Techniques to Improve Dynamic Cache Management with Static Data Classification

by

Anurag Mukkara

B.Tech. in Electrical Engineering
Indian Institute of Technology Bombay, 2014

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

© Massachusetts Institute of Technology 2016. All rights reserved.

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by . . . . . .
Daniel Sanchez
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Techniques to Improve Dynamic Cache Management with Static Data Classification

by

Anurag Mukkara

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Cache hierarchies are increasingly non-uniform and difficult to manage. Several techniques, such as scratchpads or reuse hints, use static information about how programs access data to manage the memory hierarchy. Static techniques are effective on regular programs, but because they set fixed policies, they are vulnerable to changes in program behavior or available cache space. Instead, most systems rely on dynamic caching policies that adapt to observed program behavior. Unfortunately, dynamic policies spend significant resources trying to learn how programs use memory, and yet they often perform worse than a static policy.

This thesis presents Whirlpool, a novel approach that combines static information with dynamic policies to reap the benefits of each. Whirlpool statically classifies data into *pools* based on how the program uses memory. Whirlpool then uses dynamic policies to tune the cache to each pool. Hence, rather than setting policies statically, Whirlpool uses static analysis to guide dynamic policies. Whirlpool provides both an API that lets programmers specify pools manually and a profiling tool that discovers pools automatically in unmodified binaries.

On a state-of-the-art NUCA cache, Whirlpool significantly outperforms prior approaches: on sequential programs, Whirlpool improves performance by up to 38% and reduces data movement energy by up to 53%; on parallel programs, Whirlpool improves performance by up to 67% and reduces data movement energy by up to 2.6×.

Thesis Supervisor: Daniel Sanchez
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

Firstly, I would like to thank my advisor Daniel Sanchez, without whose guidance and support this thesis would not have been possible. It has been a great experience working with Daniel for the past two years. The work in this thesis is the first major research project I was involved in. Daniel helped me pick up pace and guided me towards solving challenging problems while staying within the boundaries of my broad research interests. There were several instances where I was going off track and focusing on the wrong things. Daniel was quick to spot them and helped me overcome those minor blips. He is also a great inspiration to me and his love for research in the broad field of computer systems helps replenish my motivation levels from time to time.

I would like to thank my groupmate Nathan Beckmann, who has been a virtual coadvisor to me. It was very helpful to get a second opinion and guidance from a senior graduate student like Nathan. My thesis builds on Nathan's earlier work and I had the good fortune to collaborate with Nathan on this project. In particular, Nathan has helped a lot with writing this thesis and the paper it is based on. Quite often I made several 'dumb' mistakes that only a young graduate student like me can make. Both Daniel and Nathan were extremely patient with me in such instances.

It has been great fun discussing ideas, socializing, getting help and feedback from members of Daniel's group. In particular, Nathan, Harshad and Guowei (the cool kids of 7th floor) made me eat a lot of good food at very unusual times. Thanks to my friends Nishant, Anil, Sharath and Prudhvi, for helping me maintain my sanity and relax when needed.

A special thanks to Mrinal, my ex-girlfriend and best friend. I was going through a phase of depression when I started my graduate studies at MIT. Mrinal has been a tremendous support and helped me overcome that phase. Whenever I was feeling low, she cheered me up and made me see the brighter side of things. Without her support, it would have been very difficult for me to focus on my research.

Finally, thanks to my parents for allowing me to follow my passion, even if that meant staying away from their only son for long periods of time. They always gave me a lot of freedom and let me do what I think is the best for me.

Contents

1 Introduction **13**

1.1 Contributions 14

1.2 Thesis structure 15

2 Motivation **17**

2.1 Static classification 17

2.2 Dynamic policies 19

3 Related Work **21**

3.1 Software-assisted techniques 21

3.2 Hardware techniques 22

4 Baseline and Methodology **25**

4.1 Baseline Architecture 25

4.1.1 Virtual caches 25

4.1.2 Single-lookup accesses 25

4.1.3 Reconfigurations 26

4.2 Experimental methodology 28

5 Manual Classification **31**

5.1 Application programming interface 31

5.2 Modifications to baseline system 32

5.3 Sequential applications 34

5.4 Parallel applications 37

6 Automated Data Classification **41**

6.1 Profiler 42

6.2 Analyzer 42

6.3 Runtime 46

6.4 Analysis 46

6.5 Evaluation 48

7 Conclusion **53**

List of Figures

2-1 Multicore chip with a distributed last-level cache. 18

2-2 Breakdown of `dt`'s working set and access pattern. 18

2-3 Placement of different data structures of `dt` with S-NUCA, Jigsaw and Whirlpool 18

2-4 `lbm` has two pools that, though indistinguishable on average, have markedly different access patterns in alternating program phases. . . 19

4-1 Overview of Jigsaw, our baseline NUCA system. 26

4-2 `dt`'s memory performance vs. VC size: (a) Last-level cache misses. (b) Cycles per instruction stalled on data. 27

5-1 `mis`'s memory performance vs. VC size. Vertices cache well, but edges are streaming. Whirlpool bypasses edges and gives the cache to vertices. 34

5-2 Breakdown of `mis`'s performance, energy, and accesses for different caching schemes. 35

5-3 `refine` has irregular phase changes. Whirlpool dynamically adapts its allocations and placement to retain the data structures that have reuse. 36

5-4 Partitioned work-stealing (PaWS) in Whirlpool on a 16-core system. In PaWS, each core works on a partition of the input and preferentially steals tasks from nearby cores. Colors indicate affinity between tasks and data. 38

5-5 Performance of S-NUCA, Jigsaw, Jigsaw with PaWS, and Whirlpool with PaWS on parallel applications. 39

6-1 WhirlTool overview. 41

6-2 WhirlTool measures the distance between two pools as the additional misses incurred by combining the pools vs. partitioning them separately. 43
6-3 WhirlTool's simple model to combine miss rate curves. 44
6-4 Hierarchical clustering with WhirlTool. Each graph shows the distance (x-axis) among callpoints and clusters (y-axis). Colors indicate how WhirlTool clusters callpoints into 3 pools. 45
6-5 Speedup of WhirlTool over Jigsaw with 2, 3, and 4 pools. A red dot shows the performance achieved by manual classification for the applications we ported by hand (Table 5.1). 47
6-6 WhirlTool's sensitivity to training inputs. 48
6-7 Breakdown of `cactus`'s performance, energy, and accesses for different caching schemes. 48
6-8 Breakdown of `SA`'s performance, energy, and accesses for different caching schemes. 49
6-9 Breakdown of overall performance, energy, and accesses for different caching schemes across all single-threaded benchmarks. 50
6-10 Weighted speedup of Whirlpool over Jigsaw for 4- and 16-core systems. 51

List of Tables

3.1 Desirable properties achieved by prior memory system management techniques. 21

4.1 Configuration of the simulated 16-core CMP. 28

5.1 Pools found manually in various applications, plus lines of code (LOC) modified while porting to Whirlpool. 33

Chapter 1

Introduction

Future systems will be limited by data movement, which is orders of magnitude more expensive than basic compute operations. For example, at 28 nm a 64-bit floating-point multiply-add consumes 20 pJ, while sending 256 bits across the chip costs 300 pJ, an on-chip access to a 1 MB cache costs about 1 nJ, and an off-chip DRAM access costs 20-50 nJ—1000× more energy than the multiply-add [19, 33, 55]. The trend towards lean, specialized cores means that, for efficiency reasons, caches are increasingly distributed across the chip and have non-uniform access latencies (NUCA [34]).

While distributed caches are more efficient, they are also harder to manage. Their non-uniform latency and energy means that *data placement* is critical to limit data movement. But data placement is hard: applications need to fit their most intensely-used data in nearby banks, while competing with each other for scarce capacity. Data placement is a spatial scheduling problem that, to solve well, requires accurate information about how programs use memory.

Unfortunately, all the relevant information is not generally available: static analysis or profiling can reveal program semantics (i.e., how a program uses memory), but not its dynamic or input-dependent behavior; and dynamic policies have difficulty efficiently recovering program semantics. To see this in more detail, consider the extremes of static vs. dynamic design. At one extreme, scratchpad-based systems expose the distributed memories to software, relying on static analysis to place data. Scratchpads work well on regular access patterns, but cope poorly with irregular, input-dependent, or rapidly changing patterns and varying resources in shared systems [35, 38]. At

the other extreme, cache-based systems expose a flat address space that programs access through undifferentiated loads and stores, relying on hardware-managed caches to transparently retain the right data. Most memory systems are cache-based, but recovering program semantics from this limited interface is difficult and expensive. For example, classic dynamic NUCA schemes migrate data towards the requester in response to each access, which increases data movement and requires expensive lookups [5, 7, 25]. As memory systems become more complex, ignoring program semantics becomes increasingly inefficient.

Prior work exploits static information in cache-based systems through prefetch [31], bypass [45], and cache priority [24] hints. Hints let software override dynamic policies and control the cache, reaping the benefits of static information when it is accurate. However, hints suffer from the same problems as scratchpads: with uncertain or dynamic behavior, hints are often inaccurate and hurt performance [38, 44].

1.1 Contributions

The key idea of this thesis is to combine static information with dynamic policies to reap the benefits of each. Rather than using static information to set fixed policies, we instead use it to inform dynamic policies. The insight is that, while uncertainty makes it hard to statically predict how data will be used, it is often easy to accurately group data with similar usage patterns. This approach lets dynamic policies make better decisions at lower overhead.

We demonstrate this idea through Whirlpool, a *classification-based* approach to improve data placement in multicores. In Whirlpool, programs divide their data into a small number of memory *pools*, e.g., one for each major data structure. We find that for most programs, a few pools (three or four) suffice. Hardware then monitors each pool dynamically and adapts the memory system to keep the most valuable data near where it is used. Unlike hints, pools do not encode static policies; rather, they make it easy for hardware to find the right policies dynamically. Whirlpool thus combines static program semantics with dynamic policies, and robustly adapts to changes in program behavior or available resources.

