cause this polling period to change in duration, the number of dynamic instructions attributed to these communication regions is variable. With MVAPICH2, the MPI-2 implementation, the maximum polling period can be adjusted [194]. While we were tempted to disable polling, we knew that doing so would be disadvantageous. In particular, polling greatly increases individual application performance because the blocking thread avoids the performance hit associated with going to sleep and waking back up, as it can proceed as soon as communication has finished. Thus, we decided to keep the parameters that maximized performance even though it made precise prediction more challenging.

3.5.3 Filtering

Even though Equation 3.6 is imprecise in the presence of variable execution, we find that in practice, it is still sufficient for producing reasonable degradation estimates. We also assume that the average over the N IPC samples that we collect is roughly equivalent to the actual average IPC during shutters (IPC_i^{solo}) and during normal paired execution (IPC_i^{co}). These assumptions are presented below in Equations 3.7 and 3.8.

$$Perf_i^{norm} \approx \frac{IPC_i^{co(J)}}{IPC_i^{solo}} \tag{3.7}$$

$$IPC_i^{solo} \approx \frac{\sum_{j=0}^{N_i^{solo}} IPC_{i,j}^{solo}}{N_i^{solo}} \text{ and } IPC_i^{co} \approx \frac{\sum_{j=0}^{N_i^{co}} IPC_{i,j}^{co}}{N_i^{co}} \tag{3.8}$$

POPPA gives us data in the form of a stream of blocks of IPC measurements, each consisting of K IPC measurements just before a shutter, K measurements during a shutter, and K afterward. We denote this stream of blocks as B and the lth such block as B_l; within each block B_l, the K IPC values in B_l before the shutter are denoted as IPC_l^{before}, the K IPC values during a shutter as IPC_l^{during}, and the K IPC values after a

shutter as IPC_l^{after}. Thus $B_l = (IPC_l^{before}, IPC_l^{during}, IPC_l^{after})$. We denote the arithmetic means of each of these values as $\overline{IPC_l^{before}}$, $\overline{IPC_l^{during}}$ and $\overline{IPC_l^{after}}$. Using this notation, we present the filtering algorithm (Algorithm 4) that allows us to increase the precision of the performance estimate.

Algorithm 4. *Filtered Prediction(IPC Tuples B)*

```
Initialize IPC^co and IPC^solo to 0
for each (IPC_l^before, IPC_l^during, IPC_l^after) in B do
    if |IPC_l^before − IPC_l^after| < δ and IPC_l^before < IPC_l^during and IPC_l^after < IPC_l^during
    then
        IPC^co ± = 0.5(IPC_l^before + IPC_l^after)
        IPC^solo ± = IPC_during
    end if
end for
Return (IPC^solo − IPC^co) / IPC^solo
```

Algorithm 4 aims to reduce noise from sampling IPC. It removes groups of IPC values where the IPC during a shutter is not greater than the IPC directly before and after. Since a shutter can only relieve shared resource contention, the IPC during a shutter should always exceed the IPC before and after a shutter if all measurements occur during the same computational phase. The second mechanism, which states that the absolute difference in IPC before and after cannot exceed δ works to ensure that clusters that cross phase boundaries are removed. We empirically determined $\delta = 0.05$ to be a reasonable value.

3.6 Experimental Setup

This section describes our methodology. We ran our experiments on the Gordon Supercomputer [110, 183]. Each node is dual-socket. For each socket, there is an 8-core Intel EM64T Xeon E5 (Sandy Bridge) processor. Simultaneous multithreading is disabled [231]. The CPU frequency is 2.6Ghz, and each core has private 32KB instruction

Table 3.1. Benchmarks and applications used in the study. NAS parallel benchmarks appear in the top row in plain text, proxy applications in the middle in italicized font, and full-scale scientific applications at the bottom in bold face.

Benchmarks, *Miniapps* and **Applications**
Swim [17], ADVECT3D [192], `pcubed` [145]
NAS Parallel Benchmarks: CG, FT, LU, MG [35, 179]
Lulesh [129], *MiniGhost* [12], *MiniFE* [12], *NekBone* [14, 16]
GTC [158], **LAMMPS** [11], **MILC** [13], **POP** [6]

and data L1 caches, a private 256KB L2 cache, and each socket has 20MB of L3. There are 64GB of DRAM. Compute nodes run CentOS linux with kernel version 2.6.32. The interconnect is QDR InfiniBand with 8GB/s of bidirectional bandwidth, and the topology is a 3D torus of switches [20, 221]. Our applications and benchmarks are shown in the table that follows. These benchmarks and applications encompass a wide variety of scientific domains such as subatomic particle physics [13], plasma physics [158], molecular dynamics [11], ocean modeling [6], computational fluid dynamics [14, 16], shock hydrodynamics [129], finite element methods [12] along with various other numerical methods that are of high interest to the HPC community. We also note that GTC and MILC, in particular, use a substantial number of dedicated allocation hours on many leadership class machines.

We compile GTC, LAMMPS, MILC, POP, CG, FT, LU and MG with GNU compilers version 4.7 and MVAPICH2 version 1.7. LULESH, MiniGhost, MiniFE, and NekBone are compiled with PGI compilers version 11.9 and OpenMPI version 1.6.

In our experiments, we co-locate two 8 process MPI applications together on the same set of sockets. Each socket has half its cores run one application and the other half run the other. Applications co-run together for a minimum of 5 iterations of both applications. As soon as one application ends, we immediately restart it. Data collection stops once both applications have completed 5 iterations. For the shutter mechanism, we

fix $K = 1$ and $P = 200$ms.

3.7 Evaluation

In this section, we evaluate the accuracy, overhead, and the pricing fairness of POPPA.

3.7.1 Quantifying POPPA's Base Overhead

In this section, we quantify the minimum time to execute components within the main loop of the POPPA daemon. The main loop consists of the three core operations of Algorithm 3 – measuring the IPC of the application just prior to the shutter, issuing the shutter and measuring the IPC of the application during that window, and measuring the IPC of the application immediately following the shutter.

For these experiments, we co-locate two MPI benchmarks, an auto-generated loop from the `pcubed` benchmark suite and a busy loop, called the NULL co-runner, that runs for the duration of the `pcubed` loop. In POPPA, we set all of the sleep parameters to 0, so we can measure the minimum execution time for all subcomponents of the loop. During each iteration of the main loop, we measure its total execution time, the time to measure the IPC both before and after the shutter, the total execution time of the shutter, the time to send the `SIGSTOP` and `SIGCONT` signals, and the time to make the IPC measurements during the shutter.

Figure 3.4 presents the results. On the x-axis we vary the number of threads in each job. So 4 corresponds to four `pcubed` tasks bound to cores 0, 2, 4, and 6 and four `busy loop` tasks bound to cores 1, 3, 5, and 7. The y-axis shows the total time in μs to execute the main loop. When studying this figure, several interesting trends emerge. Not surprisingly, adding more threads increases the minimum loop execution time. Execution time is dominated by IPC measurement in the form of calls to `libpfm`,

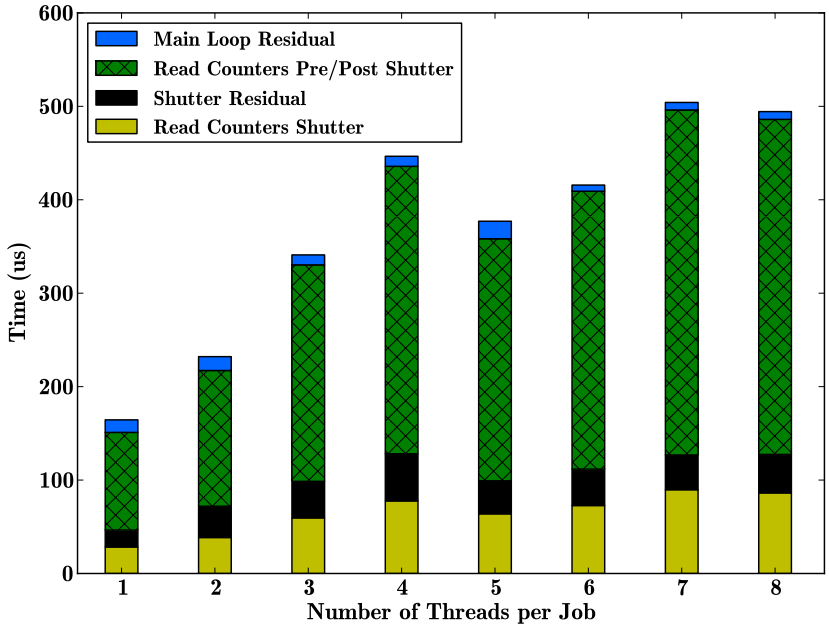


Figure 3.4. Breakdown of the base overhead to execute a single iteration of POPPA's core algorithm, where reading PMC values dominates total time

particularly those outside the shutter region. In fact, we spend about 4x as much time measuring the IPC outside of shutter regions compared to within them. This difference in overhead results from 1) we only measure active threads within a shutter, which is an optimization decision that we made, so the overhead to read the performance counters doubles outside of a shutter, and 2) we make two sets of IPC measurements outside of a shutter (before and after) versus a single set of measurements during one.

We see that the mean time to shutter does not exceed 130μs and the mean time to execute the main loop does not exceed 500μs. Thus, our mechanism is fine grained enough to measure the IPC at sub-millisecond intervals for thread counts that are representative of contemporary multi-socket systems.

In addition to the minimum delays incurred by shuttering, we quantify the effect of enlarging the amount of time spent in a shutter. For this experiment, we fix the sleep time at the end of the main loop, P (see Section 3.4.1), to 200,000μs and increase the

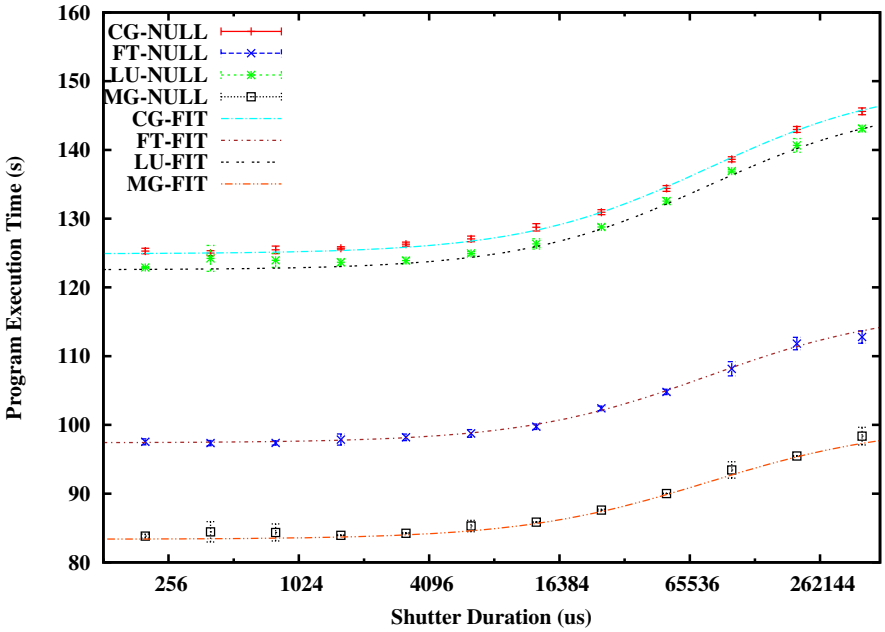


Figure 3.5. The relative overhead of expanding the duration of a shutter, where points correspond to measurements and lines correspond to instantiations of the model

shutter duration, $\mathcal{S}$ (see Section 3.4.1), multiplicatively by factors of 2 from 200μs to 409,600μs. We separately co-run each of the NAS Parallel Benchmarks (NPB) with the busy loop NULL. Since NULL generates no interference, any dilation in run time is a direct result of increasing the shutter window.

Figure 3.5 presents the results. All four benchmarks exhibit a similar trend. When $\mathcal{S}$ is small relative to P, the overhead is small, but as the ratio $\mathcal{S} : P$ increases, so does the overhead. However, the overhead begins to flatten out as $\mathcal{S}$ approaches and exceeds the value of P.

We need to formulate an analytical model for the overhead that a pricing shutter creates for an arbitrary co-located pool of n jobs. To do so, we examine the overhead from n consecutive shutters. Over the course of n shutters, each job will run in isolation once and sleep $n-1$ times while a single other job enjoys the privilege. Each such shutter has duration $\mathcal{S}$. Thus each job will sleep for $(n-1)*\mathcal{S}$ seconds.

The total time for n iterations of the main loop of the daemon is also important for the analysis. Measuring the IPC before, during and after a shutter is $3\mathcal{S}$, as each takes $\mathcal{S}$ time. After this, the daemon sleeps P seconds. This pattern is cyclic, so the combined time is $n*(3\mathcal{S}+P)$. Equation 3.9 shows ratio of sleep time to total time.

$$Z(\mathcal{S},P) = \frac{\text{sleep time}}{\text{total time}} = \frac{(n-1)*\mathcal{S}}{n*(3\mathcal{S}+P)} \tag{3.9}$$

The model for the execution time of the jobs in Figure 3.5 is shown below:

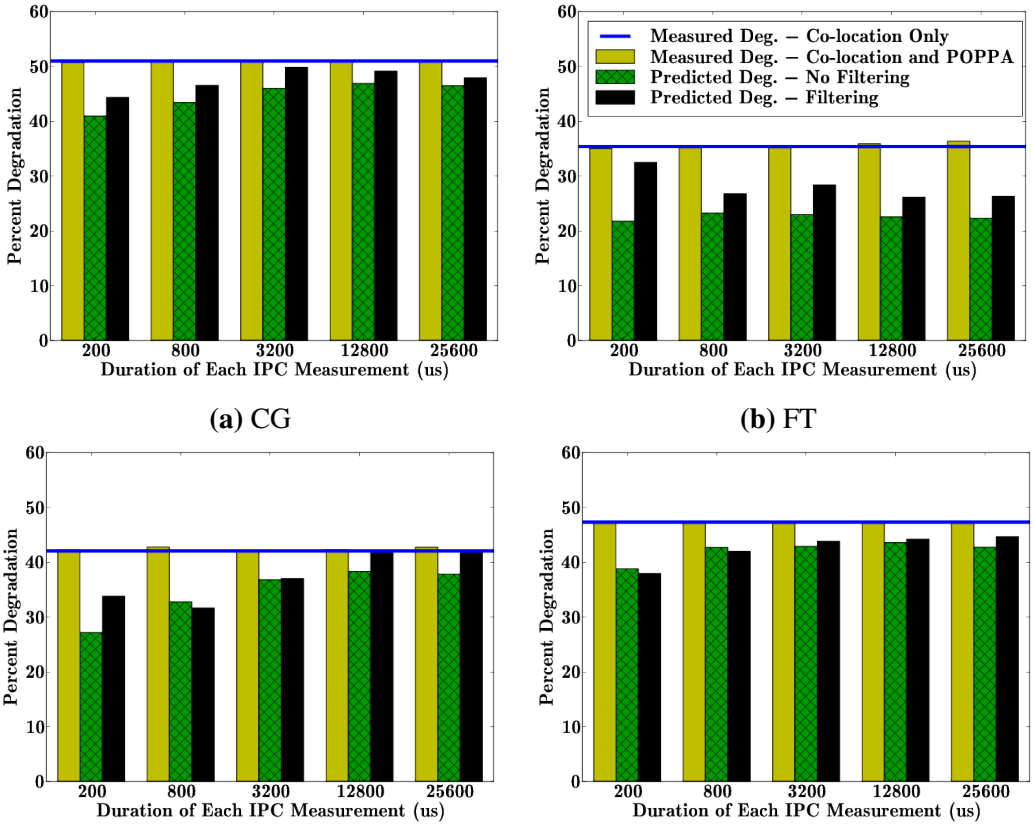
$$T(\mathcal{S},P) = T_i * \frac{1}{1-Z(\mathcal{S},P)} = T_i * \frac{n*(3\mathcal{S}+P)}{2n\mathcal{S}+nP+\mathcal{S}} \tag{3.10}$$

Here T_i is the run time of application i when co-located with the NULL co-runner. When we examine the model fit to the data in Figure 3.5, we observe that `CG-FIT`, `FT-FIT`, `LU-FIT`, `MG-FIT` almost exactly predict the actual overhead of the shutter for all $\mathcal{S}$ in $\{100*2^k\mu s | 1 <= k <= 12\}$ and a fixed P of 200ms. This model incorporates $\mathcal{S}$, P, and T; if we know any two of these quantities, we can solve for the third. Thus administrators can decide on a system by system basis what is exactly an acceptable amount of degradation due to the pricing shutter and choose values of $\mathcal{S}$ and P accordingly.

3.7.2 Determining the Sampling Rate

In this section, we evaluate the precision and overhead of the POPPA daemon for different shutter lengths ($\mathcal{S}$ values) while keeping P fixed to 200ms. We saw in the previous section, that the overhead due to the shuttering mechanism has an analytical upper bound given by Equation 3.10. Using this equation, we selected values of $\mathcal{S}$ with less than 5% overhead: 200, 400, 800, 1600, 3200, 6400, 12800, and 25600μs.

We ran two sets of pairwise experiments. In the first, we co-located the NPBs

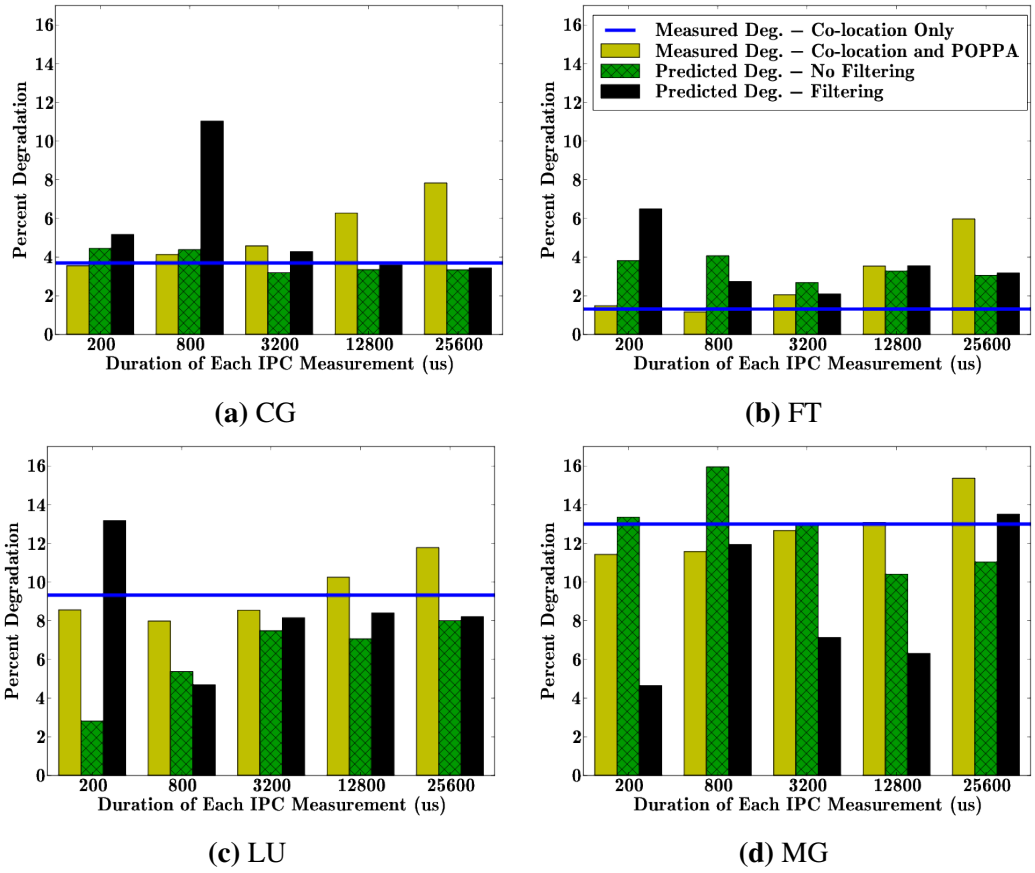


(a) CG **(b)** FT **(c)** LU **(d)** MG

Figure 3.6. Effect of shutter duration on accuracy and overhead for each NPB co-run with ADVECT3D-256

with a contentious co-runner (ADVECT3D with a grid size of 256^3), and in the other we co-scheduled the NPBs with a moderately contentious co-runner (Swim with a grid dimension of 150^3). Figures 3.6a, 3.6b, 3.6c, and 3.6d show the performance prediction accuracy of the POPPA daemon for CG, FT, LU, and MG when they are co-located with ADVECT3D. Both the accuracies of the unfiltered and filtered predictors are shown. For clarity, we opt not to present the results for 400, 1600 and 6400μs.

In this set of experiments, we are able to very accurately predict the contention with negligible overhead. Filtering improves prediction performance. Our predictors have the largest error for FT. $\mathcal{S} = 200\mu s$ gives the highest accuracy, but as $\mathcal{S}$ increases, so does the error. This error results from FT's very fine grain phases, which coarser



(a) CG **(b)** FT

(c) LU **(d)** MG

Figure 3.7. Effect of shutter duration on accuracy and overhead for each NPB co-run with Swim-150

granularity shutters have trouble capturing.

Figures 3.7a, 3.7b, 3.7c, and 3.7d show the prediction accuracy for the NPBs paired with Swim. Again, our prediction accuracy is very precise. In this case, we note that the filtered prediction is sometimes overly zealous when predicting contention. However, this result is unsurprising given that filtering removes clusters of IPC measurements where the IPC measured during a shutter does not exceed the IPC directly before and after.

A contrasting finding between the experiments with ADVECT3D and Swim concerns daemon overhead as a function of $\mathcal{S}$. In the experiments with ADVECT3D, overhead is flat regardless of $\mathcal{S}$ whereas it sharply increases with Swim. This divergence

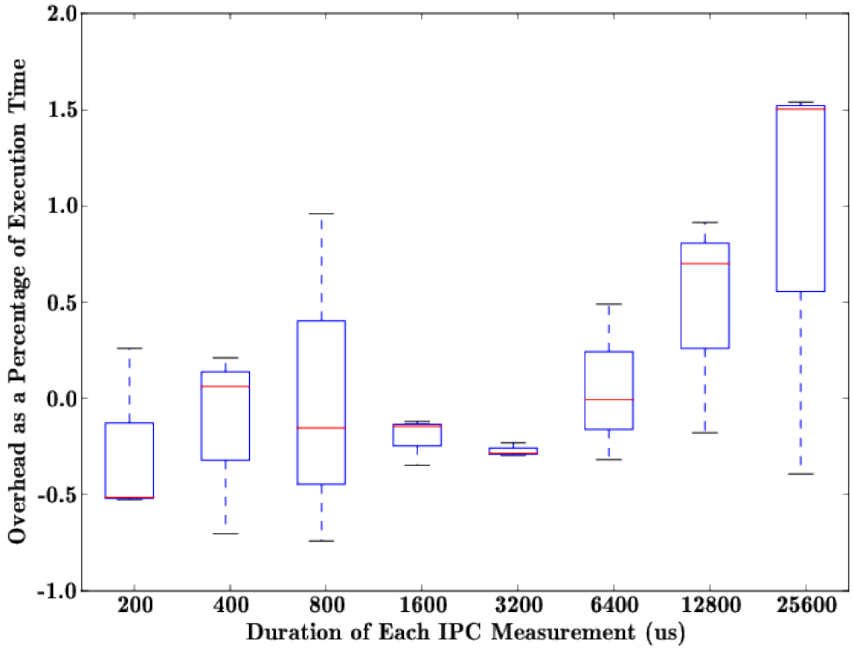


Figure 3.8. Overhead of POPPA on NAS benchmarks when co-located with ADVECT-256

is caused by the fact that ADVECT3D is configured to be contentious whereas Swim is not. During a shutter, the lone running application receives a respite from the contention generated by the other application. In the case of the NPBs with ADVECT3D, this causes each NPB to speed up by approximately 2x, which offsets the lost throughput from sleeping during alternate shutters. By contrast, Swim degrades each NPB by at most 15%, so the time spent sleeping cannot be masked.

These experiments show that the shutter duration $\mathcal{S}$ is largely irrelevant for accuracy. Thus when selecting $\mathcal{S}$, it makes sense to select a value that induces minimal overhead and run time variation. Figures 3.8 and 3.9 present both the daemon's overhead and its distribution for the surveyed values of $\mathcal{S}$. In Figure 3.8, regardless of the value of $\mathcal{S}$, overhead due to the pricing shutter never exceeds 2%. However, in Figure 3.9, this value exceeds 4%, which is clearly too costly. $\mathcal{S} = 3200\mu s$ delivers an overhead of less than 1% and with the smallest variation. For this reason, we use $\mathcal{S} = 3200\mu s$ for the

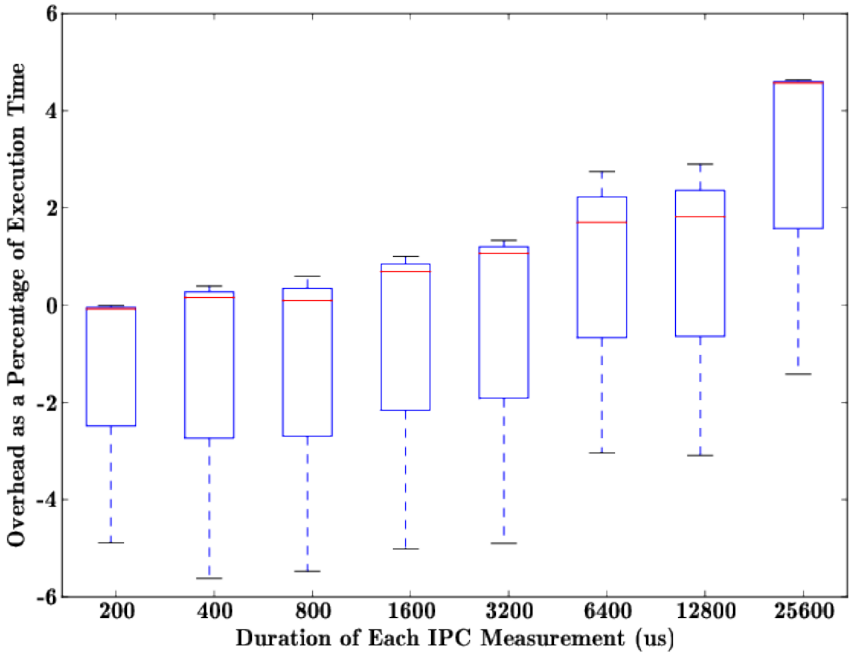


Figure 3.9. Overhead of POPPA on NAS benchmarks co-located with Swim-150

remainder of our experiments.

3.7.3 Pairwise Evaluation

In this section, we evaluate the precision of POPPA on pairwise co-locations. Since our filtered prediction was better in aggregate in our previous experiments, we apply that prediction mechanism rather than the simple one. We run co-schedules of all possible combinations of our 12 benchmarks and real applications.

Figure 3.10 shows the accuracy of our filtered predictor at quantifying degradation. The x-axis lists the names of the benchmarks, and the y-axis lists the co-runners. Individual cells present the percentage difference in predicted run time versus actual, where negative values represent underprediction and positive values represent overprediction. The top row "mean" presents the mean absolute error across the apps, and the right most column "mean" presents the mean absolute error that an application creates in the prediction accuracy for the other codes.

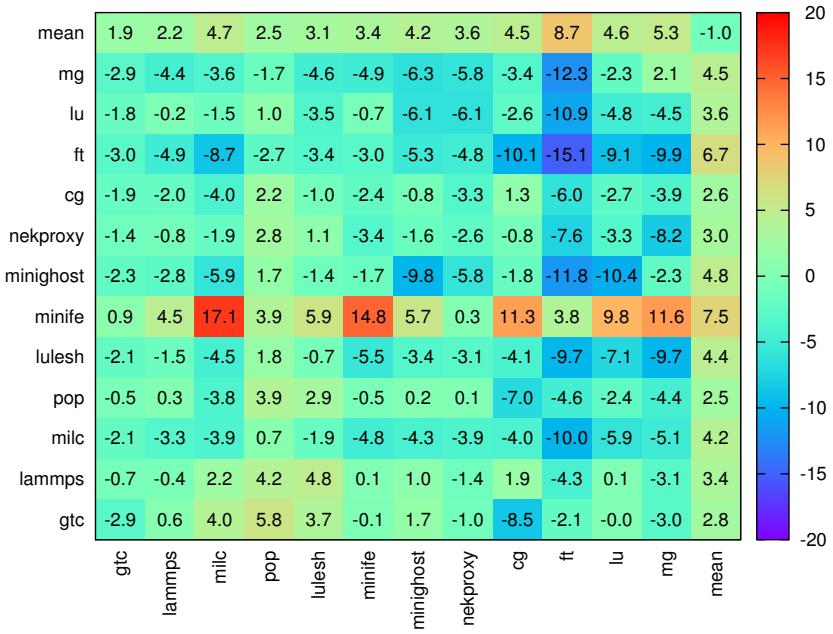


Figure 3.10. Run time prediction accuracy (%) for jobs on the x-axis co-located with jobs on the y-axis

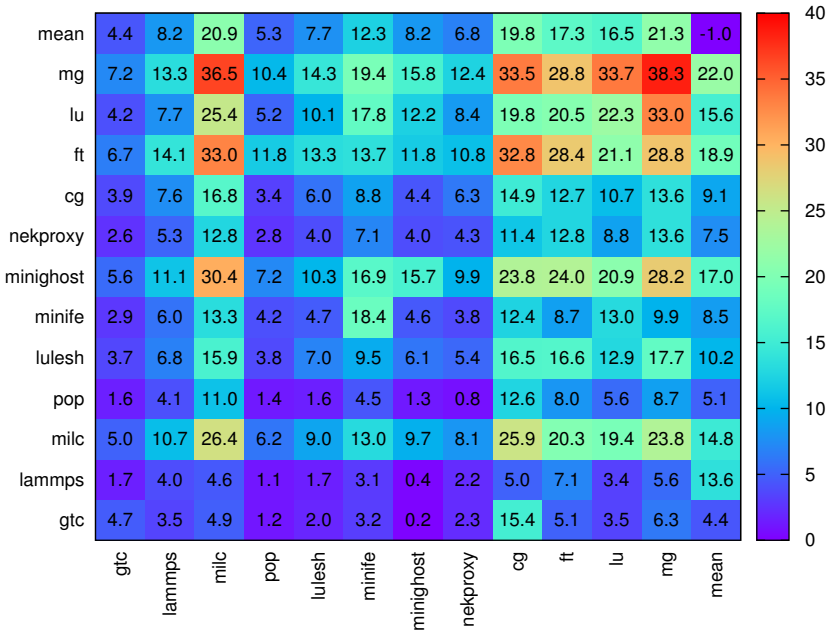


Figure 3.11. Performance degradation (%) for jobs on the x-axis co-located with jobs on the y-axis

Figure 3.11 presents the degradation of each application as a percentage of run time relative to running with the NULL co-runner, i.e half the cores vacant on each socket. The top row presents the mean degradation of each scientific code on the x-axis and the right most column presents the mean degradation each application on the y-axis causes to its co-runners.

If we study Figures 3.10 and 3.11 in concert, a number of interesting trends emerge. POPPA does well at quantifying degradation for all pairings consisting exclusively of our real applications, GTC, LAMMPS, MILC, and POP. Our mean absolute error is 2.5% and absolute error never exceeds 5.8%. We accurately characterize both ends of the spectrum. We predict high degradation for MILC paired with itself and we neither significantly underpredict or overpredict for pairings with low mutual contention such as GTC-LAMMPS and LAMMPS-POP. For pairings of real apps with benchmarks, the prediction accuracy is generally quite good except for when MILC is co-located with MiniFE and FT.

For our proxy apps LULESH, MiniFE, MiniGhost and NekProxy (NekBone), the results are more mixed. We are able to predict their performance with a mean absolute error of 3.8%. MiniFE is a particularly interesting because in each case we overpredict the degradation for its co-runner (mean of 7.5%). This overprediction is an artifact of the filtering algorithm. When we use our unfiltered predictor, we overpredict by at most 1.5% for MiniFE's co-runners. MiniGhost, by contrast causes us to underpredict contention for some of its co-runners.

On the NPBs, our prediction error is slightly higher. If we exclude FT, our mean absolute prediction error is within 5.3%. FT however, poses challenges both for its prediction and applications it is co-located with. In both cases, we underpredict the actual degradation. This underprediction is due to the duration $\mathcal{S}$ of the shutter. If we reexamine Figure 3.6b, we observe that $\mathcal{S} = 200\mu s$ yields the highest accuracy when

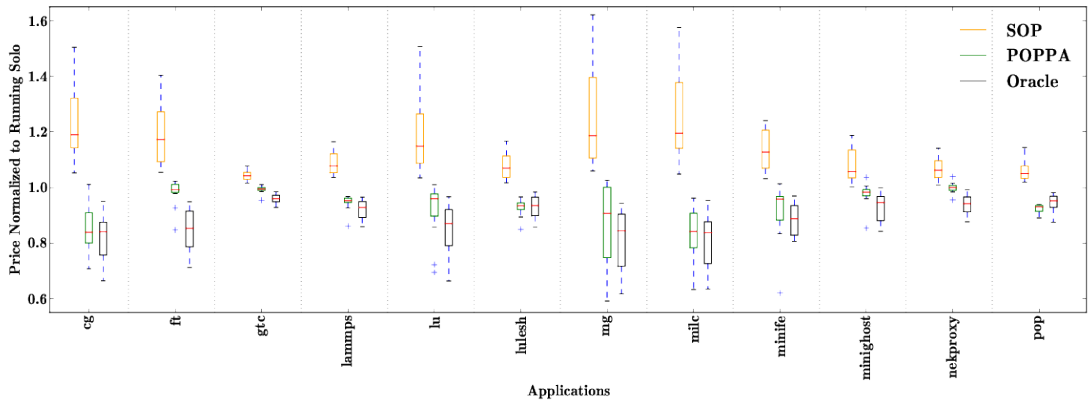


Figure 3.12. The distribution of prices a user would pay for a given application when using either the state-of-practice (SOP), POPPA, or the maximally fair Oracle

FT is co-located with a contentious co-runner. We also observe in Figure 3.7b that out of the possible values for $\mathcal{S}$, $\mathcal{S} = 3200\mu s$ prognosticates the lowest contention. On the whole, our system is generous and tends towards modestly underpredicting contention. Our mean absolute error across all pairings is 4.0%.

3.7.4 Pricing Fairness

In this section, we show POPPA's pricing fairness versus the state-of-practice and the oracle. Figure 3.12 shows the distribution of relative SUs charged for each application using the different pricing schemes. On average, the state-of-practice would charge users 14% more as result of co-locating their jobs. Jobs that degrade more, pay more. POPPA on the other hand discounts users by an average of 7.4%, which is close to the 11.5% discount that the oracle would offer.

When we examine the minimum and maximum relative SUs charged, we also see favorable results for POPPA. The maximum discount given by POPPA is 40.8%, which is close to the oracle's 38.3%. The max normalized price paid by a user using POPPA's counsel is 103.8% of the spread baseline versus the oracle's 99.8%. In the minority of cases where POPPA charges more than the spread baseline (23/144), it is usually smaller than run-to-run variation, with a mean surcharge of 1.3%. In addition,

the mean price paid for each application never exceeds 99.2% of the baseline, and thus over time, all users will receive a discount. Contrast this with the state-of-practice, where a user running MILC in the worst case can pay up to 62.1% more and on average would expect to pay 24.9% more as a result of cross-application interference.

If we consider the impact of POPPA's discounts, we find they are entirely tenable. Recall that the job striping study [58] found that co-locating MPI benchmarks and full-scale applications at scale increased mean system throughput by 12 to 23%. Thus discounting users by a mean 7.4% does not inflate the purchasing power of SUs, and so SU allocation need not be changed.

3.8 Related Work

There are a number of works that investigate pricing or identify pricing as a key issue for large scale grid and cloud infrastructures [32, 182, 212, 236]. Our work differs from these works in that we address the pricing issue in supercomputers with co-locations. To the best of our knowledge, our work is the first to explore this problem space.

Although this work addresses challenges related to fair pricing, it shares similarities with research that addresses identifying and mitigating contention in multicore systems. Early work on simultaneous multi-threading processors investigated co-scheduling of heterogeneous threads [67, 214, 215] as a way to increase throughput by reducing contention.

Cross core contention has also been extensively studied [71, 165, 166, 249]. A mechanism similar to the pricing shutter is explored in [166] but differs in that it is in the commercial data center space and in that it focuses on L3 miss rates with and without the presence of contention.

Another solution to mitigating contention has been cache partitioning both in software and in hardware [72, 177, 196, 223]. Core fusion is an architectural design

that helps reduce the cross core contention problem by dynamically combining simpler cores into larger cores [119, 224]. Others have examined using scheduling to mitigate contention [48, 49, 94, 95, 242] and [189, 226] investigate scheduling considerations in mapreduce environments.

There are also studies that evaluate the effectiveness of analytical and statistical models to solve problems related to contention [82, 103, 156, 232]. The computational complexity, heuristics and approximation algorithms for optimal multiprocessor scheduling are explored in [51, 101, 124, 130].

3.9 Conclusion

We have provided a mechanism to enable fair pricing on HPC systems, one of the fundamental roadblocks to enable node sharing on HPC systems. By employing POPPA, we can accurately measure performance degradation across a range of MPI applications. Using this data, we price users in a fashion that approaches the optimal fairness provided by the oracle, and our mean absolute prediction error is 4% across all combinations of 12 application codes.

POPPA is not a definitive solution to the pricing problem but a key part of a more holistic solution. Going forward, the development of additional, light-weight techniques for application introspection will become essential. By harnessing this dynamic information, further optimization opportunities will arise. Through combining these solutions, the road to exascale supercomputers looks bright.

3.10 Acknowledgements

The authors thank the anonymous reviewers for their time and feedback. In addition, they thank Professor Mike Norman of UCSD, Professor Leo Porter of Skidmore College, and Terri Quinn of LLNL. Some of the ideas in this chapter were inspired by

discussions with the late Allan Snavely. Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-635977). This work was supported in part by the DOE Office of Science, Advanced Scientific Computing Research, under award number 62855 “Beyond the Standard Model – Towards an Integrated Modeling Methodology for the Performance and Power”; PNNL lead institution; Program Manager Sonia Sachs. The authors acknowledge National Science Foundation support under CCF-1302682.

Chapter 3, in full, is a reprint of the material as it appears in Proceedings The International Conference for High Performance Computing, Networking, Storage and Analysis 2013. Breslow, Alexander D.; Tiwari, Ananta, Schulz, Martin, Carrington, Laura N., Tang, Lingjia, Mars, Jason, November 2013. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Black Box Performance and Energy Optimizations for GPU-Accelerated Databases

Recent years have seen an exponential increase in the amount of data, with year-on-year growth of digital information estimated at 40% to 65% [3,18,163,218]. Designing scalable systems to sift through the data and derive insights poses a monumental challenge. At the same time, CPU performance scaling has tapered due to the end of Dennard scaling [79, 88, 233], and Moore's law [175] appears to be slowing [235]. Taken together, these trends mean that there is no longer a free lunch for performance and that database systems must scale out both within and between servers. In this work, we examine this performance scaling on GPUs, one of the many candidate accelerators for database acceleration.

Unlike general-purpose CPUs which have seen comparatively meager annual improvements in performance in recent years, GPUs have seen rapid increases in throughput. A study by Phoronix found that over the last 8 years GPU performance has increased by 15x under a roughly fixed power budget [144]. Further, the two principal limitations of GPUs, (1) draconian programming models that lead to difficult to optimize code and (2) narrow PCIe® buses that lead to complex, slow data marshalling, both seem

poised to greatly improve in the next five years. Forthcoming support for C++11 [1, 9] and higher-level programming abstractions (e.g., OpenMP 4.5 [167]) promise to reduce the programmer's burden. Further, the introduction of fast new interconnect technologies that offer superior latency and bandwidth to PCIe technology(e.g., NVLink [7], CAPI [222], and CCIX [5]), promise to significantly reduce the cost of moving data to and from the GPU. These two trends coupled with (1) an increase in the bandwidth, capacity, and energy efficiency of GPU memory (e.g., 3D memory technologies such as HBM [10, 135, 147, 209]) and (2) the inclusion of storage on GPUs (e.g., AMD Radeon™ Pro SSG GPU with an on-board terabyte capacity SSD [4]) herald a new era of unprecedented practical computing capability and promise to spur widespread adoption of GPUs for many tasks beyond traditional consumer graphics and scientific computing.

One of those application domains is analytical database workloads. Due to the current challenges of transferring data over a high-latency, low-bandwidth PCIe bus, much of the effort has been spent on orchestrating efficient data transfers between CPUs and GPUs and not on fundamental challenges associated with scaling GPU-side code. As a result, much of the GPU-side database code remains in a comparatively nascent, unoptimized state that is hidden by the cost of data transfers. However, with forthcoming interconnect technologies, much of these performance trade-offs are likely to change. Instead, performance scaling of GPU-side code will become much more significant to the overall throughput of the database.

In light of these current and forthcoming advancements, this work aims to assess where we are in terms of database performance scaling on GPUs and suggests an agenda for further research opportunities within the space. Our study consists of asking a simple question: do current GPU-accelerated databases take full advantage of the resources that GPUs afford? In particular, can most GPU-deployed database operators even make meaningful use of all of the available compute capability? We conduct our study by

incrementally disabling compute units (CUs), the processors of the GPU. Disabling CUs artificially shifts the compute-to-bandwidth ratio of the GPU and facilitates the emulation of different hardware design points within a single device. When we disable CUs, we measure the impact on low-level performance metrics. In addition, we model the projected impact on energy and power of using coarse-grain per-CU power gating, a technique whereby an entire set of CUs is switched off.

As a result of these studies, we make several important observations that form the basis of our technical contributions:

- Current GPU databases cannot effectively utilize all of the computational resources on a GPU. Disabling of CUs almost universally leads to reduced query execution time and energy.

- The primary source of this poor scaling is rampant contention for the last level cache (L2) and off-chip bandwidth that is so severe that it often leads to performance stagnation or collapse as progressively more CUs are enabled. On a TPC-H workload, disabling CUs reduces query latency by as much as 24%.

- Although GPUs are already quite energy efficient, we show that the TPC-H workload is particularly suited to coarse-grain per-compute-unit power gating, often reducing energy usage by 10% to 40% with little to no performance loss.

- We corroborate earlier findings that the scan primitive is particularly benefited by GPUs by showing that it scales better with the number of active CUs than the other operators we surveyed [193].

Our studies suggest that there is a rich opportunity space for building per-database-operator power-performance cost models for GPUs. These models would permit the database to reduce its energy usage per query by actively directing sub-device-level

power gating. Further, we posit that a high-performing GPU database will likely require integrating all of the optimizations from columnar databases and recent work on GPUs. Without these optimizations, our study shows that it is unlikely that performance will properly scale in rough proportion to the enabled CUs. New memory technologies like HBM that would deliver terabytes per second (TB/s) of memory bandwidth to a single GPU may reduce these problems.

4.1 Brief Background on GPUs

GPUs are massively parallel computing devices that use multiple multiprocessors known as **compute units** (CUs) to accelerate workloads. A CU contains one or more vector processors known as SIMDs. Since SIMDs are vector processors, each instruction that they run executes in lockstep across multiple **lanes**, where lanes are similar to simple cores. In the AMD Graphics Core Next (GCN) family of GPU architectures, CUs have four SIMD units with 16 lanes per SIMD. SIMDs share an L1 data cache and L1 instruction cache that is private to their CU [19]. Often, GPUs have several to tens of CUs per device that collectively share an L2 cache and bandwidth to the off-chip main-memory.

Work on the GPU is composed into blocks of threads known as **work groups**, and each work group is mapped at execution to a CU where it is processed. Work groups are further subdivided into **wavefronts**, effectively a collection of threads that share the same program counter. Typically, multiple wavefronts are assigned to each CU to hide latency: when one instruction from one wavefront produces a stall, typically due to a high latency load, the hardware can swap in another wavefront that is ready to run. Wavefronts are further decomposed into a number of coupled threads known as **work items** that share the same program counter. In general, each wavefront is a multiple of the width in lanes of each SIMD. In GCN-based GPUs, a wavefront consists of 64 work items.

During execution, each wavefront is consumed by one of the SIMDs of the CU to which it was assigned. Since SIMDs are only 16 lanes wide, and each instruction has 64 lanes worth of work, each instruction's work is processed in batches of 16 work items. Thus, each lane per SIMD processes four work items per wavefront [19].

To achieve high performance on GPUs, both **control flow divergence** and **memory divergence** must be avoided. Control flow divergence occurs when some lanes do not participate in executing an instruction due to program control flow semantics. It reduces the effective parallelism of the device because it leads to lanes being disabled and therefore also the number of active arithmetic logic units (ALUs). Memory divergence occurs when lanes within a CU need to load data from memory locations that cannot be combined by a hardware coalescer into a single memory request, thereby typically increasing the number of cache lines that need to remain or be brought into cache to satisfy each load. Severe memory divergence can lead to performance collapse due to cache pollution and excess demand for off-chip memory bandwidth.

Like CPUs, each of the successive levels of memory from registers down to the main memory have reduced bandwidth and higher latency the farther they are from the compute elements. Therefore, optimizations such as blocking for cache via partitioning [54] and batching (a.k.a. vectorization [55]) are also important for GPUs [109].

4.2 Experimental Methodology

We conduct our studies on an AMD Radeon Firepro W9100 GPU, which has 44 CUs, 1 MB of L2 that is shared among all CUs, 16 KB of private L1 data cache per CU, and 16 GB of GDDR5 (a DRAM technology that serves as the main memory of the GPU). Execution time and other performance metrics were collected with AMD's CodeXL profiler. To modify the GPU's configuration, we use a single internal tool that modifies the GPU's firmware. The GPU's CUs and GDDR5 memory are located in two

different clock domains. We fix both the compute and memory clock domains at their maximum values of 930 MHz and 1250 MHz, respectively, which reduces run-to-run time variation due to dynamic voltage frequency scaling. In the studies, we disable CUs in multiples of 4. When CU are disabled, the L2's capacity is not reduced.