Whirlpool has both software and hardware components. In software, Whirlpool

provides a memory allocator that groups semantically similar data and tags each page with a pool id. In hardware, Whirlpool extends prior NUCA techniques [7, 9] to monitor each pool and control its placement. Whirlpool needs only a few pools, so it adds small overheads. In summary, Whirlpool gives a promising way to combine static program semantics and dynamic policies to reduce data movement.

We first present an API that lets a programmer manually classify data structures into pools to reap the benefits of Whirlpool. We show that this achieves significant performance gains, of up to 38%, when managing a large NUCA cache, and reduces data movement energy by up to 53%. We use the insights gained from manually porting applications to design *WhirlTool*, a profiling tool that automatically discovers pools in unmodified binaries. We evaluate WhirlTool on a comprehensive set of benchmarks and program mixes, and find that it works as well or better than our careful manual classification. We also show how Whirlpool improves the performance of parallel applications on a 16-core chip by up to 67% and reduces data movement energy by up to 2.6×.

1.2 Thesis structure

This thesis is structured as follows: Chapter 2 motivates the need to combine static classification with dynamic policies. Chapter 3 discusses related work. Chapter 4 describes our baseline architecture and experimental methodology. Chapter 5 introduces Whirlpool's manual classification approach, providing case studies on both sequential and parallel workloads. Chapter 6 discusses WhirlTool, the automated classification tool that extends Whirlpool. Chapter 7 concludes the thesis.

Chapter 2

Motivation

In this chapter, we will discuss the benefits and shortcomings of static and dynamic techniques and motivate why we need a policy that exploits both these techniques, while avoiding the shortcomings of either.

2.1 Static classification

Consider the multicore shown in Figure 2-1. This chip has a NUCA cache of twenty-five 512 KB banks shared by four surrounding cores, similar to the Oracle SPARC M7 [1] (see Section 4.2 for detailed methodology). We consider the benchmark `dt` (Delaunay triangulation) from the PBBS suite [56], running in the leftmost core. Our goal is to use this distributed cache capacity as efficiently as possible by placing `dt`'s most intensely used data near where `dt` is running.

Figure 2-2 shows how `dt` accesses memory. It has a 6 MB working set that easily fits in the cache. It accesses three data structures: **points**, **vertices**, and **triangles**, which take 0.5 MB, 1.5 MB, and 4 MB, respectively. Accesses are split roughly evenly across the three data structures, so their access *intensity* (i.e., accesses per MB) varies: **points** are accessed most intensely, followed by **vertices** and **triangles**.

How should we place `dt`'s data to reduce data movement? Many commercial processors adopt a static NUCA (S-NUCA) design that hashes addresses evenly across banks [34, 50]. Figure 2-3a shows how S-NUCA places `dt`'s data. This figure shows all 25 cache banks, with colors indicating where data is placed. Because S-NUCA spreads

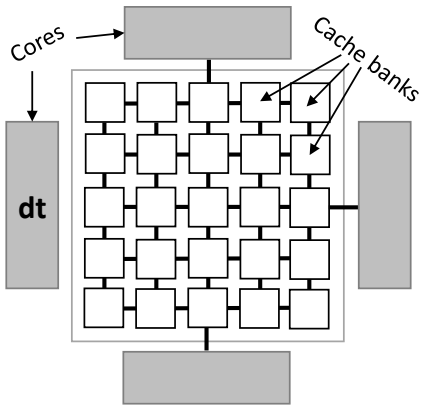


Figure 2-1: Multicore chip with a distributed last-level cache.

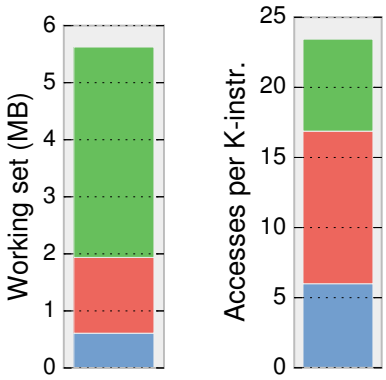
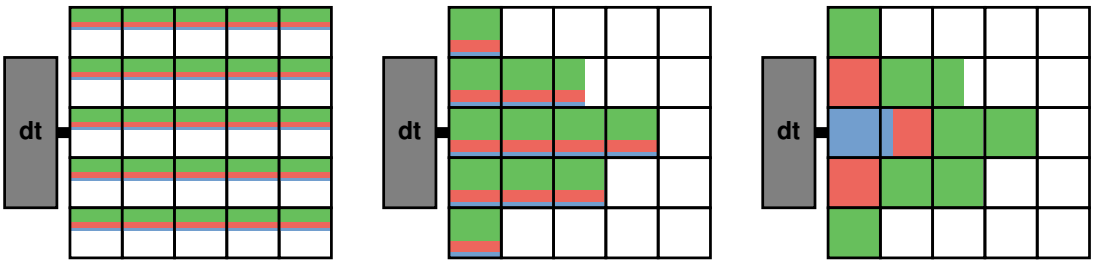


Figure 2-2: Breakdown of `dt`'s working set and access pattern.

`dt`'s 6 MB working set across 12.5 MB of cache, banks are left half-empty.

Since capacity is available in closer banks, we can reduce data movement by concentrating `dt`'s data closer to where `dt` is running. Figure 2-3b shows how Jigsaw [7, 9], the NUCA scheme that Whirlpool builds on, places `dt`'s data. Jigsaw tightly packs `dt`'s working set near the left of the chip, but it cannot distinguish between the different data structures because it is blind to program semantics.

We can reduce data movement further by placing the most intensely accessed data even closer to where `dt` is running. Figure 2-3c shows how Whirlpool places `dt`'s data. Whirlpool classifies data into pools statically, but dynamically monitors each pool to decide data placement. In this case, because **points** is most intensely accessed, it is placed in the closest cache banks. Likewise, **vertices** is placed in three next-closest banks, and **triangles** in the closest remaining banks.



(a) S-NUCA (b) Jigsaw (c) Whirlpool

Figure 2-3: Placement of different data structures of `dt` with S-NUCA, Jigsaw and Whirlpool

Whirlpool improves `dt`'s performance by 19% over S-NUCA and by 15% over

Jigsaw, and reduces data movement energy (cache and memory dynamic energy) by 42% over S-NUCA and by 27% over Jigsaw.

This example shows how static, program-level data classification can help dynamic policies reduce data movement. By identifying key data structures statically, Whirlpool can place data without wasting resources in learning how data is used. By contrast, many prior schemes that achieve a similar data placement do so by migrating data on demand [16, 40, 64]. The extra data movement from migrations can exceed the energy savings of smart data placement (see Chapter 2). Hence, static classification is crucial to Whirlpool's benefits.

2.2 Dynamic policies

Whirlpool leverages static data classification from either the programmer or an automatic profile-guided tool. Whirlpool also monitors each pool at run-time and uses this dynamic information to reconfigure the cache. By decoupling classification and policy, Whirlpool robustly adapts to changes in application behavior and system configuration. By contrast, other techniques that involve application-level or code changes, such as software prefetching [44], reuse/non-temporal hints [10, 12, 24, 45, 59], or loop tiling [18, 36, 60], directly encode a fixed policy in the program and cannot adapt to changing application or system behavior.

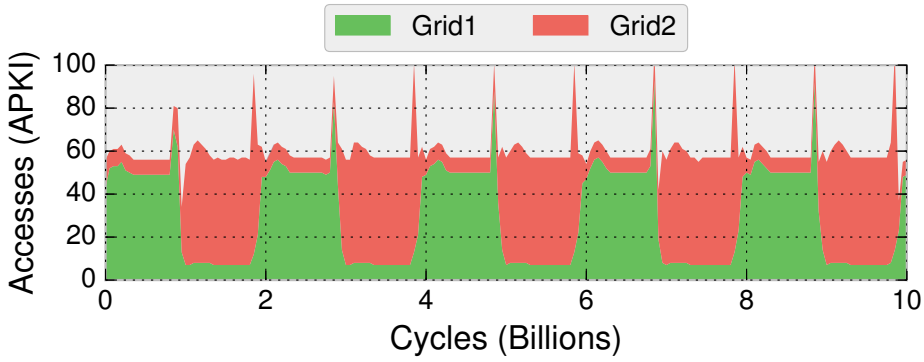


Figure 2-4: `lbm` has two pools that, though indistinguishable on average, have markedly different access patterns in alternating program phases.

To see why adapting dynamically is important, consider `lbm` from SPEC CPU2006. On each timestep, `lbm` operates on two grids, source and destination, with markedly

different access patterns: source is accessed more often and enjoys good reuse, while destination sees little reuse. At the end of each timestep, lbm swaps the source and destination pointers, resulting in an alternating access pattern to both memory pools, shown in Figure 2-4. Whirlpool continuously monitors both pools and adopts different policies on even and odd phases, caching source data near lbm and bypassing accesses to the destination grid. As a result Whirlpool outperforms Jigsaw by 4.8%, and reduces data movement energy by 12%. By contrast, each pool looks the same on average, so the best static placement yields no improvement over Jigsaw.

This example shows that while static *classification* into pools may suffice, dynamic *policies* are needed to reduce lbm's data movement. Moreover, program phase changes are not the only source of unpredictability. The appropriate caching policy depends on many factors that are hard to capture statically: irregular reference patterns, different program inputs, cache contention in shared systems, etc. Chapter 5 explores how Whirlpool's dynamic policies adapt to these sources of dynamic variability.

Chapter 3

Related Work

Whirlpool builds upon many prior memory management techniques. Broadly, these techniques fall into two categories, based on whether they consider modifying the program to improve memory system behavior. Table 3.1 summarizes the pros and cons of each technique.