Ocelot [112] is the database we use; it is an extension to the in-memory columnar database MonetDB [53] that allows for offloading the bulk of query processing to the GPU. We chose Ocelot over other GPU-accelerated databases for several key reasons:

1. Ocelot aims to execute the full query on the GPU. Unlike many other candidates, this means that it gives a more balanced picture of the opportunities that GPUs afford for query processing.

2. Because Ocelot leverages many of the optimizations that MonetDB and other in-memory databases pioneered (e.g., dictionary encoding, bitmap joins, columnar storage), its evaluation provides insight into how well these optimizations perform on the GPU and directions for further research.

3. Ocelot aims to be cross platform. Given the enormous cost associated with optimizing code for discrete platforms, it is necessary to see how well such a system can perform.

4. Ocelot achieves rough performance parity with newer works like GPL [187], which demonstrates that the implementation is comparatively high quality.

Our queries are the modified TPC-H queries that come standard with Ocelot [112], apart from queries 9 and 14, which are from the GPL work [187]. All experiments are conducted on a database with a scale factor of 10, a measure of the database size for the TPC-h benchmark. We conducted additional experiments at a scale factor of 50, but the results were similar, and so we omit them. In addition, we explored dynamic

voltage and frequency scaling (DVFS) [125, 146, 239] and varied the CU and memory frequencies. However, we found that these DVFS studies had secondary, unintended effects beyond controlling the compute-to-bandwidth ratio of the GPU. For instance, reducing the CU frequency not only affected the vector arithmetic logic units (VALUs) but also the bandwidth of data buses and the L1 data caches. Thus, even though the workloads were bandwidth bound and in theory reducing compute frequency could improve energy efficiency with little reduction in performance, it had the secondary effect of reducing the rate at which caches could issue requests to the memory controllers and thus the peak memory bandwidth of the system. Similarly, reducing memory frequency was almost always a bad idea because most of the workloads were severely bandwidth bound, and so every bit of off-chip memory bandwidth was precious.

4.2.1 GPU Power and Energy Models

In this section, we briefly detail our power and energy models. Since disabling of CUs primarily affects the GPU, we do not report results for the rest of the server's power.

Our GPU does not support programmatic power gating at the CU granularity when CUs are disabled. Thus, we model power using an equation that we validated on one of our integrated GPUs that has both the same Graphics Core Next microarchitecture and that power gates CUs when they are disabled. Equation 4.1 calculates the combined mean GPU die and GDDR5 power during execution of a GPU kernel. It excludes GPU fan power and AC to DC conversions. We determine each of the parameters by running FurMark [8], a compute-intensive benchmark and profile power using an Extech 380801 power meter. *Idle Power Other* corresponds to idle power that corresponds to components other than CUs. *VALU Activity Factor*, the VALU activity factor measures the mean fraction of lanes that execute arithmetic instructions on their VALUs per

compute cycle. A low VALU activity factor reduces GPU dynamic power.

$$\begin{aligned} &Kernel\ Power\ =\ Enabled\ Compute\ Units\ *\ Idle\ Power\ Per\ CU \\ &+Idle\ Power\ Other\ +\ GDDR5\ Power \\ &+\ Enabled\ Compute\ Units*\ VALU\ Activity\ Factor\ * \\ &(Peak\ Total\ Power\ Per\ CU\ -\ Idle\ Power\ Per\ CU) \end{aligned} \tag{4.1}$$

The mean query power is computed using Equation 4.2, which computes an average over the mean powers of each of the kernels weighted by each one's execution time. Query energy is computed using Equation 4.3, which is the sum over the individual energy contributions of each of the kernels that is executed.

$$Query\ Power\ = \frac{\Sigma_{k=1}^{Kernels}\Big(Kernel\ Power_k\ *\ Time_k\Big)}{\Sigma_{k=1}^{Kernels}\ Time_k} \tag{4.2}$$

$$Query\ Energy\ = \Sigma_{k=1}^{Kernels}\Big(Kernel\ Power_k\ *\ Time_k\Big) \tag{4.3}$$

In addition to energy, we also compute **energy-delay product** [102] using Equation 4.4, where EDP is calculated as the product of the combined energy of each kernel and the total kernel execution time. Because EDP is the product of energy and delay (execution time), an improvement in EDP means that either (1) both energy and delay improve or (2) that the improvement in one metric multiplicatively offsets any degradation in the other. For example, a decrease in energy of 20% ($\frac{4}{5}$x) and an increase in delay of 25% ($\frac{5}{4}$x) would cancel out and lead to no change in EDP. In general, policies that decrease EDP relative to a baseline are considered favorable.

$$EDP\ =\ Query\ Energy\ *\ \Big(\Sigma_{k=1}^{Kernels}\ Time_k\Big) \tag{4.4}$$

4.3 Results

In this section, we present the impact of scaling the active CUs in the GPU. We begin by examining the breakdown of execution time for each query (Figure 4.1). In keeping with prior work, joins account for the majority of execution time for most queries. Queries 3, 4, 5, 7, 8, 9, 10, 11, and 21 spend upwards of 50% of their time executing joins. Queries 6, 12, 14, and 19 spend most of their time in scans, and only 15 and 17 are dominated by grouping operations.

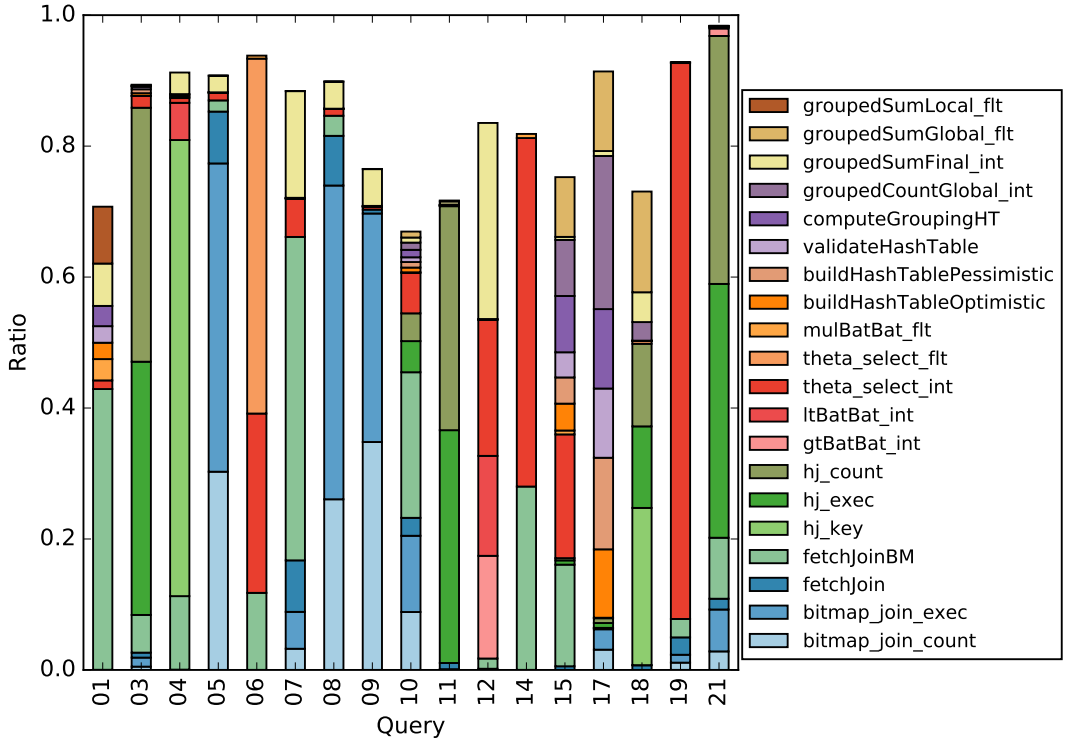


Figure 4.1. Ratio of execution time by kernel. Joins dominate execution time for most queries. Labeled kernels contribute at least 5% or more of the run time of one or more queries. Other kernels are excluded for clarity of exposition.

To better understand how query composition by operator affects run time, we continue by analyzing how their make-up impacts low-level metrics (derived from hardware

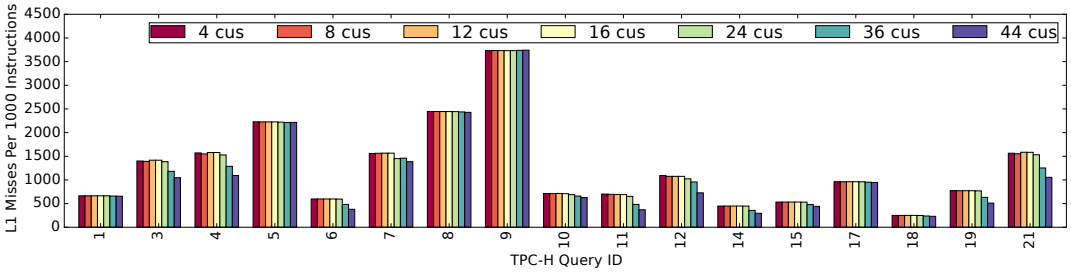


Figure 4.2. Private L1 cache misses per 1000 dynamic instructions (lower is typically better). Most of the queries exhibit many private L1 misses per 1000 instructions.

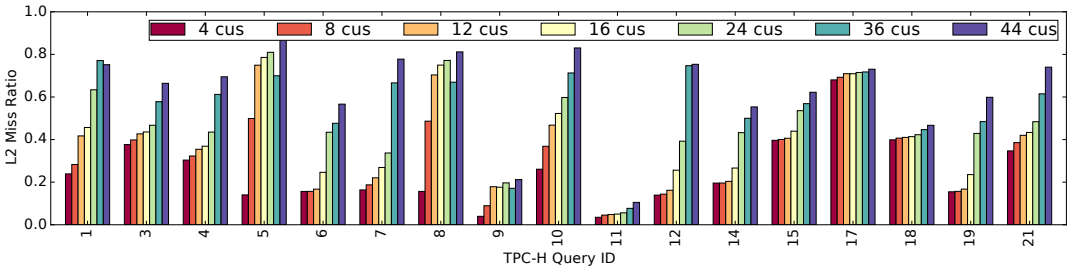


Figure 4.3. Shared last level (L2) cache miss ratio (lower is typically better). As the number of active CUs increases, for most queries, pressure on the last level L2 cache balloons and so does the miss ratio.

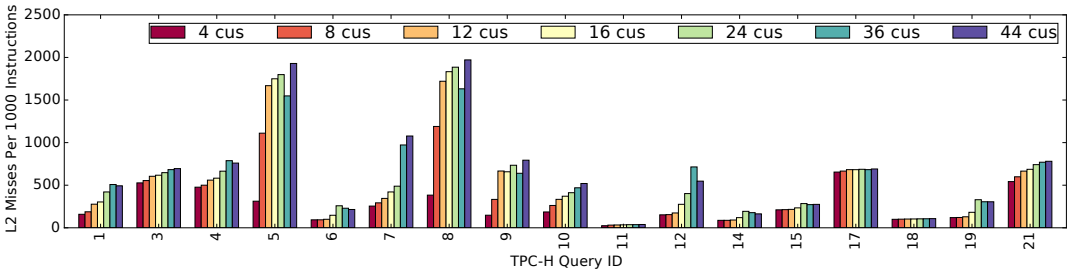


Figure 4.4. Shared last level (L2) cache misses per 1000 instructions (lower is typically better). For many queries, there are approximately 1 to 2 memory transactions per instruction.

performance counters), performance, energy, and energy delay product (energy * run time).

L1 Miss Metrics – We begin by analyzing private L1 data cache misses per 1000 instructions (a.k.a. L1 MPKI). Figure 4.2 presents the results. In general, queries that are

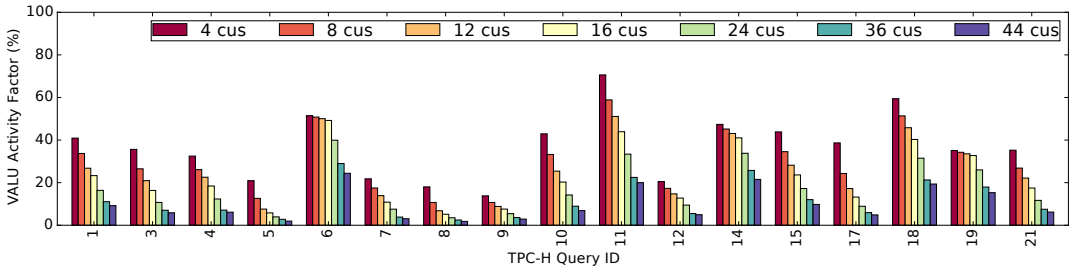


Figure 4.5. Vector ALU Activity Factor (higher is better). This metric measures the average number of lanes per enabled CU that perform useful work per clock cycle.

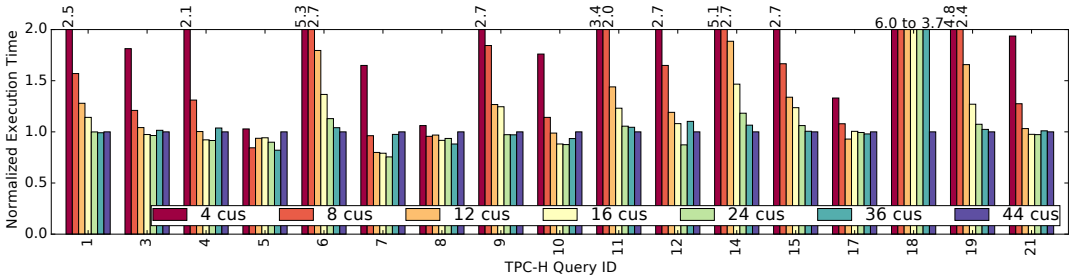


Figure 4.6. Performance normalized to 44 active CUs (lower is better). Most queries see little performance benefit from having all CUs active.

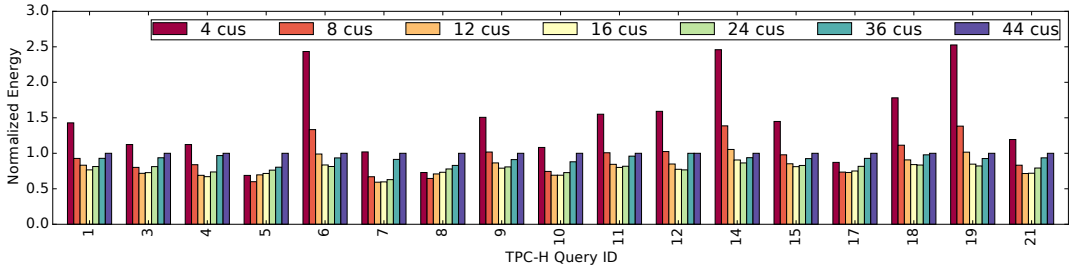


Figure 4.7. Energy normalized to 44 active CUs (lower is better). Somewhere between 8 to 24 active CUs yields the lowest energy for the surveyed queries.

dominated by bitmap joins (queries 5, 8, and 9) have the highest L1 MPKI. Since we fix the number of threads when disabling CUs, some if not all of the remaining enabled CUs receive more work groups to process. For many kernels, Ocelot intentionally launches a single work group per CU to lessen the severity of the memory subsystem contention. Due to this consolidation, many CUs have more active threads in flight than when all

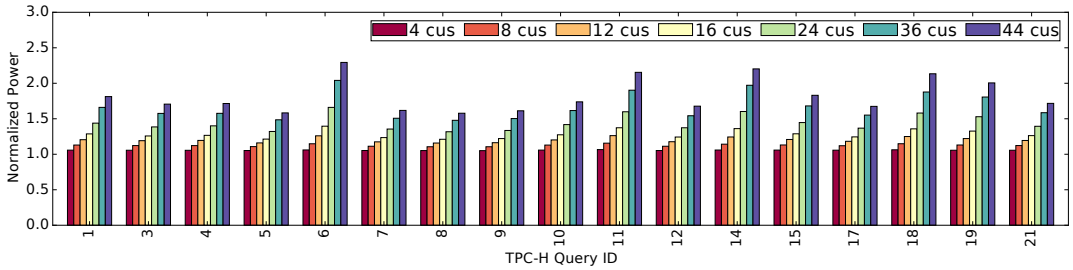


Figure 4.8. Mean Power. A low VALU activity factor leads to low mean power utilization that is dominated by GDDR5 and idle power. Results are normalized to the sum of *Idle Power Other* and *GDDR Power*.

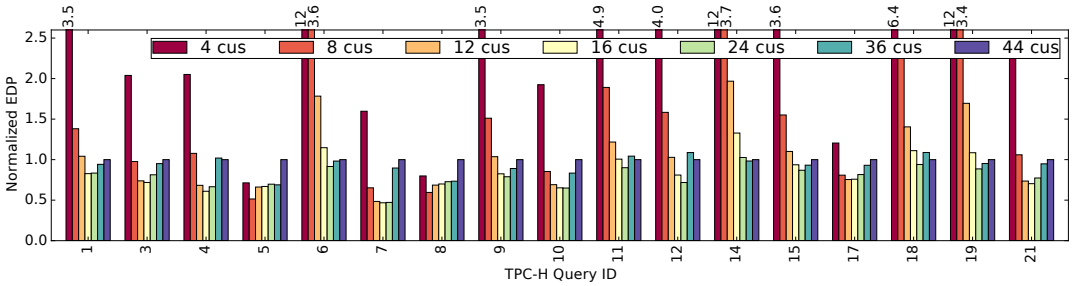


Figure 4.9. Energy delay product normalized to 44 active CUs (lower is better).

CUs are enabled. While having more threads in-flight increases contention on the L1 and the L1 MPKI, as each additional wavefront generates additional L1 cache pressure, we find that most of the workloads showed no reduction in performance. For these workloads, the delta in L1 misses due to consolidation on fewer CUs is a poor indicator of performance and performance scaling. Even though most queries see an increase in L1 data cache misses as we disable CUs, this increase is not reflected in an increase in application execution time. Similarly, having little to no change in L1 data cache misses as we vary the number of CUs does not appear to be a strong indicator of performance either. Instead the deltas in the L2 miss ratio and L2 MPKI yield superior insights.

L2 Miss Metrics – High L1 MPKI yield high L2 accesses with the potential to thrash the L2 to due to resource contention. For these workloads, much of the time, high L1 MPKI values also herald high L2 miss ratios. However, that is not always the case. It

is possible for the collective set of in-flight wavefronts to have a hot working set that is too big to cache in the combined L1 data cache capacity across all CUs (44 * 16 KB = 704 KB) but which mostly fits in the 1.7 MB of combined L1 and L2 caches. Figure 4.3 shows these two contrasting trends. Queries 5 and 8, which have some of the highest L1 MPKI also exhibit high L2 miss ratios. By contrast, query 9 has the highest L1 MPKI, yet it incurs the second lowest L2 miss ratio. Its working set can be fit almost entirely in the L1 and L2 caches.

A high L2 miss ratio alone is not sufficient to make the cache the bottleneck. If the bulk of loads hit in the L1, the L2 is able to keep pace with demand, and the latency can be hidden through hardware multithreading, then performance is unlikely to acutely suffer. If we examine L2 misses per 1000 dynamic instructions (L2 MPKI) in Figure 4.4, we observe that most queries exhibit on the order of 0.5 to 2.0 L2 misses per dynamic instruction. These figures prove problematic because they are indicative of applications that crave more off-chip demand than the hardware can satisfy, which can lead to long stalls in the execution pipeline. Further, they suggest that loads are poorly coalesced (memory divergence). A load without any coalescing can trigger up to 64 different 64-byte cache lines to be brought into cache from memory. If the prevalence of these loads is frequent and temporal reuse is poor, then with high probability many loads are likely to suffer the latency penalty of retrieving data from memory. While the GPU is adept at hiding memory latency, this latency hiding is dependent on at least one wavefront per SIMD being able to run. If no such wavefront exists, then the latency is placed on the critical path. The challenge with highly memory divergent code is that the additional wavefronts' contribution to thrashing the cache and contending for off-chip bandwidth may outweigh the latency-hiding benefit that they provide. When examining Figures 4.3 and 4.4, it becomes clear that *most queries see a marked decrease in both the L2 miss ratio and MPKI metrics as more CUs are disabled*. We attribute the bulk of this

benefit to having fewer active CUs. With fewer CUs, there are also fewer threads actively contending for the L2 at any one time. This reduced contention increases the likelihood that a greater fraction of a thread's state remains in L2. As a result, threads benefit from accessing more of their data from the higher bandwidth, lower-latency L2 and less from the lower bandwidth, higher-latency GDDR5-based main memory.

VALU Activity Factor – To understand the relationship between memory subsystem contention and useful computation, we measure both the VALU activity factor (Figure 4.5) and execution time as we vary the number of active CUs from 4 to 44 (Figure 4.6). Ideal activity factors are close to 100%. Figure 4.5 shows that as pressure on the L2 and off-chip bandwidth is alleviated by disabling CUs, arithmetic logic units spend a greater ratio of the time in active computation. The low activity factor demonstrates that CUs frequently stall due to resource contention. Further, as L2 cache hit ratios improve due to reduced active CUs issuing cache and memory requests, stalls reduce, and the VALUs show increased activity.

Execution Time – Figure 4.6 presents the run time of each query normalized to the default configuration where all CUs are active. Due to the high contention for the memory subsystem, many queries perform best when a large minority or majority of the CUs are disabled. Of the queries, only those that are scan heavy (6, 14, and 19) fail to achieve performance benefit from disabling CPUs. The queries that see the most benefit are queries 5 and 7, which have some of the sharpest reductions in L2 cache misses (Figure 4.4). The optimum CUs per query varies and is integrally tied to its operator composition.

Power, Energy, and EDP – For most of the queries, there are large flat regions where varying CUs has little to no impact on performance (Figure 4.6). For many of the queries, these regions occur both when most of the CUs are disabled and near the region of peak performance. This trait allows for significant savings in energy without

adversely compromising on performance. In Figure 4.7, we see that most queries realize an approximately 20% to 40% reduction in energy by disabling CUs. A large part of those savings comes from reducing idle power, i.e., power that is consistently consumed regardless of whether a component is conducting active computation. In Figure 4.8, we present the mean power that running each query consumes when varying the number of enabled CUs. Plotted values are normalized to the sum of the GDDR5 memory's power and the idle power attributed to components other than the CUs. For these TPC-H workloads, the low VALU activity factor means that transistors spend much less time actively switching, and thus dynamic power is significantly reduced. Since total power is the sum of idle power and dynamic power, reducing dynamic power means that idle power constitutes a much greater percentage of the total power. Further, because much of that idle power comes from the CUs, power gating CUs can sharply reduce the total power.

With all 44 CUS enabled, CU idle power accounts for about 56% of the total idle power or approximately 23% to 30% of the total power for the surveyed TPC-H queries. For each group of 4 CUs that we disable and power gate, we reduce the total power by about 2% to 3%. Since most of the queries can operate at improved or near-peak performance with a large plurality of the CUs disabled, much of the energy savings come from this power gating. A representative TPC-H query that observes equivalent performance with 12 CUs and 44 CUs enabled, would see a 22% to 25% reduction in total energy. These figures are in line with the mean savings that our study achieves. Cases where energy savings exceed these values occur when performance also improves (e.g., queries 5 and 7).

We also examine EDP (Figure 4.9). Because EDP factors in both run time and energy, it is a good indicator of whether energy savings are fundamentally at odds with throughput. Since the EDP results are as good or better, this validates that per-CU power

gating is not at odds with performance for this decision support workload.

4.4 Discussion

In this section, we describe the implications of these results as well as important caveats. First and foremost, Ocelot is a research prototype that attempts to span multiple platforms. While it includes many of the optimizations of MonetDB, we observed that some of the GPU kernel implementations were simple and did not use the fastest known algorithm. For instance, the hash join does not use the state-of-the-art bucketized cuckoo hash table and employs a no-partitioning join rather than a radix-hash join [54]. We spotted other simplifications that led to memory and branch divergence (Section 4.1). Many of these simplifications are justified by the fact that PCIe bus transfer bandwidth and CPU-side marshalling code consume much of the execution time. However, once these optimizations are applied, it is likely that there will be a reduction in some of the energy savings and performance scaling should improve.

Despite that, these results are still valuable because they demonstrate areas for further optimization once new hardware obviates the PCIe bus bottleneck. Further, given that we modelled coarse-grain power gating, there is almost certainly a much greater opportunity for databases to specify power gating at the region of individual operators or routines. A GPU-centric research effort into this space is warranted.

4.5 Related Work

Much of the work on GPU databases has been inspired by two decades of research on in-memory columnar databases [54, 55, 143, 199, 220]. With the proliferation of inexpensive, dynamic random access memory in the 1990s, the bottleneck shifted from IO to memory [25, 54, 247].

Many works have looked at leveraging GPUs for query processing [38, 61, 108, 109, 111, 112, 128, 187, 244, 245]. Bress et al. provide an excellent survey [62]. Much of the early work was done by He et al. [108, 109] and modularized the problem into expressing query operators as a combination of one or more parallel processing primitives (e.g., scatter, gather, and sort). Further optimizations have followed including leveraging new hardware-software support for automated asynchronous migration of data between the CPU and GPU's memory.

Wu et al. provided a methodology to automated JIT compilation of SQL to GPU kernel code, with fusion of operators and their kernels to reduce memory traffic [244,245]. Other works examined additional trade-offs in systems where query co-processing occurs on a CPU with an integrated GPU [61, 111].

On the power management side, a number of works have attempted to make database systems and operating systems more power aware. Weiser et al. pioneered early work that used trace-based simulation to evaluate different scheduling algorithms when coupled with DVFS on a time-sharing system [239]. Follow-on work has aimed to address similar issues [97,180]. Recent work has argued for and demonstrated the success of incorporating DVFS into operating and runtime systems for power management of multicore systems [139,160,216,234]. Other techniques such as racing or computationally sprinting through work followed by entering a low-power state or napping have also been employed [171, 197].

In the databases space, Tsirogiannis et al. was one of the first works to characterize the performance and energy trade-offs within a database system. Due to the lack of energy proportionality in server hardware [42, 160], they find that optimizing for the performance of key database operators often leads to the best energy efficiency [230]. More recent work by Xu et al. presents a control system for DVFS of a PostgreSQL system [251] and energy-aware query planning [250].

Our study shows that memory subsystem contention presents challenges to scaling OLAP database workloads on the GPU and that reducing the number of enabled CUs, and thus the pressure on the shared L2, has the potential to improve performance or energy efficiency. Other works explore similar challenges related to memory contention and factor this in to determining the ideal parallelism to employ. In the GPU architecture community, a number of works in simulation demonstrate that small microarchitectural modifications to the hardware that (1) reduce the number of in-flight threads [131, 202], cap the number of loads that threads can issue per cycle [202], (2) prioritize the scheduling of some work groups and their memory requests over others [126,210], (3) switch between greedy and round-robin hardware wavefront schedulers subject to the scaling behavior of the application as a function of the work groups per CU [149], and (4) schedule work groups or wavefronts in a locality-conscious fashion [126, 202] can greatly improve performance by reducing contention for the L1 data caches, shared L2, and off-chip bandwidth to GDDR5.

Our study is distinct from these works in that it considers both performance and energy scaling of GPGPU databases on real hardware. Further, we believe this work to be the first to evaluate the implications of per-compute-unit power gating for database workloads and to show its potential for energy savings and performance improvement on real hardware.

4.6 Conclusions

We have presented a detailed evaluation of the performance of an in-memory columnar database on a GPU running TPC-H. Our results show that further research into memory-subsystem-aware database primitives is necessary to exploit the full potential of GPUs for query processing: current code saturates in performance for the bulk of the queries when only a minority of the CUs are enabled. We then proposed powergating the

disabled CUs. Our evaluation of power gating showed that query latency, energy, and energy-delay product can be reduced by as much as 24%, 42%, and 53%, respectively. The findings demonstrate that there is a ripe opportunity for databases to orchestrate power gating within accelerators to boost both energy efficiency and performance. Finer grain power gating, both at the time and hardware level, will almost certainly lead to further benefit.

4.7 Acknowledgements

We thank Gabe Loh for conducting an expedited review of this work and for his comments. In addition, we thank Onur Kayıran for his thoughts on prior work within the GPU microarchitecture community.

AMD, the AMD Arrow logo, AMD Radeon, and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple, Inc. used by permission by Khronos. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Chapter 4, in part is currently being prepared for submission for publication of the material. Breslow, Alexander D.; Zhang, Dong Ping, Greathouse, Joseph L., Jayasena, Nuwan, Tullsen, Dean M. The dissertation author was the primary investigator and author of this material.

Chapter 5

Optimizing Hash Tables for the Memory Hierarchy

Hash tables are fundamental data structures that are ubiquitous in high performance and big-data applications such as in-memory relational databases [54, 80, 133] and key-value stores [90, 96].[1] Typically these workloads are read-heavy [34, 199]: the hash table is built once and is seldom modified in comparison to total accesses. A hash table that is particularly suited to this behavior is a bucketized cuckoo hash table (BCHT), a type of open-addressed hash table.[2] BCHTs group their cells into buckets: associative blocks of two to eight slots, with each slot capable of storing a single element.

When inserting an element, BCHTs typically select between one of two independent hash functions, each of which maps the key-value pair, call it KV, to a different *candidate bucket*. If one candidate has a free slot, KV is inserted. In the case where neither has spare slots, BCHTs resort to *cuckoo hashing*, a technique that resolves collisions by evicting and rehashing elements when too many elements vie for the same bucket. In this case, to make room for KV, the algorithm selects an element, call it KV', from one of KV's candidate buckets, and replaces it with KV. KV' is then subsequently rehashed to

[1] Figure 4.1 in Chapter 4 demonstrates the importance of optimizing hash tables for OLAP database workloads. The figure shows that Ocelot spends the bulk of its kernel execution time for TPC-H queries 3, 4, 11, 15, 17, 18, and 21 in routines that operate on hash tables. These routines primarily consist of hash joins and group-and-aggregate operations that employ grouping hash tables.

its alternate candidate using the remaining hash function. If the alternate bucket for KV' is full, KV' evicts yet another element and the process repeats until the final displaced element is relocated to a free slot.

Although these relocations may appear to incur large performance overheads, prior work demonstrates that most elements are inserted without displacing others and, accordingly, that BCHTs trade only a modest increase in average insertion and deletion time in exchange for high-throughput lookups and load factors that often exceed 95% with greater than 99% probability [87], a vast improvement over the majority of other techniques [190, 203].

BCHTs, due to this space efficiency and high throughput, have enabled recent performance breakthroughs in relational databases and key-value stores on server processors [90, 190, 203] as well as on accelerators such as GPUs [255], the Xeon Phi [73, 190], and the Cell processor [117, 203]. However, although BCHTs are higher-performing than other open addressing schemes [190,203], we find that as the performance of modern computing systems becomes increasingly constrained by memory bandwidth [25,53,169,247], they too suffer from a number of inefficiencies that originate from how data is fetched when satisfying queries.

Carefully coordinating table accesses is integral to throughput in hash tables. Because of the inherent randomness of hashing, accesses to hash tables often exhibit poor temporal and spatial locality, a property that causes hardware caches to become increasingly less effective as tables scale in size. For large tables, cache lines containing previously accessed hash table buckets are frequently evicted due to capacity misses before they are touched again, degrading performance and causing applications to become memory-bandwidth-bound once the table's working set cannot be cached on-chip. Given these concerns, techniques that reduce accesses to additional cache lines in the table prove

[2]Under open addressing, an element may be placed in more than one location in the table. Collisions are resolved by relocating elements within the table rather than spilling to table-external storage.

invaluable when optimizing hash table performance and motivate the need to identify and address the data movement inefficiencies that are prevalent in BCHTs.

Consider a BCHT that uses two independent hash functions to map each element to one of two candidate buckets. To load balance buckets and attain high load factors, recent work on BCHT-based key-value stores inserts each element into the candidate bucket with the least load [90, 255], which means that we expect half of the elements to be hashed by each function. Consequently, on positive lookups, where the queried key is in the table, 1.5 buckets are expected to be examined. Half of the items can be retrieved by examining a single bucket, and the other half require accessing both. For negative lookups, where the queried key is not in the table, 2 lookups are necessary. Given that the minimum number of buckets that might need to be searched (for both positive and negative lookups) is 1, this leaves a very significant opportunity for improvement.

To this end, this chapter presents Horton tables,[3] a carefully retrofitted bucketized cuckoo hash table, which trades a small amount of space for achieving positive and negative lookups that touch close to 1 bucket apiece. Our scheme introduces **remap entries**, small and succinct in-bucket records that enable (1) the tracking of past hashing decisions, (2) the use of many hash functions for little to no cost, and (3) most lookups, negative and positive alike, to be satisfied with a single bucket access and at most 2. Instead of giving equal weight to each hash function, which leads to frequent fetching of unnecessary buckets, we employ a single **primary hash function** that is used for the vast majority of elements in the table. By inducing such heavy skew, most lookups can be satisfied by accessing only a single bucket. To permit this biasing, we use several secondary hash functions (7 in our evaluations) to rehash elements to alternate buckets when their preferred bucket lacks sufficient capacity to directly store them. Rather than forgetting our choice of secondary hash function for remapped elements, we convert one of the slots

[3] Named for elephants' remarkable recall powers [81].

in each bucket that overflows into a **remap entry array** that encodes which hash function was used to remap each of the overflow items. It is this ability to track all remapped items at low cost, both in terms of time and space, that permits the optimizations that give Horton tables their performance edge over the prior state-of-the-art.

To achieve this low cost, instead of storing an explicit tag or fingerprint (a succinct hash representation of the remapped object) as is done in cuckoo filters [89] and other work [45, 56], we instead employ implicit tags, where the index into the remap entry array is a tag computed by a hash function H_{tag} on the key. This space optimization permits all buckets to use at most 64 bits of remap entries in our implementation while recording all remappings, even for high load factors and tables with billions of elements. As a further optimization, we only convert the last slot of each bucket into a remap entry array when necessary. For buckets that do not overflow, they remain as standard buckets with full capacity, which permits load factors that exceed 90 and 95 percent for 4- and 8-slot buckets, respectively.

Our main contributions are as follows:

- We develop and evaluate Horton tables and demonstrate speed improvements of 17 to 89% on graphics processors (GPUs). Although our algorithm is not specific to GPUs, GPUs represent the most efficient platform for the current state of the art, and thus it is important to demonstrate the effectiveness of Horton tables on the same platform.

- We present algorithms for insertions, deletions, and lookups on Horton tables.

- We conduct a detailed analysis of Horton tables by deriving and empirically validating models for their expected data movement and storage overheads.

- We investigate additional optimizations for insertions and deletions that further reduce data movement when using multiple hash functions, reclaiming remap

entries once their remapped elements are deleted, even when they are shared by two or more table elements.

This chapter is organized as follows: In Section 5.1 we elaborate on the interplay between BCHTs and single instruction multiple data (SIMD) processors, in Section 5.2 we describe BCHTs, in Section 5.3 we provide a high-level overview of Horton tables, in Section 5.4 we describe the lookup, insertion, and deletion algorithms for Horton tables, and then in Section 5.5 we present our models for Horton tables that include the cost of insertions, deletions, and remap entry storage. Section 5.6 covers our experimental methodology, and Section 5.7 contains our performance and model validation results. Related work is described in Section 5.8 and Section 5.9 concludes.

5.1 The Role of SIMD

In key places in this chapter, we make references to SIMD and GPU architectures. Although not necessary for understanding our innovations, these references are present due to SIMD's importance for high-throughput implementations of in-memory hash tables and BCHTs in particular.

Recent work in high-performance databases that leverages BCHTs has shown that SIMD implementations of BCHTs, as well as larger data processing primitives, are often more than $3\times$ faster than the highest performing implementations that use scalar instructions alone [40, 150, 190]. These SIMD implementations enable billions of lookups per second to be satisfied on a single server [190, 255], an unimaginable feat only a few years ago. At the same time, SIMD implementations of BCHTs are faster than hash tables that use other open addressing methods [190, 203]. Such implementations are growing in importance because both CPUs and GPUs require writing SIMD implementations to maximize performance-per-watt and reduce total cost of ownership.

For these reasons, we focus on a SIMD implementation of BCHTs as a starting point and endeavor to show that all further optimizations provided by Horton tables not only have theoretical models that justify their performance edge (Section 5.5) but also that practical SIMD implementations deliver the performance benefits that the theory projects (Section 5.7).[4]

5.2 Background on BCHTs

In this section, we describe in detail BCHTs and the associated performance considerations that arise out of their interaction with the memory hierarchy of today's systems.

To begin, we illustrate two common scenarios that are triggered by the insertion of two different key-value pairs KV_1 and KV_2 into the hash table, as shown in Figure 5.1. Numbered rows correspond to buckets, and groups of four cells within a row to slots. In this example, H_1 and H_2 correspond to the two independent hash functions that are used to hash each item to two candidate buckets (0 and 2 for KV_1, 3 and 6 for KV_2). Both H_1 and H_2 are a viable choice for KV_1 because both buckets 0 and 2 have free slots. Deciding which to insert into is at the discretion of the algorithm (see Section 5.2.3 for more details).

For KV_2, both H_1 and H_2 hash it to buckets that are already full, which is resolved by evicting one of the elements (in this case u), and relocating it and other conflicting elements in succession using a different hash function until a free slot is found.[5] So e moves to the empty position in bucket 5, m to e's old position, u to m's old position, and KV_2 to u's old position. Li, et al. demonstrated that an efficient way to perform these

[4]For a primer on SIMD and GPGPU architectures, we recommend these excellent references: H&P (Ch. 4) [113] and Keckler et al. [132].

[5]So in this example, elements on the chain that were originally hashed with H_1 would be rehashed using H_2 and vice versa.

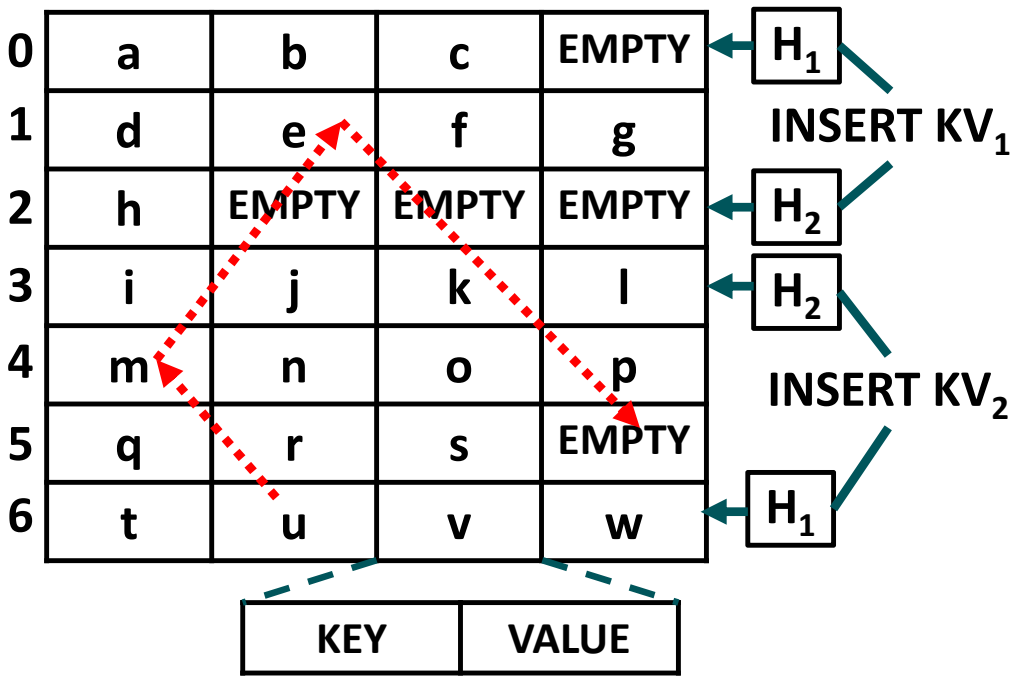


Figure 5.1. Inserting items $\mathbf{KV_1}$ and $\mathbf{KV_2}$ into a BCHT

displacements is to first conduct a breadth-first search starting from the candidate buckets and then begin moving elements only once a path to a free slot is discovered [153].

5.2.1 State-of-Practice Implementation

A number of important parameters affect the performance of BCHTs. In particular, the number of hash functions (f) and the number of slots per bucket (S) impact the achievable load factor (i.e., how full a table can be filled) as well as the expected lookup time. A hash table with more slots per bucket can more readily accommodate collisions without requiring a rehashing mechanism (such as cuckoo hashing) and can increase the table's load factor. Most implementations use four [90, 190, 203] or eight [255] slots per bucket, which typically leads to one to two buckets per hardware cache line. Using more slots comes at the cost of more key comparisons on lookups, since the requested element could be in any of the slots.

Increasing f, the number of hash functions, allows a key-value pair to be mapped

to more buckets, as each hash function maps the item to one of f different buckets. This improved flexibility when placing an item permits the hash table to achieve a higher load factor. However, on lookups, more buckets need to be searched because the element could be in more locations. In practice, $f = 2$ is used most often because it permits sufficient flexibility in where keys are mapped without suffering from having to search too many buckets [90, 173, 201].

BCHTs are primarily designed for fast lookups. The get operation on any key requires examining the contents of at most f buckets. Because buckets have a fixed width, the lookup operation on a bucket can be unrolled and efficiently vectorized. These traits allow efficient SIMD implementations of BCHTs that achieve lookup rates superior to linear probing and double-hashing-based schemes on past and present server architectures [190, 203] and accelerators such as Intel's Xeon Phi [190].

5.2.2 Memory Traffic on Lookups

Like prior work, we divide lookups into two categories: (1) positive, where the lookup succeeds because the key is found in the hash table, and (2) negative where the lookup fails because the key is not in the table.

Prior work diverges on the precise method of accessing the hash table during lookups. The first method, which we term **latency-optimized**, always accesses all buckets where an item may be found [153, 203]. Another technique, which we call **bandwidth-optimized**, avoids fetching additional buckets where feasible [90, 255].

Given f independent hash functions where each of them maps each item to one of f candidate buckets, the latency-optimized approach always touches f buckets while the bandwidth-optimized one touches, on average, $(f+1)/2$ buckets on positive lookups and f buckets on negative lookups. For our work, we compare against the **bandwidth-optimized** approach, as it moves less data on average. Reducing such data movement

is a greater performance concern on throughput-oriented architectures such as GPUs, since memory latency is often very effectively hidden on these devices [92]. Thus, we compare against the more bandwidth-oriented variant of BCHT, which searches 1.5 buckets instead of 2 (or more, if there are more hash functions) for positive lookups.

5.2.3 Insertion Policy and Lookups

Unlike the latency-optimized scheme, the bandwidth-optimized algorithm searches the buckets in some defined order. If an item is found before searching the last of the f candidate buckets, then we can reduce the lookup's data movement cost by skipping the search of the remaining candidate buckets. Thus if f is 2, and we call the first hash function H_1 and the second H_2, then the average lookup cost across all inserted keys is `1 * (fraction of keys that use` H_1`) + 2 * (fraction of keys that use` H_2`)`. Therefore, the insertion algorithm's policy on when to use H_1 or H_2 affects the lookup cost.

Given that hash tables almost always exhibit poor temporal and spatial locality, hash tables with working sets that are too large to fit in caches are bandwidth-bound and are quite sensitive to the comparatively limited off-chip bandwidth. In the ideal case, we therefore want to touch as few buckets as possible. If we can strongly favor using H_1 over H_2 during insertions, we can reduce the percentage of buckets that are fetched that do not contain the queried key, which reduces per-lookup bandwidth requirements as well as cache pollution, both of which improve lookup throughput.

Existing high-throughput, bandwidth-optimized BCHT implementations [90,255] attempt to load-balance buckets on insertion by examining all buckets the key can map to and placing elements into the buckets with the most free slots. As an example, in Figure 5.1, KV_1 would be placed in the bucket hashed to by H_2. The intuition behind this load balancing is that it both reduces the occurrence of cuckoo rehashing, which

is commonly implemented with comparatively expensive atomic swap operations, and increases the anticipated load factor. Given this policy, H_1 and H_2 are both used with equal probability, which means that 1.5 buckets are searched on average for positive lookups. We refer to this approach as the **load-balancing baseline**.

An alternative approach is to insert items into the first candidate bucket that can house them. This technique reduces the positive lookup costs, since it favors the hash functions that are searched earlier. We refer to this as the **first-fit** heuristic. As an example, in Figure 5.1, KV_1 would be placed in the final slot of the top bucket of the table even though bucket 2 has more free slots. This policy means that items can be located with fewer memory accesses, on average, by avoiding fetching candidate buckets that follow a successful match. When the table is not particularly full, most elements can be inserted and retrieved by accessing a single table cache line.

Although prior work mentions this approach [87, 203], they do not evaluate its performance impact on lookups. Ross demonstrated its ability to reduce the cost of inserts but does not present data on its effect on lookups, instead opting to compare his latency-optimized lookup algorithm that always fetches f buckets to other open addressing methods and chaining [203]. Erlingsson et al. use the first-fit heuristic, but their results focus on the number of buckets accessed on insertions and feasible load factors for differing values of f and S (number of slots per bucket) and not the heuristic's impact on lookup performance [87]. For the sake of completeness, we evaluate both the load-balancing and first-fit heuristics in Section 5.7.

One consequence of using first-fit is that, because it less evenly balances the load across buckets, once the table approaches capacity, a few outliers repeatedly hash to buckets that are already full, necessitating long, cuckoo displacement chains when only 2 hash functions are used. Whereas we were able to implement the insertion routine for the load-balancing baseline and attain high load factors by relocating at most one

or two elements, the first-fit heuristic prompted us to implement a serial version of the BFS approach described by Li et al. [153] because finding long displacement chains becomes necessary for filling the table to a comparable level. One solution to reducing these long chains is to use more hash functions. However, for BCHTs, this increases both the average- and worst-case lookup costs because each item can now appear in more distinct buckets. In the sections that follow, we demonstrate that these tradeoffs can be effectively circumvented with our technique and that there are additional benefits such as fast negative lookups.

5.3 Horton Tables

Horton tables are an extension of bucketized cuckoo hash tables that largely resolve the data movement issues of their predecessor when accessing buckets during lookups. They use two types of buckets (Figure 5.2): one is an unmodified BCHT bucket (**Type A**) and the other bucket flavor (**Type B**) contains additional in-bucket metadata to track elements that primarily hash to the bucket but have to be remapped due to insufficient capacity. All buckets begin as Type A and transition to Type B once they overflow, enabling the aforementioned tracking of displaced elements. This ability to track all remapped items at low cost, both in terms of time and space, permits the optimizations that give Horton tables their performance edge over the prior state of the art.