Scheme	Static information	Dynamic policy	Spatial placement	Single-lookup	Easy to use
Scratchpads	✓	✗	✓	✓	✗
Code hints	✓	✗	✗	✓	✓
Cache replacement	✗	✓	✗	✓	✓
Private D-NUCA	✗	✓	✓	✗	✓
Shared D-NUCA	✗	✓	✓	✓	✓
Whirlpool	✓	✓	✓	✓	✓

Table 3.1: Desirable properties achieved by prior memory system management techniques.

3.1 Software-assisted techniques

Architectures with software-managed scratchpad memories provide the highest efficiency and degree of control, letting programmers or compilers manage data placement and movement. But scratchpads are hard to use. Even with advanced compilers, only highly regular programs can use them well [38]. Scratchpads also do not support dynamic adaptation.

For these reasons, most memory systems use cache hierarchies instead. Caches are transparent by default, but programs can include various types of access hints such as software prefetch instructions and non-temporal hints to bypass the cache hierarchy [10, 12, 24, 45, 59]. However, these techniques embed specific static policies in the program, which may degrade performance.

The common drawback of these techniques is their lack of dynamism, which makes these optimizations risky: changes in workload behavior or system configuration may make these optimizations ineffective or detrimental to performance.

3.2 Hardware techniques

High-performance replacement policies often dynamically classify data and treat lines of each class differently. For example, RRIP [30] classifies lines as reused and non-reused; IbRDP [47] classifies them by the PC of their last memory access; and SHiP [61] classifies lines by PC or memory address.

Similarly, Whirlpool's classification could also improve replacement decisions. We explored a Whirlpool-based replacement policy that extends DRRIP [30] to adapt insertion priority across pools (similar to TA-DRRIP [29, 30] and CAMP [46]). However, we found that the benefits from static classification within a monolithic cache were marginal: cache replacement is a simpler problem than data placement, and dynamic replacement policies like DRRIP and SHiP perform well at relatively low overhead. We therefore focus on the harder problem of NUCA data placement.

Dynamic NUCAs (D-NUCAs) try to reduce data movement by placing data near where it is used. These schemes can be broadly classified into two categories based on whether they start from a private or shared cache organization.

Private-baseline D-NUCAs treat the cache as a fine-grained hierarchy, accessing the closest banks first, then checking farther-away banks on a miss [4, 34, 64]. Upon a hit, these D-NUCAs move data closer to the accessing core, displacing other data further away, similar to how high-performance replacement policies promote lines upon reuse. Hence, over time, private-baseline D-NUCAs gradually place frequently-used data nearby.

However, private-baseline D-NUCAs suffer from two problems. First, migrating

data in response to each access increases overall data movement and wastes significant energy [5]. Second, since addresses do not have a known location, they also require complex lookup mechanisms (e.g., multi-level lookups, broadcasts, or directories) that add area, latency, and energy [4, 48, 64]. For example, schemes that use global directory lookups beyond the local cache bank would not reduce `dt`'s data movement for **vertices** or **triangles**—the majority of its data accesses. As a result, prior work has consistently shown that it is far more efficient to avoid moving data among banks in response to accesses [5, 7, 20, 25, 48].

Shared-baseline D-NUCAs [2, 7, 17, 25] leverage the virtual memory system to control data placement. A page's location is tracked in software and infrequently updated in response to program behavior. Unlike private-baseline D-NUCAs, shared-baseline D-NUCAs can locate data in a single lookup and thereby avoid excessive data migration. However, they respond more slowly to changes in program behavior and must also place data at a page granularity.

Whirlpool achieves the advantages of the above techniques while minimizing their drawbacks. It leverages static, program-level data classification to achieve a good data placement without frequently migrating data, and it adapts to unpredictable run-time variability.

Chapter 4

Baseline and Methodology

4.1 Baseline Architecture

Whirlpool builds on Jigsaw, a partitioned, shared-baseline D-NUCA. We now briefly describe Jigsaw's main features; please see prior work for details [6, 7, 9].

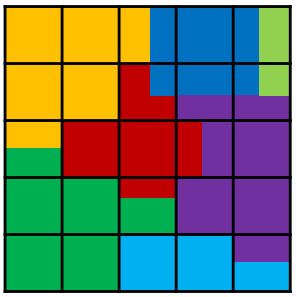
4.1.1 Virtual caches

Jigsaw builds *virtual caches* (VCs) by combining partitions of physical cache banks, as shown in Figure 4-1a (colors represent VCs). Pages are mapped to a specific VC through the TLB. Jigsaw uses three kinds of VCs: each thread has a thread-private VC; all threads in the process share a process VC; and all threads in the system share a global VC. Pages start as private to the thread that allocates them, and are upgraded lazily: an access from another thread upgrades the page to the process VC, and an access from another process upgrades the page to the global VC.

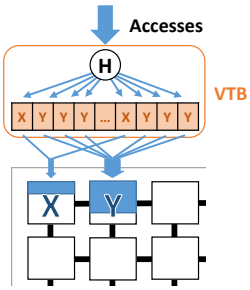
To support Whirlpool, we extend Jigsaw to allow applications to define additional VCs and map pages to them.

4.1.2 Single-lookup accesses

Jigsaw stores the placement of each VC in a small structure called the *virtual cache translation buffer* (VTB). In Jigsaw, each core requires just 3 VTB entries (for its thread-private, process, and global VCs). Each VTB entry is essentially a configurable



(a) Jigsaw groups bank partitions into virtual caches (VCs).



(b) The VTB controls data placement across banks.

Figure 4-1: Overview of Jigsaw, our baseline NUCA system.

hash function that maps an address to its *unique location*—in Jigsaw, data does *not* migrate in response to accesses. Jigsaw thus provides single-lookup accesses. The VTB controls data placement by dividing access stream across banks, as in Figure 4-1b.

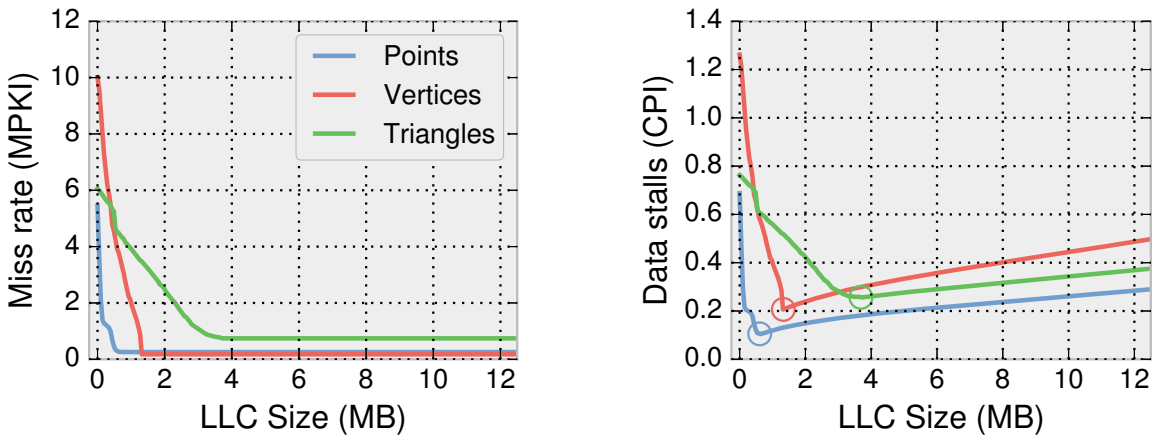
4.1.3 Reconfigurations

A lightweight OS runtime periodically (every 25 ms in our implementation) reconfigures the cache, sizing and placing VCs to minimize data movement. It does so using a simple model of memory access time that accounts for both cache misses and cache access latency. To account for cache misses, Jigsaw monitors *miss rate curves*, i.e. miss rate vs. VC size [9, 49]. To account for cache access latency, Jigsaw uses the average latency to the closest cache banks needed for a given VC size. From these components, the total latency of a VC is simply the sum of VC access latency (access rate × network and bank latency) and memory latency (miss rate × miss penalty).[1]

The runtime builds the total latency curves for each VC and uses them to partition cache capacity. Traditional cache partitioning schemes try to minimize cache misses and partition using miss rate curves [49, 51]. By using latency curves instead of miss rate curves [9], Jigsaw will not use cache banks when their reduction in miss rate does not offset their added network latency. For example, `dt` fits in half the cache banks, so the remaining banks are not used (Figure 2-3b).

Whirlpool chooses VC sizes identically to Jigsaw, with the only difference being that each memory pool gets its own VC. Figure 4-2a shows the miss rate curves for

[1]This simple model ignores memory-level parallelism, but we find it works well. Alternatively, we could model energy; this would penalize misses more and change tradeoffs somewhat.



(a) Miss rate curves.

(b) Memory latency curves.

Figure 4-2: `dt`'s memory performance vs. VC size: (a) Last-level cache misses. (b) Cycles per instruction stalled on data.

`dt`, and Figure 4-2b shows the latency curves and the sizes chosen for each VC: in this case, the full working set fits on cache, so Jigsaw chooses the sizes that minimize each VC's total latency.

After sizing VCs, the reconfiguration algorithm places them in cache banks. We use Jigsaw's trading placement algorithm [9], which initially places data using a simple, greedy heuristic, and then trades capacity between VCs to reduce data movement. The key idea is *access intensity*: lines that are accessed more frequently pay a larger penalty for poor placement, and should therefore be placed closer to cores that access them. Intensity is given as access rate per capacity, i.e. a VC's access rate divided by its size (APKI per MB). For example, the pools in `dt` are accessed at a similar rate, but since **points** is one-eighth the size of **triangles**, its access intensity is 8× larger. Intensity essentially says how many accesses are affected by placing a chunk of capacity of fixed size, and it lets us compute if trading capacity between two VCs reduces data movement.