Horton tables use $H_{primary}$, the **primary hash function**, to hash the vast majority of elements so that most lookups only require one bucket access. When inserting an item $KV = (K,V)$, it is only when the bucket at index $H_{primary}(K)$ cannot directly store KV that the item uses one of several **secondary hash functions** to remap the item. We term the bucket at index $H_{primary}(K)$ the **primary bucket** and buckets referenced by secondary hash functions **secondary buckets**. Additionally, **primary items** are key-value pairs that

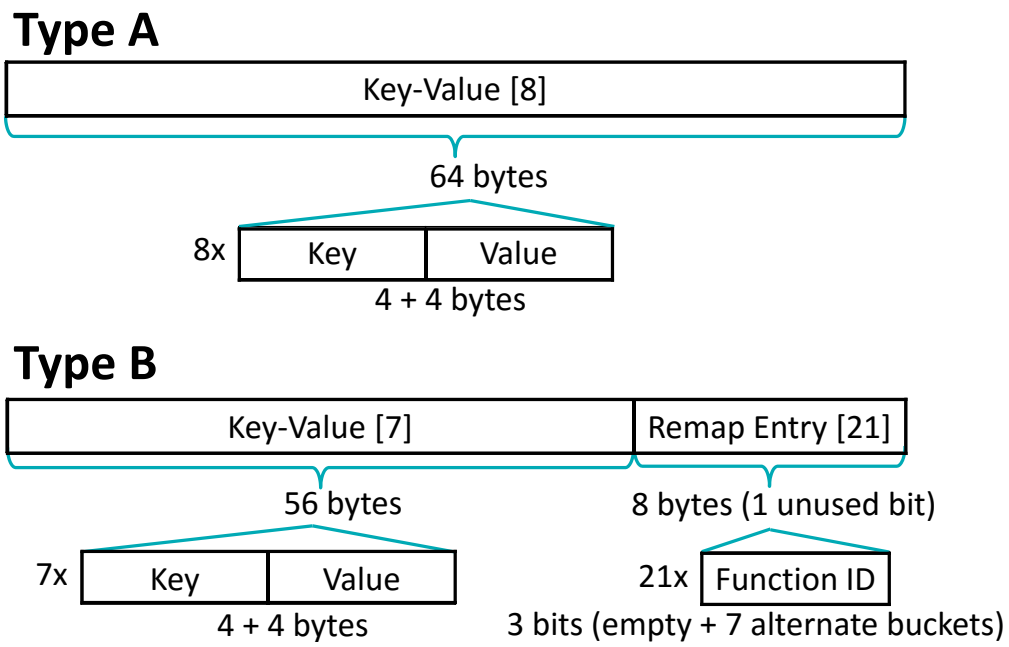


Figure 5.2. Horton tables use 2 bucket variants: Type A (an unmodified BCHT bucket) and Type B (converts final slot into remap entries)

are directly housed in the bucket referenced by the primary hash function $H_{primary}$, and **secondary items** are those that have been remapped. There is no correlation between the bucket's type and its primacy; Type A and B buckets can simultaneously house both primary and secondary elements.

Type B buckets convert the final slot of Type A buckets into a **remap entry array**, a vector of k-bit[6] elements known as **remap entries** that encode the secondary hash function ID used to rehash items that cannot be accommodated in their primary bucket. Remap entries can take on one of 2^k different values, 0 for encoding an unused remap entry, and 1 to $2^k - 1$ for encoding which of the secondary hash functions R_1 to R_{2^k-1} was used to remap the items. To determine the index at which a remapped element's remap entry is stored, a tag hash function known as H_{tag} is computed on the element's key which maps to a spot in the remap entry array.

[6] k could range from 1 to the width of a key-value pair in bits, but we have found $k = 3$ to be a good design point.

Remap entries are a crucial innovation of Horton tables, as they permit all secondary items to be tracked so that at most one primary and sometimes one secondary hash function need to be evaluated during table lookups regardless of whether (1) the lookup is positive or negative and (2) how many hash functions are used to rehash secondary items. At the same time, their storage is compactly allocated directly within the hash table bucket that overflows, boosting the locality of their accesses while still permitting high table load factors.

With Horton tables, most lookups only require touching a single bucket and a small minority touch two. At the same time remap entries typically use at most several percent of the table's capacity, leaving sufficient free space for Horton tables to achieve comparable load factors to BCHTs.

Figure 5.2 shows the Type A and Type B bucket designs given 4-byte keys and values and 8-slot buckets. The bucket type is encoded in one bit of each bucket. For Type A buckets, this costs us a bit from just one of the value fields (now 31 bits). For Type B buckets, we encode 21 3-bit remap entries into a 64-bit slot, so we have a bit to spare already. If we have a value that requires all 32 bits in the last slot of a Type A bucket, we can move it to another slot in this bucket or remap to another bucket.

Because Type B buckets can house fewer elements than Type A buckets, Type A buckets are used as much as possible. It is only when a bucket has insufficient capacity to house all primary items that hash to it that it is converted into a Type B bucket, a process known as **promotion**. To guarantee that elements can be found quickly, whenever possible we enforce that primary elements are not displaced by secondary items. This policy ensures both that more buckets remain Type A buckets and that more items are primary elements.

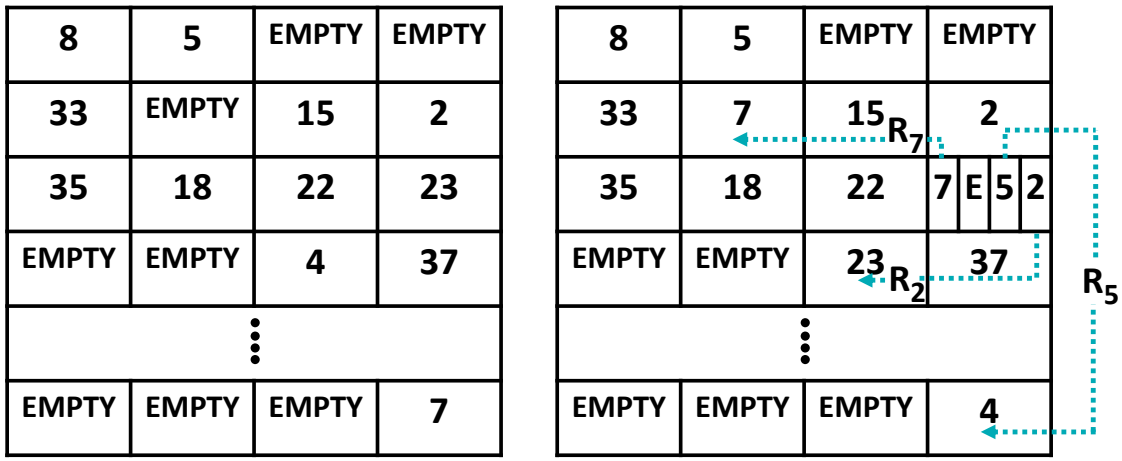


Figure 5.3. Comparison of a bucketized cuckoo hash table (L) and a Horton table (R). E = empty remap entry.

5.3.1 A Comparison with BCHTs

Figure 5.3 shows a high-level comparison of Horton tables with an $f = 2$, traditional BCHT that stores the same data. Buckets correspond to rows and slots to individual cells within each row. In the Horton table (right), each item maps to its primary bucket by default. Bucket 2 (zero indexing) has been **promoted** from Type A to Type B because its 4 slots are insufficient to directly house the 6 key-value pairs that $H_{primary}$ has hashed there: 35, 18, 22, 7, 23, and 4. Because there is insufficient space to house 7, 23, and 4 directly in Bucket 2, they are remapped with hash functions R_7, R_2, and R_5, respectively, and the function IDs are stored directly in the remap entry array at the indices referenced by $H_{tag}(7) = 0$, $H_{tag}(23) = 3$, and $H_{tag}(3) = 2$. If we contrast the Horton table and the associated cuckoo hash table, we find that, of the shown buckets, the Horton table has a lookup cost of 1 for elements 8, 5, 33, 15, 2, 35, 18, 22, and 37 and a lookup cost of 2 for 7, 23, and 4, which averages out to 1.25. By contrast the bucketized cuckoo hash table has an expected lookup cost of 1.5 [90, 255] or 2.0 [153, 203], depending on the implementation.

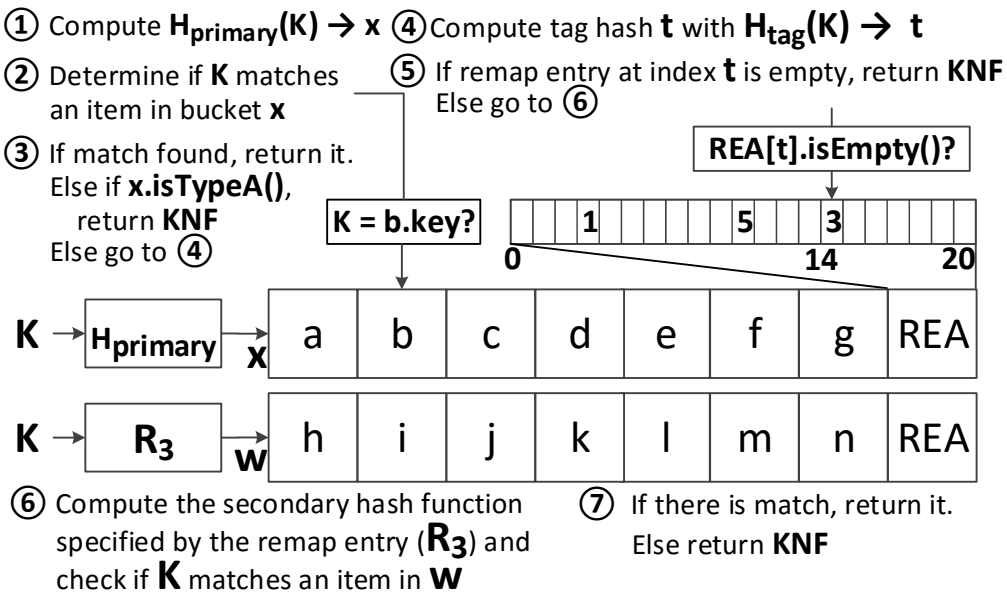


Figure 5.4. Horton table lookups. KNF and REA are abbreviations for key not found and remap entry array.

5.4 Algorithms

In this section, we describe the algorithms that we use to look up, insert, and delete elements in Horton tables. We begin each subsection by detailing the functional components of the algorithms and then, where relevant, briefly outline how each algorithm can be efficiently implemented using SIMD instructions.

5.4.1 Lookup Operation

Our lookup algorithm (Figure 5.4) works as follows. ① Given a key K, we first compute $H_{primary}(K)$, which gives us the index of K's primary bucket. ② We next examine the first $S - isTypeB()$ slots of the bucket, where $isTypeB()$ returns 1 if Bucket x is Type B and 0 if it is Type A by checking the bucket's most significant bit. ③ If the key is found, we return the value. In the case that the key is not found and the bucket is Type A, then the element cannot appear in any other bucket, and so we can declare the key not found.

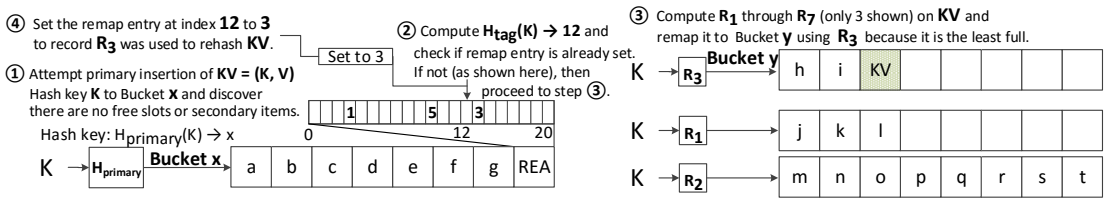


Figure 5.5. Common execution path for secondary inserts. REA is an abbreviation of remap entry array.

④ If, however, the bucket is Type B, then we must examine the remap entry array when the item is not found in the first $S-1$ slots of K's primary bucket (Bucket x). We first compute $H_{tag}(K)$ to determine the index into the remap entry array (shown as $t = 14$ in the figure). ⑤ If the value at that slot is 0, which signifies empty, then we declare the key not found because the key cannot appear in any other bucket. However, if the remap entry is set, then ⑥ we evaluate the secondary function R_i specified by the remap entry (R_3 in Figure 5.4) on a combination of the primary bucket and remap entry index (see Section 5.4.3) to get the index of the secondary bucket (Bucket w). We then compare K to the first $S-isTypeB()$ elements of w. ⑦ If we get a match, then we return it. If we do not get a match, then because an element is never found outside of its primary bucket or the secondary bucket specified by its remap entry, then we declare the key not found. It cannot be anywhere else.

5.4.2 SIMD Implementation of Lookups

Our approach leverages bulk processing of lookups and takes a large array of keys as input and writes out a corresponding array of retrieved values as output. We implement a modified version of Zhang et al.'s lookup algorithm [255], which virtualizes the SIMD unit into groups of S lanes (akin to simple cores that coexecute the same instruction stream) and assigns each group a different bucket to process. When an element is found in the first $S-isTypeB()$ slots, we write the value out to an in-cache buffer. For the minority

of lookups where more processing is necessary, e.g. computing the tag hash, indexing into the remap entry array, computing the secondary hash function, and searching the secondary bucket, we maintain a series of additional in-cache buffers where we enqueue work corresponding to these less frequent execution paths. When there is a SIMD unit's worth of work in a buffer, we dequeue that work and process it. Once a cache line worth of contiguous values have been retrieved from the hash table, we write those values back to memory in a single memory transaction.[7]

5.4.3 Insertion Operation

The primary goal of the insertion algorithm is to practically guarantee that lookups remain as fast as possible as the table's load factor increases. To accomplish this, we enforce at all levels the following guidelines:

1. Secondary items never displace primary items.

2. Primary items may displace both primary and secondary items.

3. When inserting into a full Type A bucket, only convert it to Type B if a secondary item in it cannot be remapped to another bucket to free up a slot.[8]

These guidelines ensure that as many buckets as possible remain Type A, which is important because converting a bucket from Type A to Type B can have a cascading effect: both the evicted element from the converted slot and the element that caused the conversion may map to other buckets and force them to convert to Type B as well. Further, Type B buckets house fewer elements, so they decrease the maximum load factor of the table and increase the expected lookup cost.

[7] A simpler algorithm can be used when S is a multiple of the number of lanes, as all lanes within a SIMD unit process the same bucket.

[8] An item's primacy can be detected by evaluating $H_{primary}$ on its key. If the output matches the index where it is stored, then it is primary.

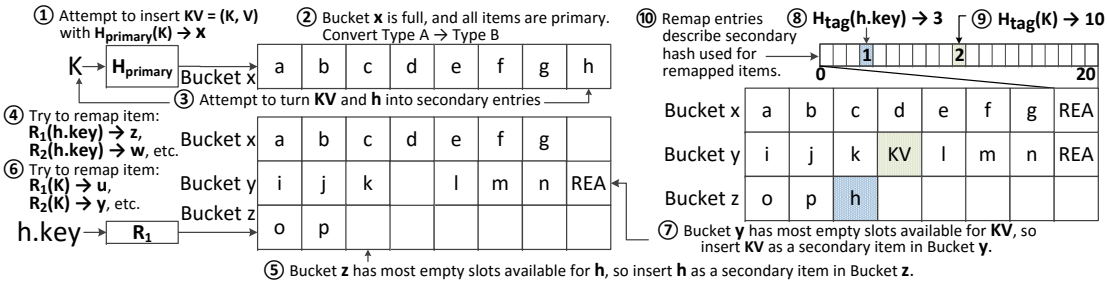


Figure 5.6. Steps for converting from Type A to Type B. REA is an abbreviation of remap entry array.

Primary Inserts

Given a key-value pair KV to insert into the table, if the primary bucket has a spare slot, then insertion can proceed by assigning that spare slot to KV. Spare slots can occur in Type A buckets as well as Type B buckets where a slot has been freed due to a deletion, assuming that Type B buckets do not atomically pull items back in from remap entries when slots become available. For the primary hash function, we use one of Jenkins' hash functions that operates on 32-bit inputs and produces 32-bit outputs [121]. The input to the function is the key, and we mod the output by the number of buckets to select a bucket to map the key to.

In the case where the bucket is full, we do not immediately attempt to insert KV into one of its secondary buckets but first search the bucket for secondary elements. If we find a secondary element KV' that can be remapped without displacing primary elements in other buckets, then we swap KV with KV' and rehash KV' to another bucket. Otherwise, we perform a secondary insert (see Sections 5.4.3, 5.4.3, and 5.4.3).

Secondary Inserts

We make a secondary insert when the item that we want to insert, $KV = (K,V)$, hashes to a bucket that is full and in which all items already stored in the bucket are primary. Most of the time, secondary inserts occur when an element's primary bucket

has already been converted to Type B (see Section 5.4.3 and Figure 5.6 for the steps for converting from Type A to B); Figure 5.5 shows the most common execution path for a secondary insert.

① We first determine that a primary insert is not possible. ② We then compute the tag hash function on the key. If the remap entry at index $H_{tag}(K)$ is not set, we proceed to step ③. Otherwise, we follow the remap entry collision management scheme presented in Section 5.4.3 and Figure 5.7. ③ At this point, we need to find a free cell in which to place KV. We check each candidate bucket referenced by the secondary hash functions R_1 through R_7, and we place the remapped element in the bucket with least load, Bucket y in Figure 5.5. Alternatively, we could have placed KV into the candidate bucket with the first free slot – we chose the load-balancing approach because it reduced the prevalence of expensive cuckoo chains for relocating elements. ④ Lastly, we update the remap entry to indicate which secondary hash function was used. In this example, R_3 was used and H_{tag} on K evaluated to 12, so we write 3 in the 12th slot of the remap entry array of x, KV's primary bucket.

Conversion from Type A to Type B

Figure 5.6 shows the series of steps involved for converting a bucket from Type A to Type B. ① – ② If there are no secondary elements that can be displaced in the primary bucket, then the bucket evicts one of the items (h) to make room for the remap entry array, ③ – ⑤ rehashes it to another bucket, and ⑥ – ⑦ then proceeds by rehashing the element that triggered the conversion. As in the algorithm in Section 5.4.3, we attempt to relocate both items to their least loaded candidate buckets. ⑧ – ⑩ Once moved, the numerical identifier of each secondary hash function that remapped each of the two items (KV and h) is stored in the remap entry array of the primary bucket at the index described by H_{tag} of each key.

Remap Entry Collision Management

A major challenge of the remap entry approach is when two or more items that require remapping map to the same remap entry. Such collisions can be accommodated if all items that share a remap entry are made to use the same secondary hash function. However, if the shared secondary hash function takes the key as input, it will normally map each of the conflicting items to a different bucket. While this property poses no great challenge for lookups or insertions, it makes deletions of remap entries challenging because without recording that a remap entry is shared, we would not know whether another item is still referenced by it. Rather than associating a counter with each remap entry to count collisions as is done in counting Bloom filters [56, 91], we instead modify the secondary hash function so that items that share a remap entry map to the same secondary bucket. Since they share the same secondary bucket, we can check if it is safe to delete a remap entry on deletion of an element KV that the entry references by computing the primary hash function on each element in KV's secondary bucket. If none of the computed hashes reference KV's primary bucket for any of the elements that share KV's secondary bucket, then the remap entry can be deleted.

To guarantee that items that share remap entries hash to the same secondary bucket, we hash on a combination of the primary bucket index and the implicit tag as computed by $H_{tag}(key)$. Since this tuple uniquely identifies the location of each remap entry, we can create a one-to-one function from tuples to unique secondary hash function inputs, shown in Equation 5.1, where i is a number that uniquely identifies each secondary hash function and which ranges from 1 to $2^k - 1$ for k-bit remap entries (e.g. R_3 is the third secondary function out of 7 when k is 3), H_{L1} and H_{L2} are hash functions, k_{sec} is the secondary key derived from the tuple, and n is the number of remap entries per remap

entry array. The uniqueness of these tuples as inputs is important for reducing collisions.

$$\begin{aligned} R_i(k_{sec}) &= (H_{L1}(k_{sec}) + H_{L2}(k_{sec}, i)) \ \% \ Total\ Buckets \\ where\ k_{sec}(bucket\ index, tag) &= bucket\ index * n + tag \end{aligned} \tag{5.1}$$

By modifying the characteristics of H_{L1} and H_{L2}, we are able to emulate different hashing schemes. We employ modified double hashing by using Jenkins' hash [121] for H_{L1} and Equation 5.2 for H_{L2} where KT is a table of 8 prime numbers. We found this approach preferable because it makes it inexpensive to compute all of the secondary hash functions, reduces clustering compared to implementing H_{L2} as linear probing from H_{L1}, and, as Mitzenmacher proved, there is no penalty to bucket load distribution versus fully random hashing [174].

$$H_{L2}(k_{sec}, i) = KT[k_{sec}\%8] * i \tag{5.2}$$

Sometimes the secondary bucket cannot accommodate additional elements that share the remap entry array. If so, we swap the item that we want to insert with another element from its primary bucket that can be rehashed. Because both elements in the swap are primary elements, this swap does not adversely affect the lookup rate.

Figure 5.7 presents a visual depiction of a remap entry collision during insertion that is resolved by having the new item map to the same bucket as the other items referenced by the remap entry. It continues from the example in Figure 5.6 and follows with inserting a new item KV'. ① When inserting KV', we first check for a free slot or an element that can be evicted from Bucket x because it is a secondary item when in x. ② If no such item exists, then we begin the process of remapping KV' to another bucket. ③ We first check the remap entry, and if it is set, ④ we proceed to the bucket it references, z in Figure 5.7. ⑤ We check for a free slot or a secondary item in z that can be displaced. If it is the former, we immediately insert KV'. Otherwise, we recursively

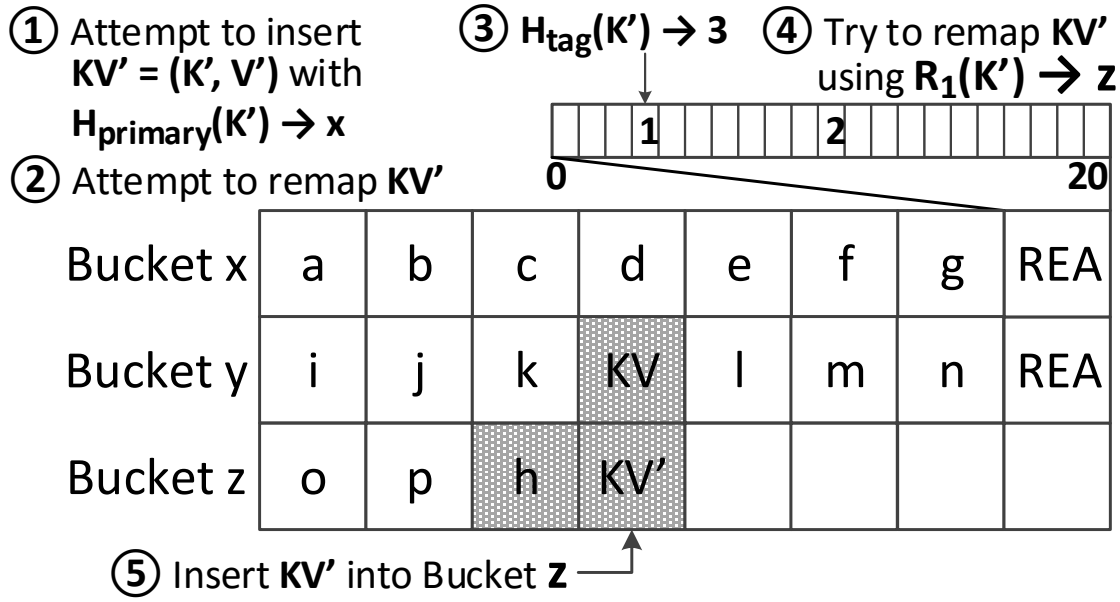


Figure 5.7. Resolution of a remap entry collision

evict elements until a free slot is found within a certain search tree height. Most of the time, this method works, but when it does not, we resort to swapping KV' with another element in its primary bucket that can be recursively remapped to a secondary bucket.

5.4.4 Deletion Operation

Deletions proceed by first calculating $H_{primary}$ on the key. If an item is found in the primary bucket with that key, that item is deleted. However, if it is not found in the primary bucket, and the bucket is Type B, then we check to see whether the remap entry is set. If it is, then we calculate the secondary bucket index and examine the secondary bucket. If an item with a matching key is found, then we delete that item. To determine whether we can delete the remap entry, we check to see if additional elements in the secondary bucket have the same primary bucket as the deleted element. If none do, we remove the remap entry.

Repatriation of Remapped Items

On deletions, slots that were previously occupied become free. In Type A buckets, there is no difference in terms of lookups regardless of how many slots are free. However, with Type B buckets, if a slot becomes free, that presents a performance opportunity to move a remapped item back into its primary bucket, reducing its lookup cost from 2 to 1 buckets. Similarly, if a Type B bucket has a combined total of fewer than $S+1$ items stored in its slots or remapped via its remap entries, it can be upgraded to a Type A bucket, which allows one more item to be stored and accessed with a single lookup in the hash table. Continual repatriation of items is necessary for workloads with many deletes to maximize lookup throughput and the achievable load factor. Determining when best to perform this repatriation, either via an eager or lazy heuristic, is future work.

5.5 Feasibility and Cost Analysis

In this section, we investigate the feasibility of using remap entries, the associated costs in terms of storage overhead, and the expected cost of both positive and negative hash table lookups.

5.5.1 Modeling Collisions

One of the most important considerations when constructing a Horton table is that each bucket should be able to track all items that initially hash to it using the primary hash function $H_{primary}$. In particular, given a hash table with B_T buckets and n inserted items, we want to be able to compute the expected number of buckets that have exactly x elements hash to them, for each value of x from 0 to n inclusive. By devising a model that captures this information, we can determine how many remap entries are necessary to facilitate the remapping and tracking of all secondary items that overflow their respective primary buckets.

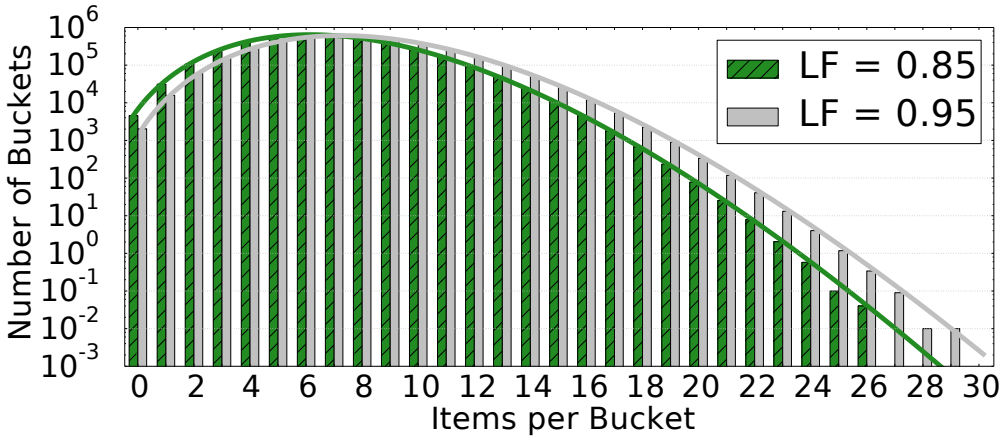


Figure 5.8. Histogram of the number of buckets to which $H_{primary}$ assigns differing amounts of load in elements for two load factors. Curves represent instantiations of Equation 5.3 and bars correspond to simulation.

If we assume that $H_{primary}$ is a permutation (i.e., it is invertible and the domain is the codomain), and that it maps elements to bins in a fashion that is largely indistinguishable from sampling a uniform random distribution, then given a selection of random keys and a given table size, we can precisely compute the expected number of buckets to which $H_{primary}$ maps exactly x key-value pairs by using a Poisson distribution based model [122]. The expected number of buckets with precisely x such elements, B_x, is given by Equation 5.3.

$$
\begin{aligned}
B_x(\lambda,x) &= Total\ Buckets\ *\ P(\lambda,x) \\
&\quad where\ P(\lambda,x) = \frac{e^{-\lambda}\lambda^x}{x!} \\
&\quad where\ \lambda = Load\ Factor\ *\ Slots\ Per\ Bucket \\
&\quad i.e.,\ \lambda = \frac{Elements\ Inserted}{Total\ Buckets}
\end{aligned}
\tag{5.3}
$$

The parameter λ is the mean of the distribution. Given a load factor, the average number of items that map to a bucket is the product of the load factor and the slots per

bucket. Figure 5.8 coplots the results of a bucketized hash table simulation with results predicted by the analytical model given a hash table with 2^{22} 8-slot buckets. In our simulation, we created n unique keys in the range $[0, 2^{32} - 1]$ using a 32-bit Mersenne Twister pseudorandom number generator [168] and maintained a histogram of counts of buckets with differing numbers of collisions. We found little to no variation in results with different commonly utilized hash functions (e.g., CityHash, SpookyHash, Lookup3, Wang's Hash, and FNV [99, 121, 188]). Therefore, we show only the results using one of Jenkins' functions that maps 32-bit keys to 32-bit values. Figure 5.8 shows a close correlation between the simulation results and Equation 5.3 for two load factors. Bars correspond to simulation results and curves to Equation 5.3. In each case, the model very closely tracks the simulation.

A high-level conclusion of the model is that with billions of keys and 8-slot buckets, there is a non-trivial probability that a very small subset of buckets will have on the order of 30 keys hash to them. This analysis informs our decision to use 21 remap entries per remap entry array and also the need to allow multiple key-value pairs to share each remap entry in order to reduce the number of remap entries that are necessary per bucket.

5.5.2 Modeling Remap Entry Storage Costs

In our hash table design, each promoted bucket trades one slot for a series of remap entries. To understand the total cost of remap entries, we need to calculate what percentage of buckets are Type A and Type B, respectively. For any hash table with S slots per bucket, Type A buckets have no additional storage cost, and so they do not factor into the storage overhead. Type B buckets on the other hand convert one of their S slots, i.e. $1/S$ of their usable storage, into a series of remap entries. Thus the expected space used by remap entries O_{re}, on a scale of 0 (no remap entries) to 1 (entire table is

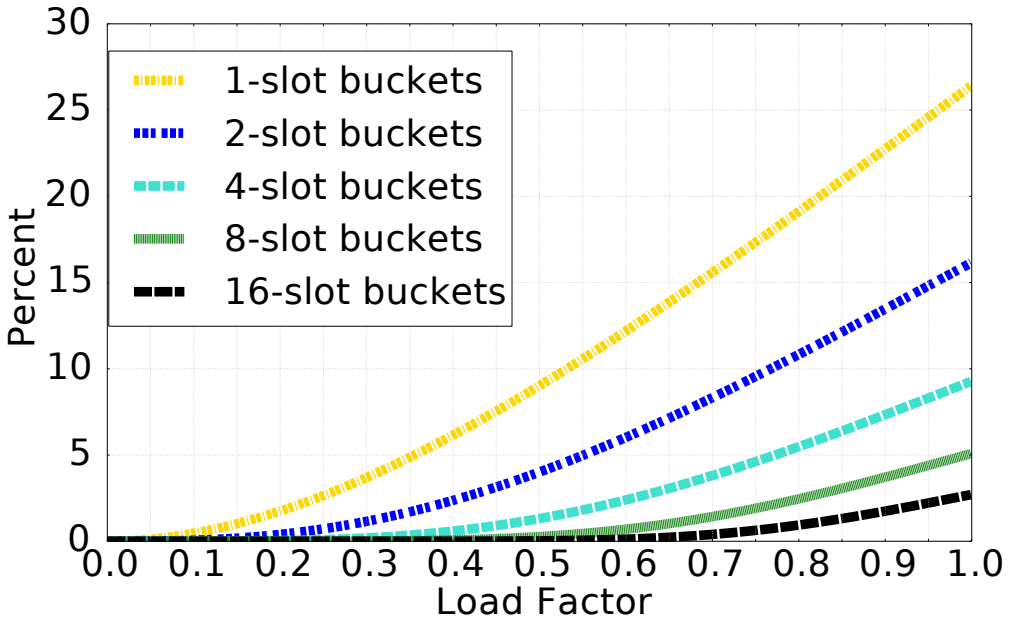


Figure 5.9. Expected percentage of hash table storage that goes to remap entries as the load factor is varied

remap entries), is the product of the fraction of Type B buckets and the consumed space $1/S$ (see Equation 5.4). For simplicity, we assume that each item that overflows a Type B bucket is remappable to a Type A bucket and that these remaps do not cause Type A buckets to become Type B buckets. This approximation is reasonable for two reasons. First, many hash functions can be used to remap items, and second, secondary items are evicted and hashed yet again, when feasible, if they prevent an item from being inserted with the primary hash function $H_{primary}$.

$$O_{re} = \frac{1}{S} \sum_{x=S+1}^{n} P(\lambda, x) \tag{5.4}$$

Figure 5.9 shows the expected percentage of hash table storage that goes to remap entries when varying the number of slots per bucket as well as the load factor. As the remap entries occupy space, the expected maximum load factor is strictly less than or

equal to $1 - O_{re}$. We see that neither 1 slot nor 2 slots per bucket is a viable option if we want to achieve load factors exceeding 90%. Solving for the expected bound on the load factor, we find that 4-, 8-, and 16-slot hash tables are likely to achieve load factors that exceed 91, 95, and 96%, respectively, provided that the remaining space not consumed by remap entries can be almost entirely filled.

5.5.3 Modeling Lookups

The expected average cost of a positive lookup is dependent on the percentage of items that are first-level lookups, the percentage of items that are second-level lookups, and the associated cost of accessing remapped and non-remapped items. For a bucket with S slots, if $x > S$ elements map to that bucket, $x - S + 1$ elements will need to be remapped, as one of those slots now contains remap entries. In the case where $x \leq S$, no elements need to be remapped from that bucket. The fraction of items that require remapping, I_{remap}, is given by Equation 5.5, and the fraction that do not, $I_{primary}$, is given by Equation 5.6. As stated previously, lookups that use $H_{primary}$ require searching one bucket, and lookups that make use of remap entries require searching two. Using this intuition, we combine Equations 5.5 and 5.6 to generate the expected positive lookup cost given by Equation 5.7. Since $I_{primary}$ is a probability, and $1 - I_{remap}$ is equivalent to $I_{primary}$, we can simplify the positive lookup cost to 1 plus the expected fraction of lookups that are secondary. Intuitively, Equation 5.7 makes sense: 100% of lookups need to access the primary bucket. It is only when the item has been remapped that a second bucket needs to be searched.

$$I_{remap} = \frac{\sum_{x=S+1}^{n}(x - S + 1) * P(\lambda, x)}{\lambda} \tag{5.5}$$

$$I_{primary} = \frac{\sum_{x=1}^{S}(x) * P(\lambda, x) + \sum_{x=S+1}^{n}(S-1) * P(\lambda, x)}{\lambda} \tag{5.6}$$

$$\begin{aligned} \textit{Positive Lookup Cost} &= I_{primary} + 2I_{remap} \\ &= 1 \ + \ I_{remap} \end{aligned} \tag{5.7}$$

Figure 5.10 shows the expected positive lookup cost in buckets for 1-, 2-, 4-, 8-, and 16-slot bucket designs. Like before, buckets with more slots are better able to tolerate collisions. Therefore, as the number of slots per bucket increases for a fixed load factor, so does the ratio of Type A buckets to total buckets, which reduces the number of second-level lookups due to not needing to dereference a remap entry. In the 1- and 2-slot bucket cases, the benefit of remap entries is less pronounced but is still present. For the 1-slot case, there is a point at LF = 0.70 where we expect a baseline bucketized cuckoo hash table to touch fewer buckets. However, this scenario is not a fair comparison as, for a baseline BCHT with 1-slot buckets and two functions, the expected load factor does not reach 70%. To reach that threshold, many more hash functions would have to be used, increasing the number of buckets that must be searched. For the 4-, 8-, and 16-slot cases, we observe that the expected lookup cost is under 1.1 buckets for hash tables that are up to 60% full, and that even when approaching the maximum expected load factor, the expected lookup cost is less than 1.3 buckets. In the 8-slot and 16-slot cases, the expected costs at a load factor of 0.95 are 1.18 and 1.1 buckets, which represents a reduced cost of 21% and 27%, respectively, over the baseline.

The cost of a negative lookup follows similar reasoning. On a negative lookup, the secondary bucket is only searched on a false positive tag match in a remap entry. The expected proportion of negative lookups that exhibit tag aliasing, I_{alias}, is the product of the fraction of Type B buckets and the mean fraction of the tag space that is utilized per Type B bucket (Equation 5.8). In the implicit tag scheme, for a 64-bit remap entry array with 21 3-bit entries, the tag space is defined as the set $\{i \ \in \mathbb{N} | \ 0 \leq i \leq 20\}$ and has a tag

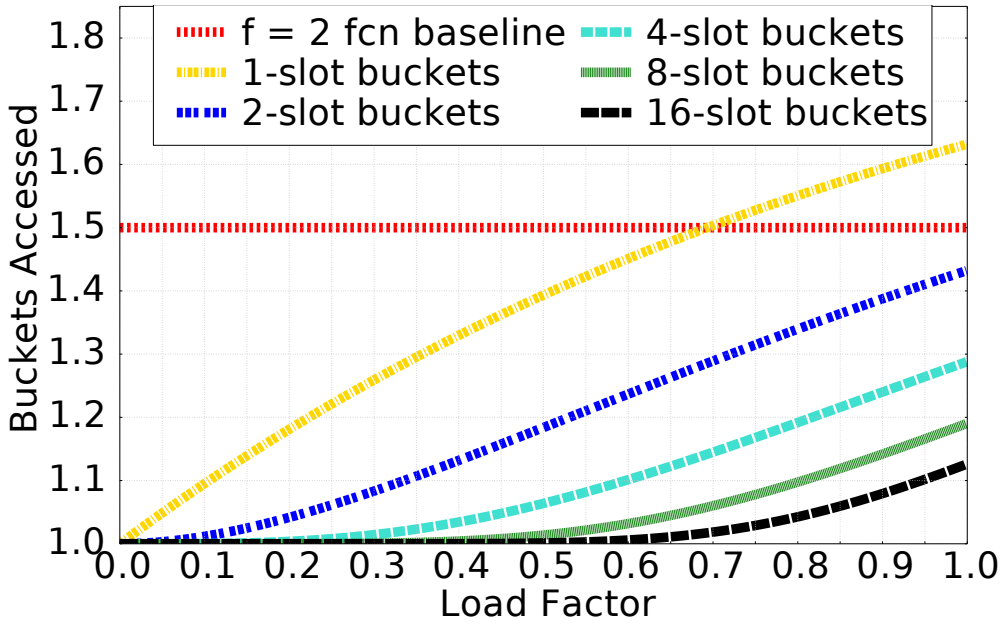


Figure 5.10. The expected buckets accessed per positive lookup in a Horton table vs. a baseline BCHT that uses two hash functions

space cardinality, call it C_{tag}, of 21. Alternatively, with explicit t-bit tags, C_{tag} would be 2^t minus any reserved values for designating states such as `empty`. For our model, we assume that there is a one-to-one mapping between remapped items and remap entries (i.e., each remap entry can only remap a single item). We further assume that conflicts where multiple items map to the same remap entry can be mitigated with high probability by swapping the element that would have mapped to an existing remap entry with an item stored in one of the slots that does not map to an existing remap entry, then rehashing the evicted element, and finally initializing the associated remap entry. These assumptions allow for at most $S-1+C_{tag}$ elements to be stored in or remapped from a Type B bucket.

$$I_{alias} = \frac{\sum_{x=S+1}^{S-1+C_{tag}}((x-S+1) * P(\lambda,x))}{C_{tag}} \tag{5.8}$$

$$\begin{aligned} \textit{Negative Lookup Cost} &= I_{\textit{no alias}} + 2I_{\textit{alias}} \\ &= 1 + I_{\textit{alias}} \end{aligned} \tag{5.9}$$

Like before, we can simplify Equation 5.9 by observing that all lookups need to search at least one bucket, and it is only on a tag alias that we search a second one. Because secondary buckets are in the minority and the number of remapped items per secondary bucket is often small relative to the tag space cardinality, the alias rate given in Equation 5.8 is often quite small, meaning that negative lookups have a cost close to 1.0 buckets.

In Figure 5.11, we plot the expected negative lookup cost for differing numbers of slots per bucket under progressively higher load factors with a tag space cardinality of 21. In contrast to positive lookups, adding more slots has a tradeoff. At low load factors, a table with more slots has a smaller proportion of elements that overflow and less Type B buckets, which reduces the alias rate. However, once the buckets become fuller, having more slots means that buckets have a greater propensity to have more items that need to be remapped, which increases the number of remap entries that are utilized. However, despite these trends, we observe that for 1-, 2-, 4-, 8-, and 16-slot buckets, aliases occur less than 8% of the time under feasible load factors, yielding an expected, worst-case, negative lookup cost of 1.08 buckets. Thus we expect Horton tables to reduce data movement on negative lookups by 46 to 50% versus an $f = 2$ BCHT.

5.6 Experimental Methodology

We run our experiments on a machine with a 4-core AMD A10-7850K with 32GB of DDR3 and an AMD Radeon^{TM} R9-290X GPU with a peak memory bandwidth

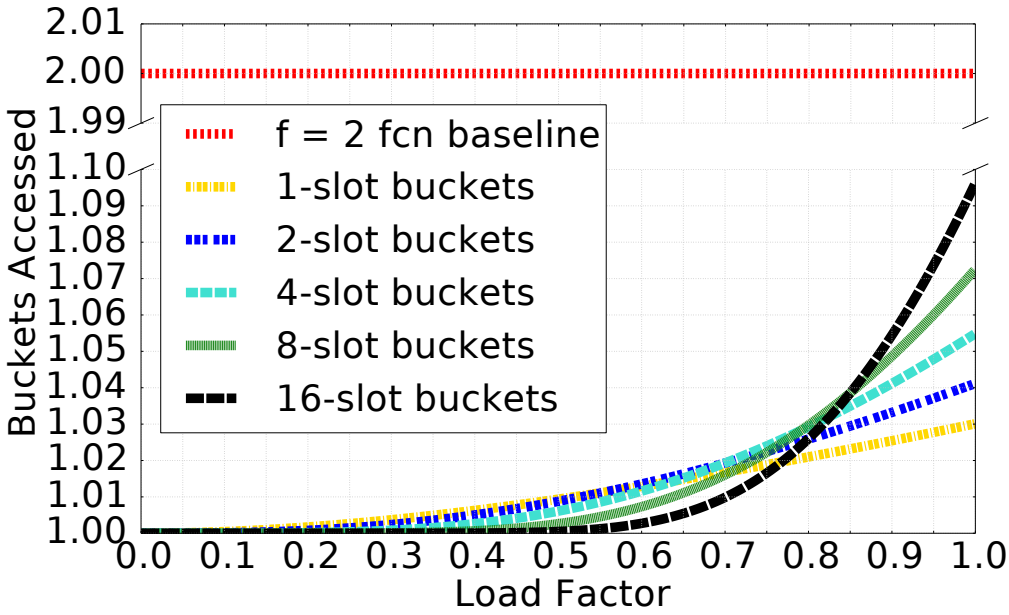


Figure 5.11. The expected buckets accessed per negative lookup in a Horton table vs. a baseline BCHT that uses two hash functions

of 320 GB/s and 4GB of GDDR5. The L2 cache of the GPU is 1 MiB, and each of the 44 compute units has 16 KiB of L1 cache. Our system runs Ubuntu 14.04LTS with kernel version 3.16. Results for performance metrics are obtained by aggregating data from AMD's CodeXL, a publicly available tool that permits collecting high-level performance counters on GPUs when running OpenCL™ programs.

For the performance evaluation, we compare the lookup throughput of the load-balanced baseline, first-fit, and Horton tables. Our baseline implementation is most similar to Mega-KV [255], with the greatest difference being that we only use a single hash table rather than multiple independent partitions and use Jenkins' hash functions.

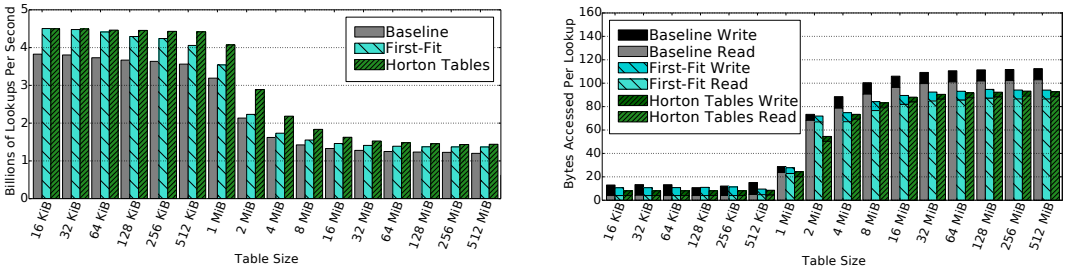
Insertions and deletions are implemented in C and run on the CPU. As our focus is on read-dominated workloads, we assume that insertion and deletion costs can largely be amortized and do not implement parallel versions. For each of the hash table variants, lookup routines are implemented in OpenCL [219] and run on the GPU, with each implementation independently autotuned for fairness of comparison. Toggled parameters

include variable loop unrolling, the number of threads assigned to each compute unit, and the number of key-value pairs assigned to each group of threads to process. When presenting performance numbers, we do not count data transfer cost over PCIe because near-future integrated GPUs will have high-bandwidth, low-latency access to system memory without such overheads. This approach mirrors that of Polychroniou et al. [190] on the Xeon Phi [73].

As part of our evaluation, we validate the models presented in Section 5.5. We calculate the remap entry storage cost and lookup cost per item in terms of buckets by building the hash table and measuring its composition. Unless indicated elsewhere, we use 32-bit keys and values, 8-slot buckets and remap entry arrays with 21 3-bit entries. The probing array that we use for key-value lookups is 1 GiB in size. All evaluated hash tables are less than or equal to 512 MiB due to memory allocation limitations of the GPU; however, we confirmed that we were able to build heavily-loaded Horton tables with more than 2 billion elements without issue.