Jigsaw outperforms state-of-the-art D-NUCAs and adds small overheads [7, 9]. In hardware, Jigsaw adds less than 0.6% area overhead over LLC banks; in software, Jigsaw consumes less than 0.4% of system cycles. Whirlpool extends Jigsaw to support static classification of data into pools by building VCs for each pool. We make small modifications to Jigsaw to exploit opportunities presented by static classification, but do not modify its core hardware mechanisms or software reconfiguration runtime.

Cores	4/16 cores, x86-64 ISA, Nehalem-like OOO, 2 GHz
L1 caches	32 KB, 8-way set-associative, split D/I, 4-cycle latency
L2 caches	128 KB private per-core, 8-way set-associative, inclusive, 6-cycle latency
L3 cache	512 KB per bank, 4-way 52-candidate zcache, 9 cycle bank latency
Coherence protocol	MESI, 64 B lines, in-cache directory, no silent drops; sequential consistency
NUCA NoC	5×5/9×9 mesh, 128-bit flits and links, X-Y routing, 3-cycle pipelined routers, 2-cycle links
Memory	1/4 MCUs, 1 channel/MCU, 120 cycles zero-load latency, 12.8 GB/s per channel

Table 4.1: Configuration of the simulated 16-core CMP.

4.2 Experimental methodology

We perform microarchitectural, execution-driven simulation using zsim [52]. We simulate systems with 4 and 16 OOO cores with parameters in Table 4.1. The 4-core system has a NUCA cache with 5×5 with 512 KB banks (3.1 MB/core), as shown in Figure 2-1. The 16-core system has 9×9 banks (2.5 MB/core), as shown in Figure 5-4. We compute data movement (uncore) energy using McPAT 1.1 [39] at 22 nm for caches and NoC, and Micron DDR3L datasheets [43] for main memory. Additionally, we evaluated systems with stream prefetchers: Whirlpool's performance relative to other schemes is unchanged. We do not include prefetchers because they add undesirable data movement energy, especially in mixes.

We compare Whirlpool with D-NUCA and S-NUCA configurations. Jigsaw and Whirlpool both use latency-aware capacity allocation and trading data placement [9]. For private-baseline D-NUCAs, we model an idealized shared-private D-NUCA scheme, IdealSPD, which we grant additional capacity. In IdealSPD, each core has a private 1.5 MB L3 that replicates the 3 closest NUCA banks, followed by a fully-provisioned directory and an exclusive, S-NUCA L4. L4 banks act as a victim cache and are accessed in parallel with the directory to minimize latency. IdealSPD upper-bounds D-NUCA schemes that partition the LLC between private and shared regions, as private (L3) regions do not reduce the capacity of the shared (L4) region. Herrero et al. [27] show that this idealized scheme *always* outperforms several state-of-the-art

private-baseline D-NUCA schemes that include shared-private partitioning, selective replication, and adaptive spilling (DCC [26], ASR [4], and ECC [27]), often significantly (up to 30%).

For shared-baseline D-NUCAs, we compare against a D-NUCA scheme proposed by Awasthi et al. [2] that uses page coloring to periodically migrate a few most heavily accessed pages to nearby banks. The scheme uses simple hardware extensions and an OS runtime, similar to Whirlpool. Because Awasthi manages individual pages, it doesn't require tagging pools. But per-page monitoring also limits the information Awasthi can gather, and it therefore places pages incrementally using a simple heuristic that can get stuck in local optima (see Figure 5-1). In contrast, Whirlpool monitors pools in detail, models end-to-end latency, and performs full reconfigurations, achieving lower AMAT. We have implemented Awasthi as proposed, sweeping implementation parameters α_A, α_B to find the values that perform best. Other shared-baseline D-NUCAs use placement heuristics that compare unfavorably to Awasthi and Whirlpool; e.g., R-NUCA [25] achieves 6.8%/7.2% lower performance than Awasthi on 4-/16-core mixes of SPEC CPU2006.

We use SPEC CPU2006 and PBBS [56] apps. In single-program experiments, SPEC apps are executed for 10 B instructions after fast-forwarding 20 B instructions, and PBBS apps are fast-forwarded to the start of their region of interest, and run for the full region. We consider the applications with >5 L2 MPKI: 15 from SPEC CPU2006 (`bzip2`, `gcc`, `mcf`, `milc`, `zeusmp`, `cactusADM`, `leslie3d`, `soplex`, `GemsFDTD`, `libquantum`, `lbm`, `astar`, `omnetpp`, `sphinx3`, and `xalancbmk`) and 16 from PBBS (all but `nbody`).

We also simulate mixes of single-threaded SPEC CPU2006 apps, using a fixed-work methodology similar to prior work [7, 28, 30]: we run random mixes with 1 B instructions per app after fast-forwarding for 20 B instructions. All apps are kept running until all finish 1 B instructions, and we only consider the first 1 B instructions of each app.

Chapter 5

Manual Classification

We now present the design of Whirlpool and explore how it reduces data movement by combining static, program-level classification with dynamic caching policies.

Whirlpool classifies data used by an application into different regions, which we call memory pools. Memory pools let Whirlpool manage data that has similar access patterns as a single entity. It is also the granularity at which Whirlpool gathers information to drive dynamic policies.

In this chapter, we present an interface that lets an application programmer create memory pools and tag data used by the application to different pools. We then explore how Whirlpool improves placement through case studies. In Chapter 6, we present a profiling framework that automatically classifies data into memory pools.

5.1 Application programming interface

In our implementation, a pool is an independent region of heap-allocated memory. Our memory allocator ensures that pages belong to exactly one pool (or none) at a time, so that we can use the virtual memory system to classify data into pools, as in Jigsaw.

The programmer creates a new memory pool by calling:

```
pool_t pool_create();
```

which returns an id for the newly created pool. To allocate `size` bytes of memory from the pool, the programmer calls:

```
void* pool_malloc(size_t size, pool_t pool_id);
```

Similarly, other variants like `pool_calloc`, `pool_realloc`, etc. augment the standard arguments with `pool_id`.

This API lets the programmer tune the application's cache performance by providing high-level hints at memory allocation time. At first, it might seem to be a tedious task for the programmer to reason about the access patterns and cache locality of different data. However, we find it often suffices to identify a few prominent data regions and allocate them to different pools. Thus, only a few lines of code need to be modified to port applications manually to Whirlpool.

Table 5.1 shows the applications we have manually ported, their key data structures, and the lines of code changed. Overall, Whirlpool improves performance on these applications by 7.3% over Jigsaw and reduces data movement energy by 12%. Detailed results are presented in Chapter 6.

5.2 Modifications to baseline system

System calls to manage VCs: We expose VCs to user-level programs with a few additional system calls: `sys_vc_alloc` allocates a user-level VC, returning its unique id; `sys_vc_free` deallocates an existing VC; and `sys_vc_tag` tags a range of pages with a user-level VC. We also modify `sys_mmap` to optionally tag new pages with a specific VC. These system calls perform the adequate checks to ensure safety (e.g., allowing each process to map pages only to its own user-level VCs).

Our allocator uses this low-level interface to map each pool to a different VC and tag pages from each pool with the right VC id. Our implementation is built on top of Doug Lea's malloc [37], but other allocators could be used instead.

Support for more VCs per core: The baseline Jigsaw system supports 3 VTB entries per core for thread-private, process, and global VCs. To support user-level VCs, we add extra VTB entries and utility monitors (specifically, GMONs [9]). As we will see, supporting up to 4 pools is enough for most programs. In the 4-core system, Whirlpool adds 6 KB in VTB entries and 24 KB of monitors, or 0.3% of cache area.

Application	Pools	Data structures	LOC
Breadth-first search	4	Vertices, edges, frontier, visited	16
Delaunay triangulation	3	Points, vertices, triangles	11
Maximal matching	3	Vertices, edges, result	13
Delaunay refinement	3	Vertices, triangles, misc	8
Maximal independent set	3	Vertices, edges, flags	13
Spanning forest	3	Union-find parents, output tree, input edges	13
Minimal spanning forest	3	Union-find parents, output tree, input edges	11
Convex hull	2	Points, hull array	10
401.bzip2	4	arr1, arr2, ftab, tt	43
470.lbm	2	Source and destination grids	21
429.mcf	2	Nodes and arcs	14
436.cactusADM	2	Pugh variables, staggered-leapfrog grid data	53

Table 5.1: Pools found manually in various applications, plus lines of code (LOC) modified while porting to Whirlpool.

Bypassing VCs: Programs often have data structures that get negligible reuse in the cache, and it is more efficient for them to bypass the cache entirely [58]. Bypassing is particularly beneficial in Whirlpool, since its static classification helps accurately identify data that should be bypassed.

We therefore extend Jigsaw to support bypassing VCs. We add a bypass bit to each VTB entry. Bypassed VCs have no LLC space allocated, and their L2 misses go directly to main memory. Bypassing is allowed only if the VC is accessed by a single thread. Coherence is maintained by invalidating the VC in the LLC when it enters bypassing mode (extending Jigsaw's existing reconfiguration mechanism [9]), and by invalidating the VC in the L2 when it exits bypassing mode. Finally, Jigsaw's software runtime decides whether to bypass a VC by modifying the inputs to its existing partitioning algorithm. Specifically, it excludes cache access latency in its access latency model if the VC is allocated no space. With this trivial change, the partitioning algorithm will

only allocate space to a VC when bypassing hurts performance (see below).