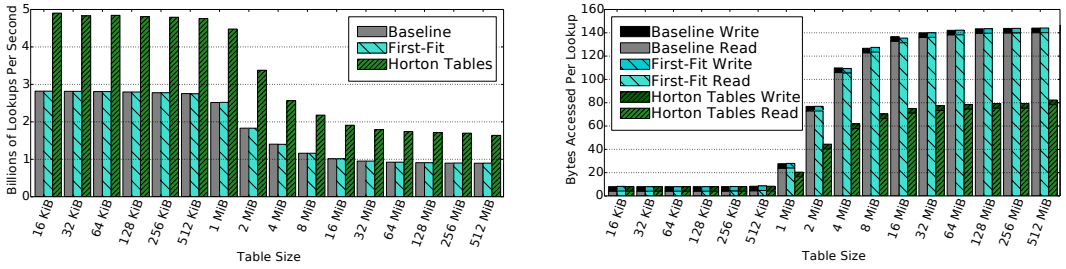
Keys and values for the table and the probing array are generated in the full 32-bit unsigned integer range using C++'s STL Mersenne Twister random integer generator that samples a pseudo-random uniform distribution [168]. We are careful to avoid inserting duplicate keys into the table, as that reduces the effective load factor by inducing entry overwrites rather than storing additional content. Since the probing array contains more keys than there are elements in the hash table, most keys appear multiple times in the probing array. We ensure that all such repeat keys appear far enough from one another such that they do not increase the temporal or spatial locality of accesses to the hash table. This approach is necessary to precisely lower bound the throughput of each algorithm for a given hash table size.

5.7 Results



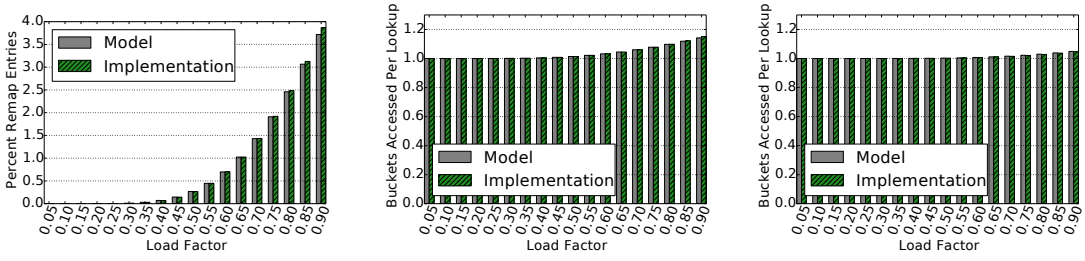
(a) Lookup throughput **(b)** Data movement from global memory

Figure 5.12. Comparison of BCHTs with a Horton table (load factor = 0.9 and 100% of queried keys found in table)



(a) Lookup throughput **(b)** Data movement from global memory

Figure 5.13. Comparison of BCHTs with a Horton table (load factor = 0.9 and 0% of queried keys found in table)



(a) Remap entry storage cost **(b)** Positive lookup cost **(c)** Negative lookup cost

Figure 5.14. Validation of our models on an 8 MiB Horton table

In this section, we validate our models and present performance results. Figures 5.12a and 5.12b compare the lookup throughput and data movement (as seen by

the global memory) between the load-balancing baseline (Section 5.2.3), BCHT with first-fit insert heuristic (Section 5.2.3) and Horton tables (Section 5.3) for tables from 16 KiB to 512 MiB in size. We see that Horton tables increase throughput over the baseline by 17 to 35% when all of the queried keys are in the table. In addition, they are faster than a first-fit approach, as Horton tables enforce primacy (Section 5.4.3, Guideline 1) on remapping of elements whereas first-fit does not. This discrepancy is most evident from 512 KiB to 8 MiB, where Horton tables are up to 29% faster than first-fit BCHTs.

These performance wins are directly proportional to the data movement saved. Initially, there is no sizeable difference in the measured data movement cost between the baseline, first-fit, and Horton tables, as the hash tables entirely fit in cache. Instead, the bottleneck to performance is the cache bandwidth. However, at around 1 MiB, the size at which the table's capacity is equal to the size of the last level cache (L2), the table is no longer fully cacheable in L2, and so it is at this point that the disparity in data movement between the three approaches becomes visible at the off-chip memory.

Figures 5.13a and 5.13b show the opposite extreme where none of the queried keys are in the table. In this case, Horton tables increase throughput by 73 to 89% over the baseline and first-fit methods because, unlike a BCHT, Horton tables can satisfy most negative searches with one bucket access. These results and those for positive lookups from Figures 5.12a and 5.12b align very closely with the reduction in data movement that we measured with performance counters. For a workload consisting entirely of positive lookups, baseline BCHTs access 30% more cache lines than Horton tables. At the opposite extreme, for a workload of entirely negative lookups, both first-fit and baseline BCHTs access 90% more hash table cache lines than Horton tables.

If we examine the total data movement, we find that both our BCHT and Horton table implementations move an amount of data close to what our models project. At a load factor of 0.9, our model predicts 1.15 and 1.05 buckets accessed per positive and

negative query, respectively. Since cache lines are 64 bytes, this cost corresponds to 74 and 67 bytes per query worth of data movement. On top of that, for each lookup query we have an additional 8 bytes of data movement for loading the 4-byte query key and 4 bytes for storing the retrieved value, which puts our total positive and negative lookup costs at 82 and 75 bytes, respectively. These numbers are within 10% of the total data movement that we observe in Figures 5.12b and 5.13b once the hash table is much larger than the size of the last-level cache. Similarly, we found that our models' data movement estimates for BCHTs were within similar margins of our empirical results.

Figures 5.14a, 5.14b, and 5.14c show that each of our models accurately capture the characteristics of our implementation. On average, our table requires fewer than 1.15 bucket lookups for positive lookups and fewer than 1.05 for negative lookups at a load factor of 0.9, and both have a cost of essentially 1.0 up to a load factor of 0.55. These results are a dramatic improvement over the current state of practice and validate the soundness of our algorithms to achieve high load factors without measurably compromising on the percentage of primary lookups.

5.8 Related Work

There has been a long evolution in hash tables. Two pieces of work that share commonality with our own are the cuckoo filter [89] and MemC3 [90]. MemC3 is a fast, concurrent alternative to Memcached [96] that uses a bucketized hash table to index data, with entries that consist of (1) tags and (2) pointers to objects that house the full key, value, and additional metadata. For them, the tag serves two primary functions: (1) to avoid polluting the cache with long keys on most negative lookups and (2) to allow variable-length keys. Tags are never used to avoid bringing additional buckets into cache. If the element is not found in the first bucket, the second bucket is always searched. Similarly, the cuckoo filter is also an $f = 2$ function, 4-slot bucketized cuckoo

hash set that is designed to be an alternative to Bloom filters [52] that is cache friendly and supports deletions.

Another related work is Stadium Hashing [134]. Their focus is to have a fast hash table where the keys are stored on the GPU and the values in the CPU's memory. Unlike us they use a non-bucketized hash table with double hashing and prime capacity. They employ an auxiliary data structure known as a ticket board to filter requests between the CPU and GPU and also to permit concurrent put and get requests to the same table. Barber et al. use a similar bitmap structure to implement two compressed hash table variants [41].

The BCHT [87] combines cuckoo hashing and bucketization [98, 185, 186]. Another improvement to cuckoo hashing is by way of a stash—a small, software victim cache for evicted items [138].

Other forms of open addressing are also prevalent. Quadratic hashing, double hashing, Robin Hood hashing [68], and linear probing are other commonly used open addressing techniques [76]. Hopscotch hashing attempts to move keys to a preferred neighborhood of the table by displacing others [114]. It maintains a per-slot *hop information* field that is often several bits in length that tracks element displacements to nearby cells. By contrast, Horton tables only create remap entries when buckets exceed their baseline capacity.

Other work raises the throughput of concurrent hash tables by using lock free approaches [172,228,229] or fine-grain spinlocks [153]. Additional approaches attempt to fuse or use other data structures in tandem with hash tables to enable faster lookups [154, 198, 217].

In application, hash tables find themselves used in a wide of variety of data warehousing and processing applications. A number of in-memory key-value stores employ hash tables [90, 96, 115, 181, 255], and others accelerate key lookups by locating

the hash table on the GPU [115, 116, 255]. Early GPU hash tables have been primarily developed for accelerating applications in databases, graphics and computer vision [26, 27, 100, 142, 151]. In in-memory databases, there has been significant effort spent on optimizing hash tables due to their use in hash join algorithms on CPUs [39,40,50,54,190], coupled CPU-GPU systems [109, 111, 128], and the Xeon Phi [123, 190].

This work on high-performance hash tables is complementary to additional research efforts that attempt to retool other indexes such as trees to take better advantage of system resources and new and emerging hardware [152, 155, 162, 246].

5.9 Conclusion

This chapter presents the Horton table, an enhanced bucketized cuckoo hash table that achieves higher throughput by reducing the number of hardware cache lines that are accessed per lookup. It uses a single function to hash most elements and can therefore retrieve most items by accessing a single bucket, and thus a single cache line. Similarly, most negative lookups can also be satisfied by accessing one cache line. These low access costs are enabled by remap entries: sparingly allocated, in-bucket records that enable both cache and off-chip memory bandwidth to be used much more efficiently. Accordingly, Horton tables increase throughput for positive and negative lookups by as much as 35% and 89%, respectively. Best of all, these improvements do not sacrifice the other attractive traits of baseline BCHTs: worst-case lookup costs of 2 buckets and load factors that exceed 95%.

5.10 Acknowledgements

The authors thank the reviewers for their insightful comments and Geoff Kuenning, our shepherd, for his meticulous attention to detail and constructive feedback. We were very impressed with the level of rigor that was applied throughout the review and

revision process. We also thank our peers at AMD Research for their comments during internal presentations of the work and Geoff Voelker for providing us invaluable feedback that elevated this chapter's quality.

Chapter 5, in full, is a reprint of the material as it appears in Proceedings of the USENIX Annual Technical Conference 2016. Breslow, Alexander D.; Zhang, Dong Ping, Greathouse, Joseph L., Jayasena, Nuwan, Tullsen, Dean M., USENIX Organization, June 2016. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Concluding Remarks

In this dissertation, we explore several key problems within the domain of accurately measuring and improving application performance and energy efficiency for today's often memory-limited multicore systems. We demonstrated that this domain presents a number of challenges but also that relatively simple solutions can yield modest to large performance and energy improvements.

Chapter 2 explores the performance and energy efficiency opportunities afforded by investigating two alternate policies for mapping large, distributed supercomputer applications to compute nodes. The first, job spreading, explored undersubscribing compute nodes as a way to reduce memory subsystem contention. When undersubscribing compute nodes by a factor of 2, some applications run more than 2x faster. For them, using more nodes for a fixed number of processes is more cost effective for the user than requesting the minimum number of compute nodes necessary to assign one MPI task to each core. The second, job striping, builds on the intuition behind job spreading by scheduling pairs of applications on a shared set of servers, with each application receiving half of the cores per socket. Job striping yields most of the per-application performance improvements of job spreading but requires only half as many machines to process both jobs concurrently. On real-world scientific applications at a scale of 128 machines, job striping improves throughput and reduce energy by 12% and 11%,

respectively. These figures are significant because 12% more jobs can be processed on the same infrastructure for the same budget. Alternatively, a smaller system could be bought to run the same load, which would reduce cost and allow for more systems to be built.

Chapter 3 examines one of the practical implications of transitioning to employing job striping on a supercomputer, which is how to fairly handle accounting of applications that share compute nodes. Because co-located applications have threads that can contend for the same scarce memory subsystem, network, and IO resources, co-located applications affect one another's run time. Depending on the co-located jobs and the types of resource sensitivities that each job possesses coupled with the types of resource demands that their partners make, run time for a job can be markedly different. This disparity in run time requires revamping the default job pricing system that bills users proportionally to application run time. Otherwise jobs that are particularly sensitive to resource contention pay comparatively more than peers that are not. Further, the potential variability in job pricing makes it difficult for a scientist to properly budget their limited pool of credits. To combat these challenges, we propose and implement a prototype runtime system that dynamically determines contention between co-running applications that share a node. These degradation estimates then served as inputs to a revised accounting scheme that achieves greater fairness by discounting users when their jobs suffer degradation. This system shows it is indeed possible to support fair pricing in environments where jobs are co-located.

Chapter 4 shifts focus to combating memory subsystem contention for GPU-accelerated databases. We focus on GPUs due to their continued strong performance scaling and because their high memory bandwidth and ability to hide memory latency through hardware multithreading aligns well with the data-intensive nature of analytical database workloads. However, even a GPU is comparatively bandwidth-limited compared

to its peak compute capability, and so we pose a simple question: can a GPU even use all of its parallel capability productively on database workloads, and if not, how much can query latency and energy be reduced through disabling whole compute units? What we found is that when all compute units are active, many of the TPC-H queries experience last level cache hit ratios below 20%. However, as compute units are disabled, these figures drastically improve due to fewer threads actively contending for the last level cache. By optimally disabling compute units at the query granularity, query run time and energy use reduce by as much as 24% and 42%, respectively, without modifying existing database code. On the whole, there is a large potential to save energy in this manner on systems with GPUs and coupling this technique with other methods will yield further opportunities for improving performance and reducing energy use.

Chapter 5 proposes the Horton table, a novel hash table design that is specifically tailored to make efficient use of cache and memory bandwidth. The study in Chapter 4 demonstrates that for several of the queries, the majority of time on the GPU is spent executing kernels that operate on hash tables. This concurs with prior work that demonstrates that high-performance hash tables are necessary for a fast, OLAP query processing system, because of their breadth of time-intensive uses. They employed both in joins and grouping operations, each of which typically consume a considerable portion of query processing time. At the same time, large data workloads often employ hash tables that are many times larger than the last level cache, and thus most table lookups require transferring cache lines over the relatively narrow memory buses from main memory. Thus any savings in the number of unique cache lines accessed per lookup query significantly improves the number of key-value pairs that can be retrieved from memory per unit time. Motivated by this knowledge, we identify that the prior state-of-the-art bucketized cuckoo hash table often accesses 1.5x to 2.0x more cache lines than should be ideally necessary when satisfying a lookup query. The Horton table largely reduces

this waste and improves lookup performance by as much as 1.95x.

Going forward we anticipate conducting continued work in workload management and scheduling (Chapters 2 and 3), tuning of software parallelism on parallel computing platforms (Chapters 2 and 4), and application-level optimizations for data movement (Chapter 5). There is significant research that remains to be done in resolving the best way to harness multicore systems. The recent move to multicore and the relatively nascent support for integration of accelerators such as GPUs means that many applications can only capture a small amount of the performance and energy efficiency provided by today's hardware. New operating system primitives, application libraries, runtime systems, and computing frameworks are necessary to address the challenges facing performance scaling in an age of increasingly diverse and complex hardware that offers a rich set of features. We intend to remain active in this space and to continue advancing software technologies that better enable use of computing platforms from across the spectrum.

Bibliography

[1] A Brief Intro to the Heterogeneous Compute Compiler. http://www.gpuopen.com/a-brief-intro-to-boltzman-hcc. Accessed: 2016-11-07.

[2] ASC Sequoia Benchmark Codes. https://asc.llnl.gov/sequoia/benchmarks/.

[3] Big Data Universe Beginning to Explode. http://www.csc.com/insights/flxwd/78931-big_data_universe_beginning_to_explode. Accessed: 2016-08-30.

[4] Breaking the Memory Barrier with Radeon Pro SSG. http://www.radeon.com/en-us/radeon-pro-ssg. Accessed: 2016-11-07.

[5] Cache Coherent Interconnect for Accelerators (CCIX). http://www.ccixconsortium.com/. Accessed: 2016-11-07.

[6] CESM1.0: PARALLEL OCEAN PROGRAM (POP2). http://www.cesm.ucar.edu/models/cesm1.0/pop2/.

[7] Developing for OpenPOWER and NVIDIA NVLink. https://developer.nvidia.com/openpower. Accessed: 2016-11-07.

[8] FurMark: GPU Stress Test OpenCL Benchmark. http://www.ozone3d.net/benchmarks/fur/. Accessed: 2016-03-23.

[9] HCC is an Open Source, Optimizing C++ Compiler for Heterogeneous Compute. https://github.com/RadeonOpenCompute/hcc/wiki. Accessed: 2016-11-07.

[10] High-Bandwidth Memory (HBM): Reinventing Memory Technology. https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf. Accessed: 2016-11-07.

[11] Large-scale Atomic/Molecular Massively Parallel Simulator. http://lammps.sandia.gov/.

[12] Mantevo Suite. http://www.mantevo.org/.

[13] MIMD Lattice Computation (MILC) Collaboration. http://www.physics.indiana.edu/~sg/milc.html.

[14] Nek5000 Project. https://nek5000.mcs.anl.gov/index.php/Main_Page.

[15] Perfmon 2: Improving Performance Monitoring on Linux. http://perfmon2.sourceforge.net/.

[16] Proxy-Apps for Thermal Hydraulics. https://cesar.mcs.anl.gov/content/software/thermal_hydraulics.

[17] SPEC CPU 2000 Benchmark Suite. www.spec.org/cpu2000.

[18] The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm. Accessed: 2016-08-30.

[19] Whitepaper — AMD Graphics Core Next (GCN) Architecture. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf. Accessed: 2016-06-30.

[20] Gordon User Guide. http://www.sdsc.edu/us/resources/gordon/, 2012.

[21] Extreme Science and Engineering Discovery Environment. www.xsede.org, 2013.

[22] Innovative and Novel Computational Impact on Theory and Experiment. http://www.doeleadershipcomputing.org/incite-program/, 2013.

[23] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now, 2013.

[24] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*, abs/1603.04467, 2016.

[25] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, 1999.

[26] Dan A Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Real-Time Parallel Hashing on the GPU. In *Proc. of the ACM SIGGRAPH Conf. and Exhibition on Computer Graphics and Interactive Techniques in Asia (SIGGRAPH Asia)*, 2009.

[27] Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Building an Efficient Hash Table on the GPU. In Wen-mei W. Hwu, editor, *GPU Computing Gems: Jade Edition*, chapter 4, pages 39–54. Morgan Kaufmann, 2011.

[28] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*, volume 10, page 24, 2010.

[29] J.H. Anderson, J.M. Calandrino, and U.C. Devi. Real-Time Scheduling on Multicore Platforms. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr. 2006.

[30] Katie Antypas, John Shalf, and Harvey Wasserman. NERSC-6 Workload Analysis and Benchmark Selection Process. Technical report, Lawrence Berkeley National Laboratory, 2008.

[31] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs Scale-out for Hadoop: Time to Rethink? In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 20. ACM, 2013.

[32] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4), 2010.

[33] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A View of the Parallel Computing Landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[34] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[35] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel BenchmarksSummary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[36] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K.

Weeratunga. The NAS Parallel Benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[37] Allison H Baker, Todd Gamblin, Martin Schulz, and Ulrike Meier Yang. Challenges of Scaling Algebraic Multigrid Across Modern Multicore Architectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011.

[38] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103. ACM, 2010.

[39] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. of the VLDB Endowment*, 7(1):85–96, 2013.

[40] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Ozsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, 2013.

[41] Ronald Barber, G Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, G Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. Memory-Efficient Hash Joins. *Proc. of the VLDB Endowment*, 8(4):353–364, 2014.

[42] Luiz André Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40(12):33–37, December 2007.

[43] Scott Beamer, Krste Asanovic, and David Patterson. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 56–65. IEEE, 2015.

[44] P. Beckman. Looking Toward Exascale Computing. In *Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008. Ninth International Conference on*, pages 3–3. IEEE, 2008.

[45] Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. Don't Thrash: How to Cache Your Hash on Flash. *Proc. of the VLDB Endowment*, 5(11):1627–1637, 2012.

[46] E. Berg and E. Hagersten. StatCache: a Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *International Symposium on Performance Analysis of Systems and Software*, 2004.

[47] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems, www.cse.nd.edu/Reports/2008TR-2008-13.pdf, 2008.

[48] Sergey Blagodurov and Alexandra Fedorova. Towards the Contention Aware Scheduling in HPC Cluster Environment. In *Journal of Physics: Conference Series*, volume 385. IOP Publishing, 2012.

[49] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-Aware Scheduling on Multicore Systems. *ACM Transactions on Computer Systems*, 28, 2010.

[50] Spyros Blanas, Yinan Li, and Jignesh M Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)*, 2011.

[51] Jacek Blazewicz, Jan Karel Lenstra, and AHG Kan. Scheduling Subject to Resource Constraints: Classification and Complexity. *Discrete Applied Mathematics*, 5(1), 1983.

[52] Burton H Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[53] Peter A Boncz, Martin L Kersten, and Stefan Manegold. Breaking the Memory Wall in MonetDB. *Communications of the ACM*, 51(12):77–85, 2008.

[54] Peter A Boncz, Stefan Manegold, and Martin L Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, 1999.

[55] Peter A Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, volume 5, pages 225–237, 2005.

[56] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An Improved Construction for Counting Bloom Filters. In *Proc. of the Annual European Symposium (ESA)*, 2006.

[57] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, 2014.

[58] Alex D. Breslow, Leo Porter, Ananta Tiwari, Michael Laurenzano, Laura Carrington, Dean M. Tullsen, and Allan E. Snavely. The Case for Colocation of High

Performance Computing Workloads. *Concurrency and Computation: Practice and Experience*, 2013.

[59] Alex D Breslow, Ananta Tiwari, Martin Schulz, Laura Carrington, Lingjia Tang, and Jason Mars. Enabling Fair Pricing on HPC Systems with Node Sharing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 37. ACM, 2013.

[60] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016.

[61] Sebastian Breß. The Design and Implementation of CoGaDB: A Column-Oriented GPU-Accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[62] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. GPU-Accelerated Database Systems: Survey and Open Challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[63] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP 2010-The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

[64] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures. *International Journal of Parallel Programming*, 38:418–439, 2010.

[65] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Supercomputing, ACM/IEEE 2000 Conference*, page 42, nov. 2000.

[66] F. Cappello. Fault Tolerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *International Journal of High Performance Computing Applications*, 23(3):212–226, 2009.

[67] Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernández, Alex Ramirez, and Mateo Valero. Predictable Performance in SMT Processors. In *1st Conference on Computing Frontiers*, 2004.

[68] Pedro Celis, Per-Åke Larson, and Ian J Munro. Robin Hood Hashing. In *Proc. of the IEEE Annual Symp. on Foundations of Computer Science (FOCS)*, 1985.

[69] Lei Chai, Qi Gao, and Dhabaleswar K Panda. Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*, pages 471–478. IEEE, 2007.

[70] Lei Chai, Albert Hartono, and Dhabaleswar K Panda. Designing High Performance and Scalable MPI Intra-Node Communication Support for Clusters. In *2006 IEEE International Conference on Cluster Computing*, pages 1–10. IEEE, 2006.

[71] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *11th International Symposium on High-Performance Computer Architecture*, 2005.

[72] Jichuan Chang and Gurindar S Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing*. ACM, 2007.

[73] George Chrysos and Senior Principal Engineer. Intel Xeon Phi Coprocessor (Codename Knights Corner). Presented at Hot Chips, 2012.

[74] Edgar F Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[75] Edgar F Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems (TODS)*, 4(4):397–434, 1979.

[76] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[77] Ayse K. Coskun, Richard Strong, Dean M. Tullsen, and Tajana Simunic Rosing. Evaluating the Impact of Job Scheduling and Power Management on Processor Lifetime for Chip Multiprocessors. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 169–180, New York, NY, USA, 2009. ACM.

[78] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[79] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[80] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)*, 1984.

[81] Dr. Seuss. *Horton Hatches the Egg*. Random House, 1940.

[82] Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, and Jian Pei. A Practical Method for Estimating Performance Degradation on Multicore Processors, and its Application to HPC Workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012.

[83] David Eklov, David Black-Schaffer, and Erik Hagersten. Fast Modeling of Shared Caches in Multicore Systems. In *6th International Conference on High Performance and Embedded Architectures and Compilers*, 2011.

[84] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Cache Pirating: Measuring the Curse of the Shared Cache. In *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 2011.

[85] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Bandwidth Bandit: Understanding Memory Contention. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*. IEEE, 2012.

[86] Stephane Eranian. Perfmon: Linux Performance Monitoring for IA-64. *Downloadable software with documentation, http://www. hpl. hp. com/research/linux/perfmon*, 2003.

[87] Ulfar Erlingsson, Mark Manasse, and Frank McSherry. A Cool and Practical Alternative to Traditional Hash Tables. In *Proc. of the Workshop on Distributed Data and Structures (WDAS)*, 2006.

[88] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[89] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *Proc. of the ACM Int'l Conf. on Emerging Networking Experiments and Technologies (CoNEXT)*, 2014.

[90] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of the USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2013.

[91] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary Cache: a Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[92] Kayvon Fatahalian and Mike Houston. A Closer Look at GPUs. *Communications of the ACM*, 51(10):50–57, 2008.

[93] Alexandra Fedorova, Margo Seltzer, Christoper Small, and Daniel Nussbaum. Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design. In *USENIX Annual Technical Conference*, 2005.

[94] Alexandra Fedorova, Margo Seltzer, and Michael D Smith. Cache-Fair Thread Scheduling for Multicore Processors. *Division of Engineering and Applied Sciences, Harvard University, Tech. Rep. TR-17-06*, 2006.

[95] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[96] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5, Aug. 2004.

[97] Krisztián Flautner and Trevor Mudge. Vertigo: Automatic Performance-Setting for Linux. *ACM SIGOPS Operating Systems Review*, 36(SI):105–116, 2002.

[98] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space Efficient Hash Tables with Worst Case Constant Access Time. In *Proc. of the Annual Symp. on Theoretical Aspects of Computer Science (STACS)*, 2003.

[99] Glenn Fowler and Landon Curt Noll. The FNV Non-Cryptographic Hash Algorithm. http://tools.ietf.org/html/draft-eastlake-fnv-03. Accessed: 2015-12-01.

[100] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Coherent Parallel Hashing. In *Proc. of the ACM SIGGRAPH Conf. and Exhibition on Computer Graphics and Interactive Techniques in Asia (SIGGRAPH Asia)*, 2011.

[101] Michael R Garey and David S. Johnson. Complexity Results for Multiprocessor Scheduling Under Resource Constraints. *SIAM Journal on Computing*, 4(4), 1975.

[102] Ricardo Gonzalez and Mark Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of solid-state circuits*, 31(9):1277–1284, 1996.

[103] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: Quantifying Effects of Shared On-Chip Resource Interference for Consolidated Virtual Machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011.

[104] Joseph L Greathouse and Mayank Daga. Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format. In *SC14: International*

Conference for High Performance Computing, Networking, Storage and Analysis, pages 769–780. IEEE, 2014.

[105] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel computing*, 22(6):789–828, 1996.

[106] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-Aware Scheduling and Analysis for Multicores. In *7th ACM International Conference on Embedded software*, 2009.

[107] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. DjiNN and Tonic: DNN as a Service and its Implications for Future Warehouse Scale Computers. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 27–40. ACM, 2015.

[108] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational Query Coprocessing on Graphics Processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.

[109] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational Joins on Graphics Processors. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)*, 2008.

[110] Jiahua He, Arun Jagatheesan, Sandeep Gupta, Jeffrey Bennett, and Allan Snavely. DASH: a Recipe for a Flash-based Data Intensive Supercomputer. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[111] Jiong He, Mian Lu, and Bingsheng He. Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. of the VLDB Endowment*, 6(10):889–900, 2013.

[112] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-Oblivious Parallelism for In-Memory Column-Stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013.

[113] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

[114] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch Hashing. In *Proc. of the Int'l Symp. on Distributed Computing (DISC)*, 2008.

[115] Tayler H. Hetherington, Mike O'Connor, and Tor M. Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proc. of the ACM Symp. on Cloud Computing (SoCC)*, 2015.

[116] Tayler H Hetherington, Timothy G Rogers, Lisa Hsu, Mike O'Connor, and Tor M Aamodt. Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems. In *Proc. of the IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2012.

[117] H Peter Hofstee. Power Efficient Processor Architecture and the Cell Processor. In *Proc. of the Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2005.

[118] C. Iancu, S. Hofmeyr, F. Blagojevic, and Yili Zheng. Oversubscription on Multicore Processors. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –11, april 2010.

[119] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. *ACM SIGARCH Computer Architecture News*, 35(2), 2007.

[120] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[121] Bob Jenkins. 4-byte Integer Hashing. http://burtleburtle.net/bob/hash/integer.html. Accessed: 2015-12-01.

[122] Bob Jenkins. Some Random Theorems. http://burtleburtle.net/bob/hash/birthday.html. Accessed: 2015-12-01.

[123] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proc. of the VLDB Endowment*, 8(6):642–653, 2015.

[124] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and Approximation of Optimal Co-scheduling on Chip Multiprocessors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008.

[125] Yang Jiao, Heshan Lin, Pavan Balaji, and Wu-chun Feng. Power and Performance Characterization of Computational Kernels on the GPU. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 221–228. IEEE, 2010.

[126] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ACM SIGPLAN Notices*, volume 48, pages 395–406. ACM, 2013.

[127] Hyungsoo Jung, Hyuck Han, Alan Fekete, Gernot Heiser, and Heon Y Yeom. A Scalable Lock Manager for Multicores. *ACM Transactions on Database Systems (TODS)*, 39(4):29, 2014.

[128] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU Join Processing Revisited. In *Proc. of the Int'l Workshop on Data Management on New Hardware (DaMoN)*, 2012.

[129] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles H Still. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013), Boston, USA*, 2013.

[130] Hironori Kasahara and Seinosuke Narita. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. *IEEE Transactions on Computers*, 33(11), 1984.

[131] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. Neither More nor Less: Optimizing Thread-Level Parallelism for GPGPUs. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 157–166. IEEE Press, 2013.

[132] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, (5):7–17, 2011.

[133] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, 2011.

[134] Farzad Khorasani, Mehmet E. Belviranli, Rajiv Gupta, and Laxmi N. Bhuyan. Stadium Hashing: Scalable and Flexible Hashing on GPUs. In *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[135] Joonyoung Kim and Younsu Kim. HBM: Memory Solution for Bandwidth-Hungry Processors. In *Hot Chips*, volume 26, 2014.

[136] Nam Sung Kim, Todd Austin, David Baauw, Trevor Mudge, Krisztián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage Current: Moore's Law Meets Static Power. *Computer*, 36(12):68–75, 2003.

[137] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *13th International Conference on Parallel Architecture and Compilation Techniques*, Sept. 2004.

[138] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2009.

[139] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro 28, 3*, 2008.

[140] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the Walkers: Accelerating Index Traversals for In-Memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 468–479. ACM, 2013.

[141] Matthew J. Koop, Miao Luo, and Dhabaleswar K. Panda. Reducing Network Contention with Mixed Workloads on Modern Multicore Clusters. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[142] Simon Korman and Shai Avidan. Coherency Sensitive Hashing. In *Proc. of the IEEE Int'l Conf. on Computer Vision (ICCV)*, 2011.

[143] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[144] Michael Larabel. The OpenGL Speed & Performance-Per-Watt From The Radeon RX 480 To Radeon HD 4850/4870. http://www.phoronix.com/scan.php?page=article&item=amd-rv770-rx480&num=1. Accessed: 2016-06-30.

[145] Michael A. Laurenzano, Mitesh Meswani, Laura Carrington, Allan Snavely, Mustafa M. Tikir, and Stephen Poole. Reducing Energy Usage with Memory and Computation-Aware Dynamic Frequency Scaling. In *Proceedings of the 17th international Euro-Par conference on Parallel processing*, EuroPar'11, Bordeaux, France, 2011.

[146] Etienne Le Sueur and Gernot Heiser. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8, 2010.

[147] Chang-Chi Lee, Cp Hung, Calvin Cheung, Ping-Feng Yang, Chin-Li Kao, Dao-Long Chen, Meng-Kai Shih, Chien-Lin Chang Chien, Yu-Hsiang Hsiao, Li-Chieh Chen, Michael Su, Michael Alfano, Joe Siegel, Julius Din, and Bryan Black. An Overview of the Development of a GPU with Integrated HBM on Silicon Interposer. In *Electronic Components and Technology Conference (ECTC), 2016 IEEE 66th*, pages 1439–1444. IEEE, 2016.

[148] Cynthia B Lee and Allan E Snavely. Precise and Realistic Utility Functions for User-Centric Performance Analysis of Schedulers. In *Proceedings of the 16th international symposium on High performance distributed computing*, pages 107–116. ACM, 2007.

[149] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 260–271. IEEE, 2014.

[150] 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 *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2010.

[151] Sylvain Lefebvre and Hugues Hoppe. Perfect Spatial Hashing. In *Proc. of the ACM SIGGRAPH Conf. and Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2006.

[152] Justin J Levandoski, David B Lomet, and Sabyasachi Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, 2013.

[153] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proc. of the European Conf. on Computer Systems (EuroSys)*, 2014.

[154] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, 2011.

[155] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proc. of the USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2014.

[156] Seung-Hwan Lim, Jae-Seok Huh, Youngjae Kim, Galen M Shipman, and Chita R Das. D-factor: a Quantitative Model of Application Slow-Down in Multi-Resource Shared Systems. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. ACM, 2012.

[157] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap Between Simulation and Real Systems. In *14th International Symposium on High Performance Computer Architecture*, 2008.

[158] Z Lin, G Rewoldt, S Ethier, T S Hahm, W W Lee, J L V Lewandowski, Y Nishimura, and W X Wang. Particle-in-Cell Simulations of Electron Transport from Plasma Turbulence: Recent Progress in Gyrokinetic Particle Simulations of Turbulent Plasmas. *Journal of Physics: Conference Series*, 16(1), 2005.

[159] Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Pete Wyckoff, and Dhabaleswar K Panda. Performance Comparison of MPI Implementations Over InfiniBand, Myrinet and Quadrics. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 58–58. IEEE, 2003.

[160] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 301–312. IEEE Press, 2014.

[161] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. Managing Variability in the IO Performance of Petascale Storage Systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE Computer Society, 2010.

[162] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proc. of the European Conf. on Computer Systems (EuroSys)*, 2012.

[163] Bernard Marr. Big Data: 20 Mind-Boggling Facts Everyone Must Read. http://www.forbes.com/sites/bernardmarr/2015/09/30/big-data-20-mind-boggling-facts-everyone-must-read, September 2016.

[164] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *MICRO '11: Proceedings of The 44th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2011. ACM.

[165] Jason Mars, Lingjia Tang, and Mary Lou Soffa. Directly Characterizing Cross Core Interference Through Contention Synthesis. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. ACM, 2011.

[166] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention Aware Execution: Online Contention Detection and Response. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010.

[167] Matt Martineau, Simon McIntosh-Smith, and Wayne Gaudin. Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 338–347. IEEE, 2016.

[168] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[169] Sally A McKee. Reflections on the Memory Wall. In *Proc. of the ACM Int'l Conf. on Computing Frontiers (CF)*, 2004.

[170] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[171] David Meisner, Brian T Gold, and Thomas F Wenisch. PowerNap: Eliminating Server Idle Power. In *ACM Sigplan Notices*, volume 44, pages 205–216. ACM, 2009.

[172] Zviad Metreveli, Nickolai Zeldovich, and M Frans Kaashoek. CPHash: A Cache-Partitioned Hash Table. In *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[173] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 12(10):1094–1104, 2001.

[174] Michael Mitzenmacher. Balanced Allocations and Double Hashing. In *Proc. of the ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2014.

[175] Gordon E Moore. Cramming More Components onto Integrated Circuits. *Electronics*, pages 114–117, April 1965.

[176] Richard L. Moore, David L. Hart, Wayne Pfeiffer, Mahidhar Tatineni, Kenneth Yoshimoto, and William S. Young. Trestles: a High-Productivity HPC System Targeted to Modest-Scale and Gateway Users. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, New York, NY, USA, 2011. ACM.

[177] Frank Mueller. Compiler Support for Software-Based Cache Partitioning. In *ACM Sigplan Notices*, volume 30. ACM, 1995.

[178] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a Timely Dataflow System. In *Proceedings of the*

Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 439–455. ACM, 2013.

[179] NAS. Nas parallel benchmarks website, http://www.nas.nasa.gov/Resources/Software/npb.html.

[180] Ripal Nathuji and Karsten Schwan. VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 265–278. ACM, 2007.

[181] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proc. of the USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2013.

[182] Di Niu, Chen Feng, and Baochun Li. Pricing Cloud Bandwidth Reservations Under Demand Uncertainty. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. ACM, 2012.

[183] M.L. Norman and A. Snavely. Accelerating Data-Intensive Science with Gordon and Dash. In *2010 TeraGrid Conference*, 2010.

[184] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[185] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[186] Rina Panigrahy. Efficient Hashing with Lookups in Two Memory Accesses. In *Proc. of the ACM-SIAM Symp. on Discrete Algorithms (SODA)*, 2005.

[187] Johns Paul, Jiong He, and Bingsheng He. GPL: A GPU-based Pipelined Query Processing Engine. In *Proc. of the International Conference on the Management of Data (SIGMOD)*, pages 1935–1950, 2016.

[188] Geoff Pike and Jyrki Alakuijala. Introducing CityHash. http://google-opensource.blogspot.com/2011/04/introducing-cityhash.html. Accessed: 2015-12-01.

[189] Jorda Polo, David Carrera, Yolanda Becerra, Jordi Torres, Eduard Ayguadé, Malgorzata Steinder, and Ian Whalley. Performance-Driven Task Co-scheduling for Mapreduce Environments. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*. IEEE, 2010.

[190] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)*, 2015.

[191] Leo Porter. *Single Threaded Performance in the Multi-Core Era*. PhD thesis, University of California, San Diego, 2011.

[192] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J Ramanujam, and P Sadayappan. Combined Iterative and Model-Driven Optimization in an Automatic Parallelization Framework. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 2010.

[193] Jason Power, Yinan Li, Mark D. Hill, Jignesh M. Patel, and David A. Wood. Toward GPUs Being Mainstream in Analytic Processing: An Initial Argument Using Simple Scan-aggregate Queries. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, pages 11:1–11:8, New York, NY, USA, 2015. ACM.

[194] MVAPICH Team. Mvapich2 1.8 user guide. http://mvapich.cse.ohio-state.edu/support/mvapich2-1.8_user_guide.pdf, 2012.

[195] K.K. Pusukuri, R. Gupta, and L.N. Bhuyan. Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, nov. 2011.

[196] Moinuddin K Qureshi and Yale N Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006.

[197] Arun Raghavan, Laurel Emurian, Lei Shao, Marios Papaefthymiou, Kevin P Pipe, Thomas F Wenisch, and Milo MK Martin. Computational Sprinting on a Hardware/Software Testbed. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 155–166. ACM, 2013.

[198] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M Hellerstein, and Scott Shenker. Prefix Hash Tree: An Indexing Data Structure Over Distributed Hash Tables. In *Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)*, 2004.

[199] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proc. of the VLDB Endowment*, 6(11):1080–1091, 2013.

[200] Rajiv Ranjan. Streaming Big Data Processing in Datacenter Clouds. *IEEE Cloud Computing*, 1(1):78–83, 2014.

[201] Andrea W Richa, M Mitzenmacher, and R Sitaraman. The Power of Two Random Choices: A Survey of Techniques and Results. In Sanguthevar Rajasekaran, Panos M. Pardalos, J.H. Reif, and Josè Rolim, editors, *Handbook of Randomized Computing*, volume 1, chapter 9, pages 255–304. Kluwer Academic Publishers, 2001.

[202] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE Computer Society, 2012.

[203] Kenneth Ross. Efficient Hash Probes on Modern Processors. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, 2007.

[204] Joshua Ruggiero. Measuring Cache and Memory Latency and CPU to Memory Bandwidth. Intel Whitepaper, 2008.

[205] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008.

[206] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1357–1369. ACM, 2015.

[207] Andreas Sandberg, Andreas Sembrant, Erik Hagersten, and David Black-Schaffer. Modeling Performance Variation Due to Cache Sharing. In *The 19th IEEE International Symposium on High Performance Computer Architecture*, pages 155–166, 2013.

[208] Mohamed Sayeed, Hansang Bae, Yili Zheng, Brian Armstrong, Rudolf Eigenmann, and Faisal Saied. Measuring High-Performance Computing with Real Applications. *Computing in Science and Engg.*, 10(4):60–70, July 2008.

[209] Michael J Schulte, Mike Ignatowski, Gabriel H Loh, Bradford M Beckmann, William C Brantley, Sudhanva Gurumurthi, Nuwan Jayasena, Indrani Paul, Steven K Reinhardt, and Gregory Rodgers. Achieving Exascale Capabilities Through Heterogeneous Computing. *IEEE Micro*, 35(4):26–36, 2015.

[210] Ankit Sethia, D Anoushe Jamshidi, and Scott Mahlke. Mascar: Speeding up GPU Warps by Reducing Memory Pitstops. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 174–185. IEEE, 2015.

[211] J. Shalf, S. Dosanjh, and J. Morrison. Exascale Computing Technology Challenges. *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25, 2011.

[212] Bhanu Sharma, Ruppa K Thulasiram, Parimala Thulasiraman, Saurabh K Garg, and Rajkumar Buyya. Pricing Cloud Compute Commodities: A Novel Financial Economic Model. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 2012.

[213] Julian Shun and Guy E Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM SIGPLAN Notices*, volume 48, pages 135–146. ACM, 2013.

[214] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[215] Allan Snavely, Dean M Tullsen, and Geoff Voelker. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30. ACM, 2002.

[216] David C Snowdon, Etienne Le Sueur, Stefan M Petters, and Gernot Heiser. Koala: A Platform for OS-Level Power Management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 289–302. ACM, 2009.

[217] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2005.

[218] Ion Stoica. For Big Data, Moore's Law Means Better Decisions. https://amplab.cs.berkeley.edu/for-big-data-moores-law-means-better-decisions/. Accessed: 2016-08-30.

[219] John E Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[220] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat

O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a Column-Oriented DBMS. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[221] Shawn M Strande, Pietro Cicotti, Robert S Sinkovits, William S Young, Rick Wagner, Mahidhar Tatineni, Eva Hocks, Allan Snavely, and Mike Norman. Gordon: Design, Performance, and Experiences Deploying and Supporting a Data Intensive Supercomputer. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond*. ACM, 2012.

[222] J Stuecheli, Bart Blaner, CR Johns, and MS Siegel. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[223] G Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic Partitioning of Shared Cache Memory. *The Journal of Supercomputing*, 28(1), 2004.

[224] M Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N Patt. MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*. IEEE, 2012.

[225] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, New York, NY, USA, 2011. ACM.

[226] Chao Tian, Haojie Zhou, Yongqiang He, and Li Zha. A Dynamic Mapreduce Scheduler for Heterogeneous Workloads. In *Grid and Cooperative Computing, 2009. GCC'09. Eighth International Conference on*. IEEE, 2009.

[227] M.M. Tikir, M.A. Laurenzano, L. Carrington, and A. Snavely. PSINS: An Open Source Event Tracer and Execution Simulator. In *DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2009*, pages 444 –449, june 2009.

[228] Josh Triplett, Paul E McKenney, and Jonathan Walpole. Scalable Concurrent Hash Tables via Relativistic Programming. *ACM SIGOPS Operating Systems Review*, 44(3):102–109, 2010.

[229] Josh Triplett, Paul E McKenney, and Jonathan Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proc. of the USENIX Annual Technical Conf. (USENIX ATC)*, page 11, 2011.

[230] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A Shah. Analyzing the Energy Efficiency of a Database Server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 231–242. ACM, 2010.

[231] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[232] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. DejaVu: Accelerating Resource Allocation in Virtualized Environments. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2012.

[233] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 205–218. ACM, 2010.

[234] Gregor Von Laszewski, Lizhe Wang, Andrew J Younge, and Xi He. Power-Aware Scheduling of Virtual Machines in DVFS-Enabled Clusters. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[235] M Mitchell Waldrop. The Chips are Down for Moore's Law. *Nature News*, 530(7589):144, 2016.

[236] Hongyi Wang, Qingfeng Jing, Rishan Chen, Bingsheng He, Zhengping Qian, and Lidong Zhou. Distributed Systems Meet Economics: Pricing in the Cloud. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. USENIX Association, 2010.

[237] J. Weinberg and A. Snavely. Symbiotic Space-Sharing on SDSC's Datastar System. In *12th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2006.

[238] J. Weinberg and A. Snavely. User-Guided Symbiotic Space-Sharing of Real Workloads. In *20th ACM International Conference on Supercomputing*, June 2006.

[239] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for Reduced CPU Energy. In *Mobile Computing*, pages 449–471. Springer, 1994.

[240] Barry Wilkinson and Michael Allen. *Parallel Programming*, volume 999. Prentice hall New Jersey, 1999.

[241] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[242] Y. Wiseman and D.G. Feitelson. Paired Gang Scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 14(6), June 2003.

[243] Pfeiffer Wright and Snavely. Characterizing Parallel Scaling of Scientific Applications Using IPM. In *The 10th LCI International Conference on High-Performance Clustered Computing*, March 2009.

[244] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118. IEEE Computer Society, 2012.

[245] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 44. ACM, 2014.

[246] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proc. of the USENIX Annual Technical Conf. (USENIX ATC)*, 2015.

[247] Wm A Wulf and Sally A McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[248] Yuejian Xie and Gabriel H. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *CMP-MSI*, 2005.

[249] Chi Xu, Xi Chen, Robert P Dick, and Zhuoqing Morley Mao. Cache Contention and Application Performance Prediction for Multi-Core Systems. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010.

[250] Zichen Xu, Yi-Cheng Tu, and Xiaorui Wang. Exploring Power-Performance Tradeoffs in Database Systems. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 485–496. IEEE, 2010.

[251] Zichen Xu, Xiaorui Wang, and Yi-cheng Tu. Power-Aware Throughput Control for Database Management Systems. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 315–324, 2013.

[252] Jun Yan and Wei Zhang. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2008.

[253] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *International Symposium on Computer Architecture*, 2013.

[254] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 10:10–10, 2010.

[255] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-KV: A case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. *Proc. of the VLDB Endowment*, 8(11):1226–1237, 2015.

[256] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-Aware Graph-Structured Analytics. In *ACM SIGPLAN Notices*, volume 50, pages 183–193. ACM, 2015.