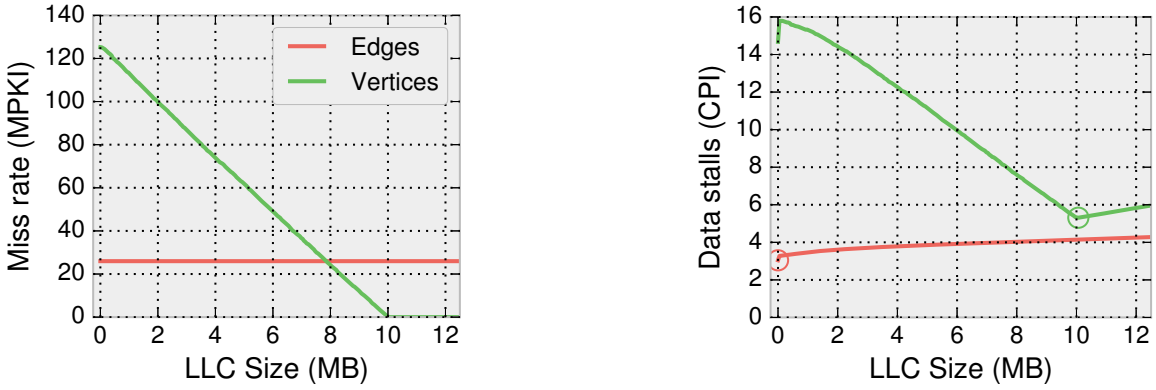
We evaluate both Jigsaw and Whirlpool with this optimization, but since Jigsaw does not separate data that gets reuse from data that does not, we find that VC bypassing is more useful in Whirlpool.

Other Jigsaw components are unmodified. In particular, the OS remains in charge of reconfiguring the cache, and the reconfiguration algorithm stays the same. Additional VCs add 0.2% of system cycles to reconfigurations.

5.3 Sequential applications

We now present several case studies using sequential applications that show how Whirlpool adapts to various sources of variability.

Whirlpool benefits from bypassing: Whirlpool decides whether to bypass VCs by modifying the latency curve. Figure 5-1 shows how this is done for the PBBS benchmark `mis` (maximal independent set). Whirlpool changes Jigsaw's memory latency model to model bypassing for VCs accessed by a single thread. Specifically, the total latency curve at a VC size of zero excludes the cache access latency. This is the only change needed, as the partitioning algorithm will then only allocate space if doing so reduces data movement vs. bypassing.



(a) Miss rate curves.

(b) Memory latency curves.

Figure 5-1: `mis`'s memory performance vs. VC size. Vertices cache well, but edges are streaming. Whirlpool bypasses edges and gives the cache to vertices.

`mis` has two pools, **vertices** and **edges**. **Vertices** cache well, but **edges** do not. Whirlpool gives the full cache to **vertices** and bypasses **edges**. This is only possible because of Whirlpool's static classification, which quickly identifies **edges** so they can be bypassed. Jigsaw does not separate accesses to **vertices** and **edges**, so it cannot allocate capacity specifically to **vertices** and it must always check the cache to maintain coherence.

Figure 5-2 analyzes `mis`'s performance, data movement energy, and LLC accesses. We compare Whirlpool against S-NUCA caches with LRU and DRRIP replacement; IdealSPD, an idealized private-baseline D-NUCA policy that is an upper-bound over several prior D-NUCAs; the shared-baseline, page-migration D-NUCA proposed by Awasthi et al. [2]; and Jigsaw, extended to support bypassing (see Section 4.2).

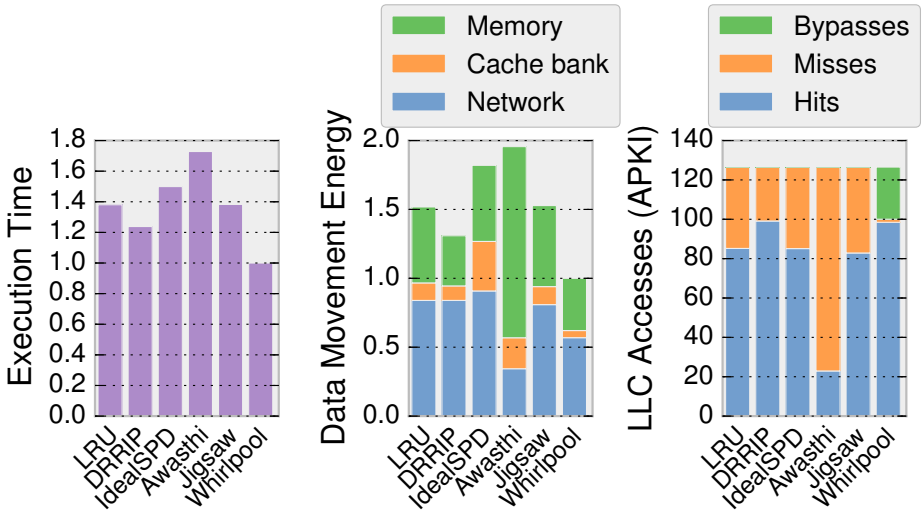
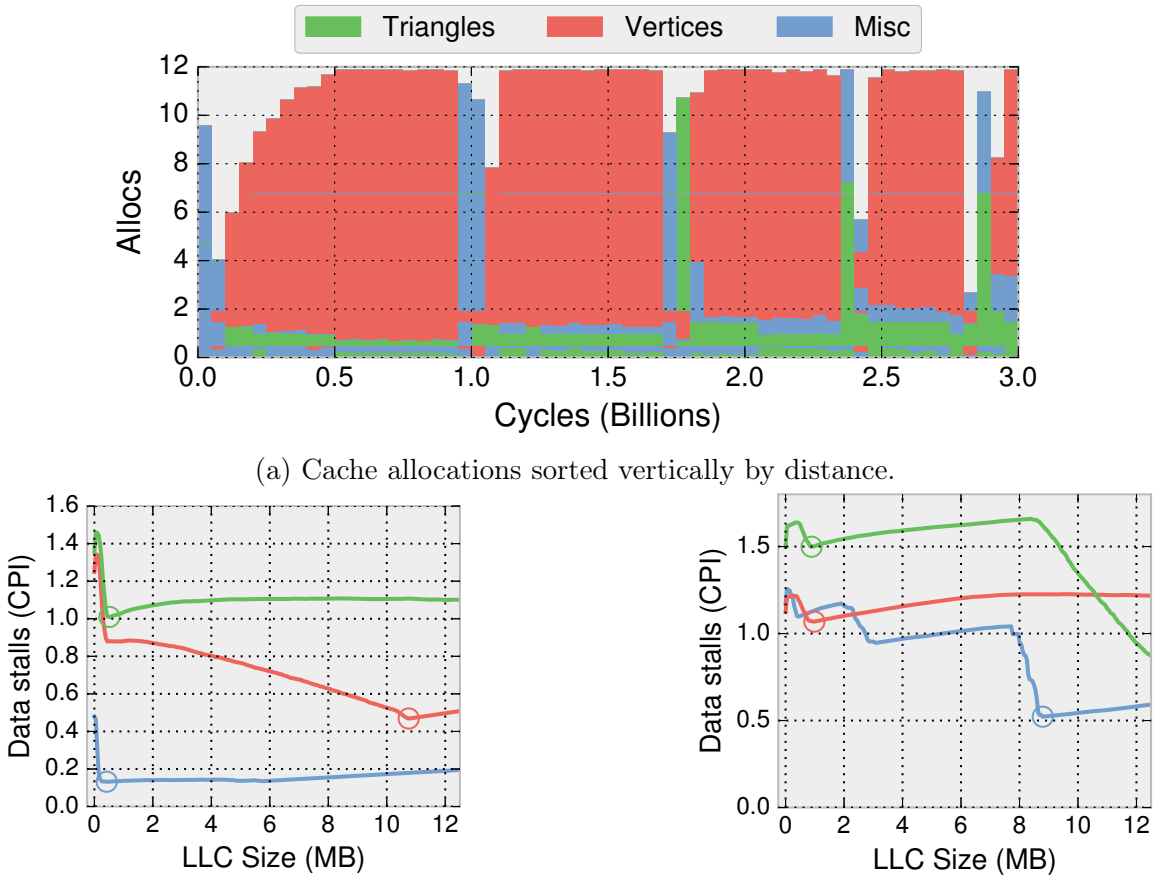


Figure 5-2: Breakdown of `mis`'s performance, energy, and accesses for different caching schemes.

Whirlpool reduces data movement because its static classification lets it adopt the right policy for each pool. Whirlpool gives enough space to fit **vertices**, achieving a similar hit rate to DRRIP and thus reducing memory energy. It also immediately bypasses accesses to **edges**, *without first checking a cache bank*. This reduces network and cache bank energy significantly over the other policies. In contrast, IdealSPD checks multiple banks (nearby banks first, then remote banks), and thus consumes the most energy. Meanwhile, Awasthi gets stuck at a small capacity allocation, incurring more misses than the other schemes. (Although Awasthi performs poorly on `mis` and several other benchmarks, it outperforms S-NUCA and saves energy on average; see

Chapter 6.) Whirlpool improves `mis`'s performance by 38% over Jigsaw and reduces data movement energy by 53%.

Whirlpool adapts to application phases: Unlike prior techniques that leverage static information through fixed policies, Whirlpool uses dynamic policies that can adapt to time-varying program behavior. Figure 2-4 showed one example for the SPEC CPU2006 benchmark `lbm`; Figure 5-3 shows another for the PBBS benchmark `refine` (Delaunay refinement).



(a) Cache allocations sorted vertically by distance.

(b) Latency curves at $\frac{1}{2}$ B cycles.

(c) Latency curves at 1 B cycles.

Figure 5-3: `refine` has irregular phase changes. Whirlpool dynamically adapts its allocations and placement to retain the data structures that have reuse.

`refine` accesses two main data structures, triangles and vertices, as well as other miscellaneous data in the misc pool. For most of `refine`'s execution, its working set fits easily on chip (Figure 5-3b). However, at irregular intervals, its behavior changes for roughly 100 M cycles: vertices becomes streaming, triangles starts fitting

on cache, and misc's working set increases substantially (Figure 5-3c).

Whirlpool adapts to this unpredictable behavior, changing its allocations and placement to retain the data structures that cache best. Figure 5-3a shows how Whirlpool allocates and places cache space for `refine`. Time is shown in cycles along the x-axis, and allocations are indicated by color along the y-axis. Additionally, allocations are sorted by distance from the core along the y-axis from bottom to top.

For most of `refine`'s execution, triangles and misc are given small allocations placed near the core, and vertices is given most of the remaining cache space. This placement minimizes data movement because it fits vertices in the cache, and accesses to triangles and misc *miss quickly*. In `refine`, bypassing triangles and misc is not advantageous (see Figure 5-3b), but placing them nearby helps by reducing network traffic.

During phase changes, this pattern inverts: vertices is streaming and is given a small allocation placed near the core, and the remaining cache space is given to either triangles or misc (depending on whether triangles fits).

5.4 Parallel applications

In addition to reducing data movement in sequential applications, Whirlpool benefits parallel applications by running tasks close to their data. Whirlpool makes small changes to task-parallel runtimes, letting it rapidly benefit many applications with minimal programmer burden.

Conventional work-stealing: Work-stealing [11] is the most widely-used scheduling technique for task-parallel programs. Each thread has a queue of ready tasks, to which it enqueues and dequeues work. When a thread runs out of work, it tries to steal tasks from a randomly-selected thread's queue. Work-stealing makes task enqueues and dequeues cheap and achieves good load balance, but, over time, each core ends up accessing data used by many tasks, hurting locality. Since work-stealing causes poor reference locality, D-NUCAs alone cannot achieve a good data placement [9]. For example, as shown in Figure 5-5, Jigsaw performs the same as S-NUCA because most data is accessed from multiple cores and mapped to the single process-level VC.

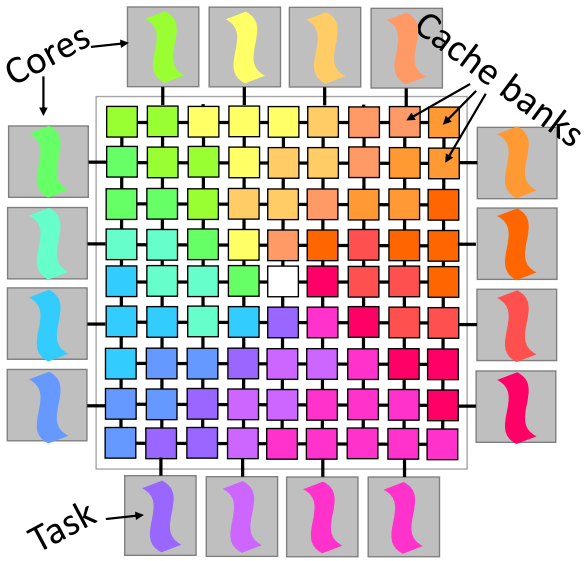
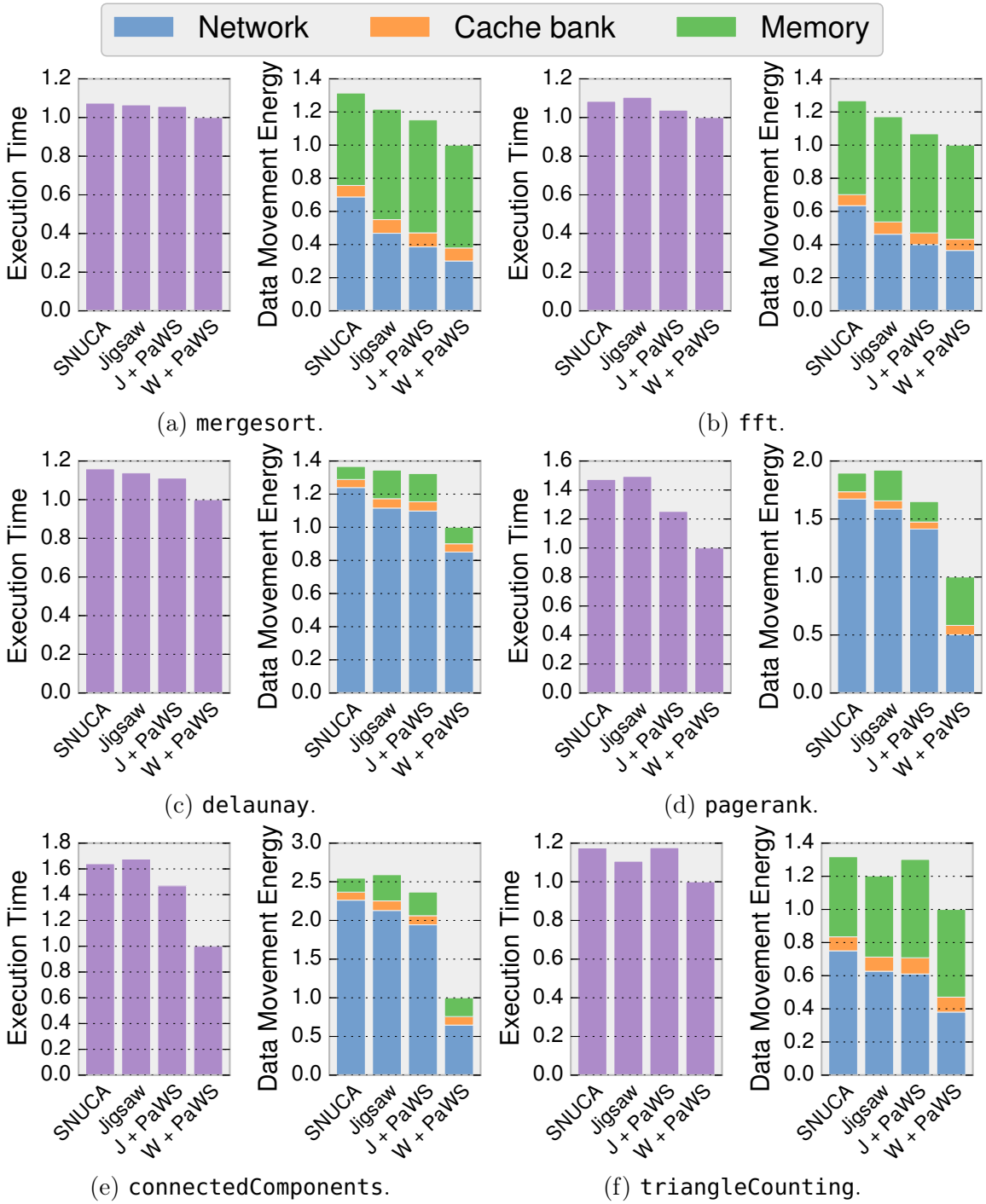


Figure 5-4: Partitioned work-stealing (PaWS) in Whirlpool on a 16-core system. In PaWS, each core works on a partition of the input and preferentially steals tasks from nearby cores. Colors indicate affinity between tasks and data.

Partitioned work-stealing (PaWS): Inspired by prior work on locality-aware placement and stealing [9, 14, 63], we develop simple extensions to improve reference locality. We leverage that, in many applications, the data accessed by each task is known when the task is created. PaWS partitions program data evenly among cores, and enqueues tasks to the core that has its input data instead of the thread's local queue, as shown in Figure 5-4. PaWS also preferentially steals tasks from neighboring cores instead of at random.

We evaluate PaWS on six memory-intensive applications from several benchmark suites: `mergesort` [53], `delaunay` [56], `fft` [23], `pagerank` [54], `connectedComponents` [3], and `triangleCounting` [3]. The first three use regular data structures that are trivial to evenly partition across cores. The last three are irregular graph algorithms, for which different partitionings can have a large impact on performance. We use METIS [32] to evenly partition their input graphs while minimizing the number of edges across partitions.

Figure 5-5 shows that PaWS improves performance moderately over Jigsaw when running on a 16-core system (up to 19% on `pagerank`). Jigsaw + PaWS improves performance because locality improves in the private caches and more data remains in the thread-private VC for longer. However, over time, work-stealing still causes



(a) `mergesort`.

(b) `fft`.

(c) `delaunay`.

(d) `pagerank`.

(e) `connectedComponents`.

(f) `triangleCounting`.

Figure 5-5: Performance of S-NUCA, Jigsaw, Jigsaw with PaWS, and Whirlpool with PaWS on parallel applications.

a large fraction of the data to be accessed from multiple cores, leading to poor data placement in Jigsaw (and other schemes, e.g. R-NUCA [25]).

Whirlpool with PaWS: Whirlpool makes it easy for PaWS to benefit from improved spatial placement in shared caches. We simply map data from each partition to a separate pool. Although load imbalance causes data to be accessed by multiple cores, each pool's VC is still placed close to the cores that use it. As shown in Figure 5-5, this results in much higher gains over Jigsaw: from 6.5% higher performance and 22% lower data movement energy on `mergesort`, to 67% higher performance and 2.6× lower data movement energy on `connectedComponents`.

In summary, Whirlpool with PaWS dramatically improves the performance and efficiency of parallel programs. Moreover, it requires only small changes to existing schedulers, and retains a familiar and productive programming model.

For graph algorithms like `PageRank`, the preprocessing step using METIS could be quite expensive. The overall runtime, including preprocessing time, will be lower than the original runtime only if the preprocessing overhead can be amortized over multiple iterations of the algorithm. This is in fact the case for iterative graph algorithms like `PageRank` and `Connected Components` that run for 10s of iterations to converge to the final solution. We can further reduce the overhead by using faster graph partitioning algorithms that trade-off partition quality for runtime. We can also use online schemes that infer a partitioning of the graph by observing the program's access patterns in the first few iterations. This partitioning can then be used to improve the performance of subsequent iterations. We leave the exploration of such techniques to future work.

Chapter 6

Automated Data Classification

While specifying pools manually gives full control to the programmer, modifying program code is not always practical. We now use the insights from Chapter 5 to design WhirlTool, a profile-guided tool that automatically classifies data into pools. WhirlTool works on unmodified binaries, often matches and sometimes outperforms our manual classification, and introduces small overheads. WhirlTool is publicly available at `http://people.csail.mit.edu/sanchez`.

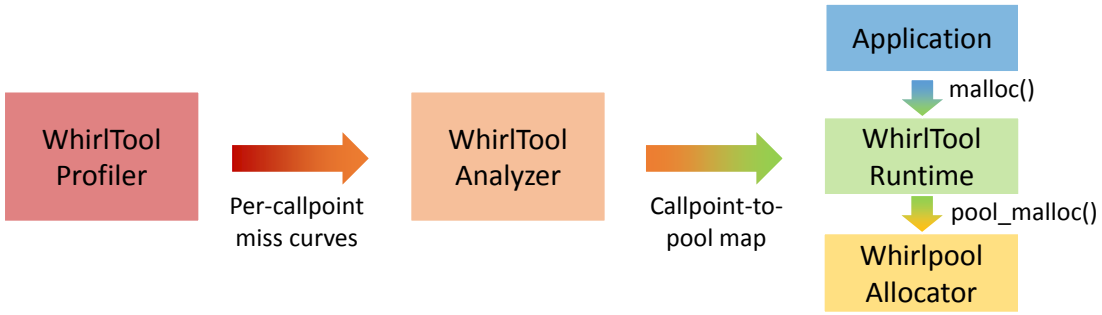


Figure 6-1: WhirlTool overview.

WhirlTool consists of three main components, shown in Figure 6-1. First, the *WhirlTool profiler* tracks a program's memory allocations and profiles their access patterns. Specifically, we sample their stack distance distributions at regular intervals. Second, the *WhirlTool analyzer* clusters allocations into pools using the profiled stack distributions. Third, the *WhirlTool runtime* replaces the default memory allocator and transparently maps each allocation to its assigned pool. Profiling and analysis are performed once (e.g., during compilation).

6.1 Profiler

To limit profiling information, Whirlpool identifies memory allocations by their *callpoint*, and profiles all allocations from the same callpoint as a single entity. This heuristic is motivated by our experience in manually porting applications, where we observed that semantically different data tend to be allocated from different points. Specifically, we produce each callpoint id by walking the stack and hashing the last two return PCs.

WhirlTool profiles applications to gather the miss rate curve of each callpoint [21, 57], then uses a distance metric based on miss rate curves to cluster callpoints into a small number of pools (discussed below). The profiler periodically records miss rate curves for all callpoints, which is important to distinguish allocations that are similar on average but whose behavior varies over time (e.g., `lbm` in Section 2.2).

We implement the profiler as a Pintool [41], though we note that profiling could be done in Jigsaw hardware directly. We sample miss rate curves every 50 M instructions. This produces 200 KB–1.25 MB of data on our benchmarks. We train WhirlTool with short runs (e.g., using SPEC CPU2006 train input sets) by default. As we show in Section 6.4, WhirlTool is quite robust to input set changes.

6.2 Analyzer

The *WhirlTool analyzer* progressively clusters callpoints into a small number of pools. Clustering uses a distance metric between pools that reflects *how many additional misses are incurred by clustering them.*

Distance metric: WhirlTool computes the distance between two pools by using their miss rate curves. First, WhirlTool estimates the *combined miss rate curve*, i.e., the curve that would result if both pools were grouped. Second, WhirlTool computes the *partitioned miss rate curve*, i.e., the curve that results from partitioning capacity between both pools. This results in fewer misses than the combined curve, since partitioning favors the pool that uses the cache best.

On a single interval, we define the distance between two pools as the area between their combined and partitioned curves. Figure 6-2 shows an example. We combine

a cache-friendly pool (m1) with two other pools (m2 and m3) in the left and right figures. In the left figure, both m1 and m2 cache well, so there is little penalty from combining them. This is reflected in the small difference between their combined and partitioned miss rate curves. However, in the right figure, m3 does not cache well, and it thus interferes more with m1. Combining these pools is a bad idea, since doing so will add many unnecessary misses to m1. This is reflected in the larger difference between their combined and partitioned miss rate curves.

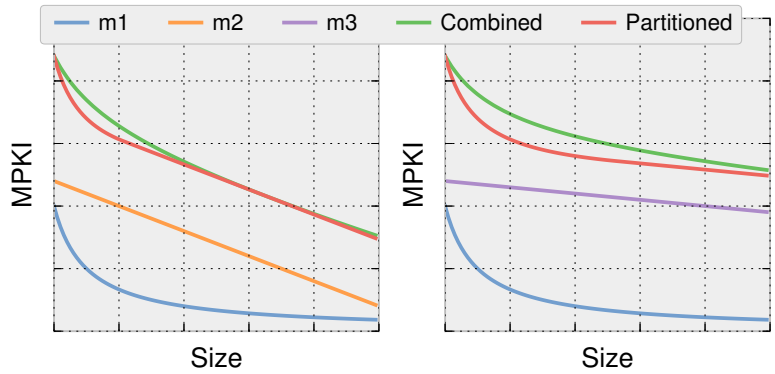


Figure 6-2: WhirlTool measures the distance between two pools as the additional misses incurred by combining the pools vs. partitioning them separately.

Finally, the distance between two pools is the sum of distances of their per-interval curves. This way, pools accessed in non-overlapping intervals have a small distance, even though they may have very distinct access patterns when active. This benefits programs that use different data over distinct phases, as they can use a small number of pools without degradation.

Modeling combined miss rate curves: Several prior models predict shared cache interference [13, 15, 22, 62], but these are somewhat complex and computationally expensive. We instead develop a simpler model that lets WhirlTool rapidly estimate the effect of combining pools.

We model the combined miss rate curve using the *flow* of lines through the cache. For simplicity, consider LRU replacement. The idea behind flow is that lines enter the cache at MRU, and are pushed towards LRU by other lines entering the cache, until they are eventually evicted. Flow is the rate that lines are being pushed towards LRU. However, flow is not constant—hits promote lines rather than evicting them, so

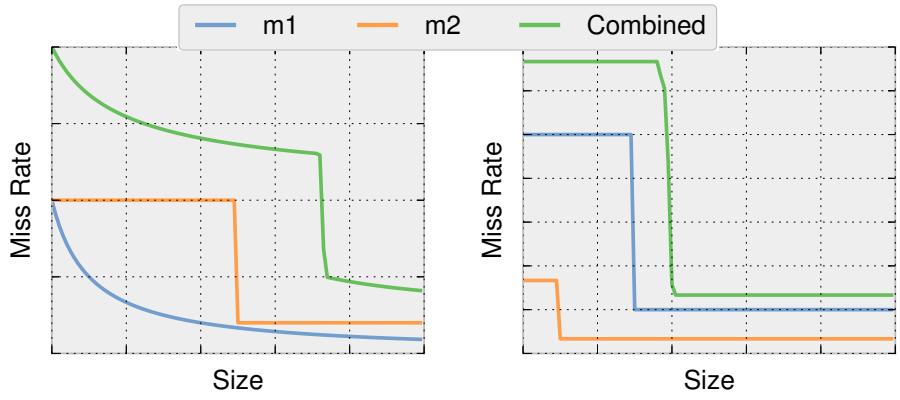
flow decreases as lines hit (see Figure 6-3). Hence, *the flow at a given point in the miss rate curve is equal to the miss rate at that size.*

Flow is useful because it gives a simple way to combine miss rate curves: when two pools are merged, accesses from either pool push lines from both pools towards LRU. In other words, flow is additive. But the rate at which lines are pushed depends on both their fraction of flow—infrequently-accessed pools have little effect on the combined miss rate curve—and how far they have already been pushed. Listing 6.1 gives pseudocode for the model.

```
def combineMissCurves(m1, m2):
    s1, s2 = 0, 0
    for s = 0 to N:
        m[s] = m1[s1] + m2[s2]
        s1 += m1[s1] / m[s]
        s2 += m2[s2] / m[s]
    return m
```

Listing 6.1: Simple model for combining miss rate curves.

One way to think about this model is that it has a single "write head" at `s` and two "read heads" at `s1` and `s2`. At each step, it writes `m` by reading the input miss rate curves at `m1` and `m2`, then moves the read heads through their input curves according to their relative flows. Figure 6-3a shows an example.



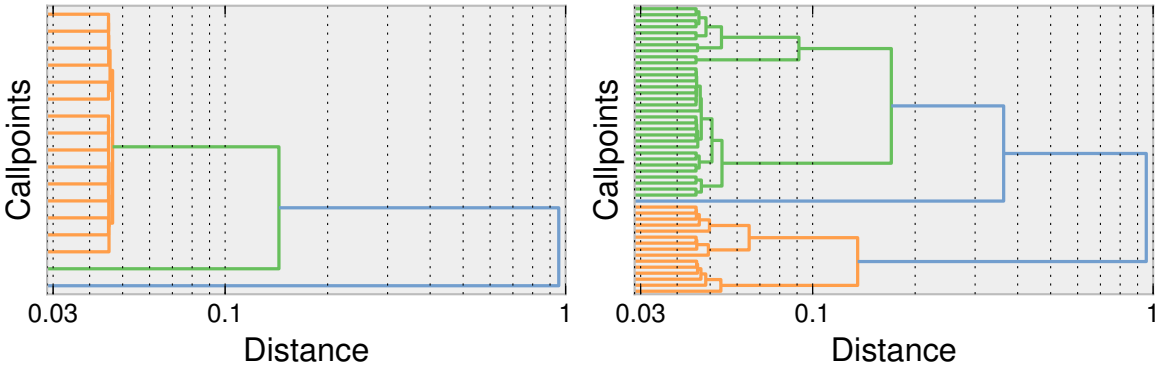
(a) Combining `m1` and `m2`. (b) Combining similar pools.

Figure 6-3: WhirlTool's simple model to combine miss rate curves.

This model has several desirable properties. It is commutative and associative, so

the order in which pools are combined does not matter. It will correctly recombine similar access patterns into a similar result, so the model is insensitive to arbitrary divisions of single pool into subpools (see Figure 6-3b). It also produces small changes when adding a pool that is accessed infrequently.

Partitioned miss curve: In principle, we could find the optimal partitioning between both pools at every size, but in practice doing so is computationally expensive [7, 49]. Instead, we compute the convex hulls of each input miss rate curve (a linear-time operation [42]), and then partition the full capacity in a single pass using convex optimization (i.e., hill climbing). This performance could be practically realized by using partitioning within each VC to achieve convex performance [8].



(a) `dt`. (b) `omnetpp`.

Figure 6-4: Hierarchical clustering with WhirlTool. Each graph shows the distance (x-axis) among callpoints and clusters (y-axis). Colors indicate how WhirlTool clusters callpoints into 3 pools.

Agglomerative clustering: WhirlTool uses a simple algorithm to cluster callpoints into pools. It first places each callpoint in a separate pool, and computes the pairwise distances between all pools. Then it proceeds iteratively. Each iteration merges the two closest pools, and computes the distance of the resulting pool to all the remaining pools. The result is a hierarchical clustering that gives the callpoint-to-pool mapping for different numbers of desired pools, as shown in Figure 6-4. This procedure takes $O(n^2)$ time with n callpoints, but we find the runtime acceptable (a few seconds) for the applications we evaluate, which have 10s-100s of callpoints. In most applications, we observed that 2-4 pools suffice to capture most of Whirlpool's benefits (Section 6.4).

6.3 Runtime

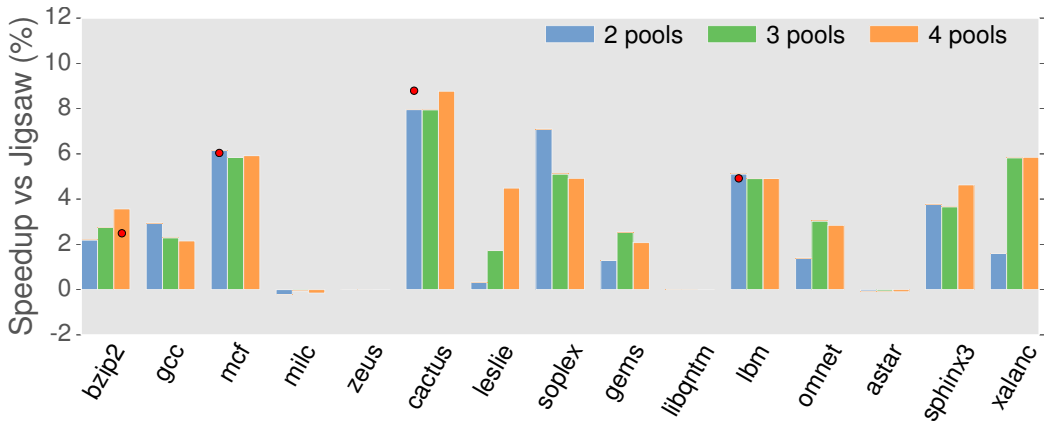
WhirlTool's runtime replaces the system's memory allocator. On each allocation call, the tool finds the callpoint id and calls the Whirlpool allocator with the corresponding pool. Allocations from an unprofiled callpoint use the thread-private pool. This instrumentation incurs small overheads, at most 0.01% over all our benchmarks (some of which have frequent allocations).

6.4 Analysis

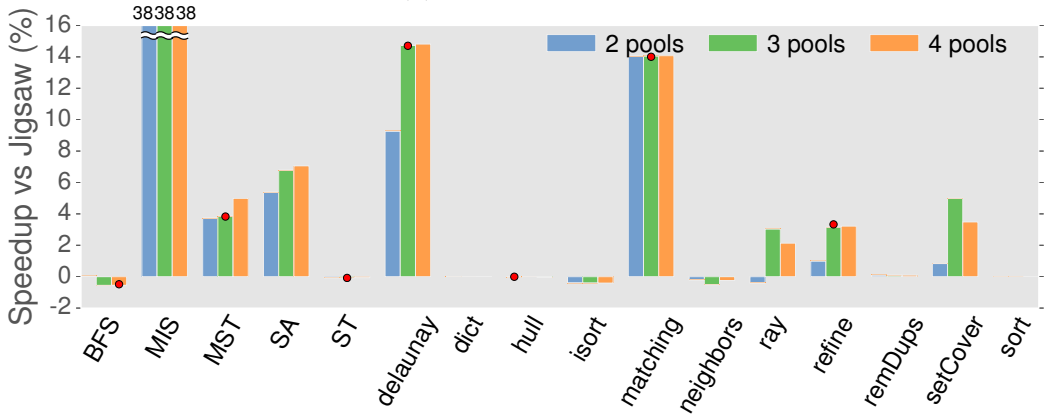
Sensitivity to the number of pools: Figure 6-5 shows how WhirlTool's performance changes with the number of pools. Each group of bars show the performance of a specific application over Jigsaw. Each bar in the group shows performance for a given number of pools, from 2 to 4. We include SPEC CPU2006 and single-threaded PBBS applications running with their largest input sets. WhirlTool uses profiling data from the train input sets for SPEC CPU2006 and the small input sets for PBBS applications. For manually-ported applications (Chapter 5), a dot shows the number of pools used by manual classification (x-axis) and the performance it achieves (y-axis). As we can see, performance improves by 5-15% for several applications, and `mis` is 38% faster.

In general, moving from 2 to 3 pools improves performance somewhat on a few applications, while 4 pools shows negligible improvements. Some applications (e.g., `gcc`, `soplex`) show a slight decrease in performance with more pools. This happens because these applications have significant variability, and partitioning their data more finely makes phase changes somewhat worse. Given these results, we consider 3 pools to be the right tradeoff, and use 3 pools in subsequent results.

WhirlTool vs. manual classification: Figure 6-5 also shows that WhirlTool matches the performance of manual classification for most applications, and outperforms it in some cases (e.g., `bzip2`). Only `cactus` performs slightly worse with automatic classification.



(a) SPECCPU2006.



(b) PBBS.

Figure 6-5: Speedup of WhirlTool over Jigsaw with 2, 3, and 4 pools. A red dot shows the performance achieved by manual classification for the applications we ported by hand (Table 5.1).

Sensitivity to training data: WhirlTool's performance is robust across input sets on most applications. To quantify this, we compare WhirlTool's performance when using the default training input sets (train/small) for profiling vs. the full input sets used in our experiments (ref/large). WhirlTool's performance is only significantly different in 4 out of the 31 applications, shown in Figure 6-6. In these applications, the training inputs result in lower performance than when using the full inputs, e.g. by 5.5% for `leslie` and by 6.7% for `omnet`. This happens because the training inputs exhibit different access patterns than the full inputs. However, over all benchmarks WhirlTool is robust to different inputs, yielding just 0.4% lower performance on training inputs.

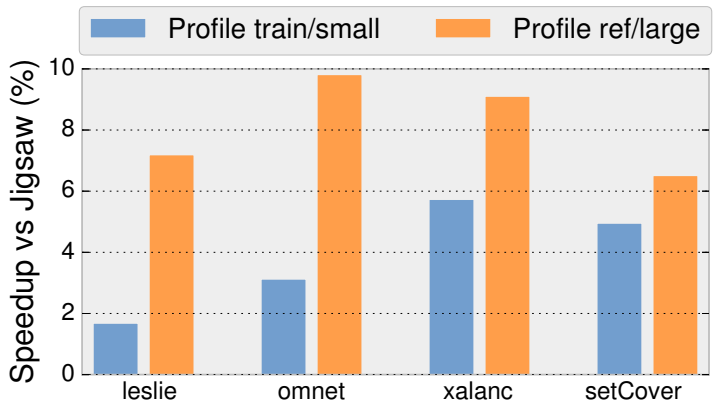


Figure 6-6: WhirlTool's sensitivity to training inputs.

6.5 Evaluation

Figure 6-7 and Figure 6-8 give two examples of how WhirlTool achieves similar benefits to manual classification.

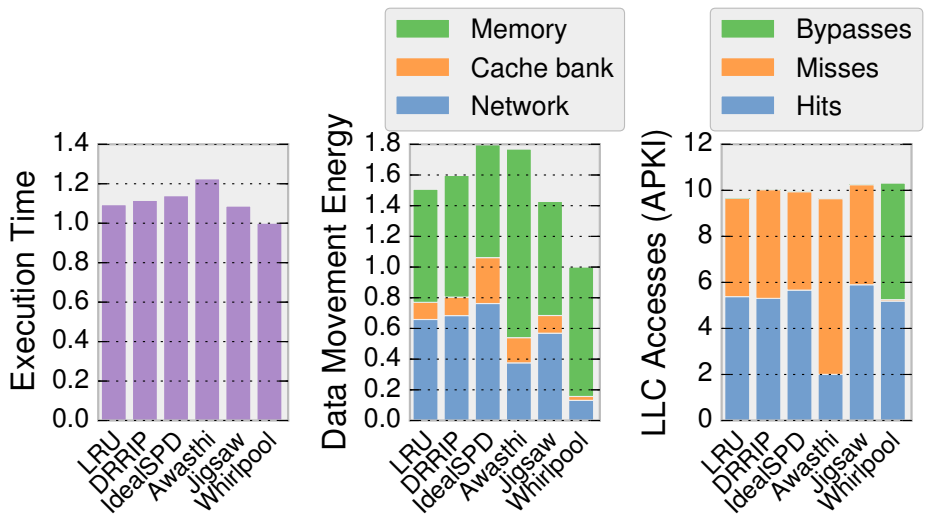


Figure 6-7: Breakdown of `cactus`'s performance, energy, and accesses for different caching schemes.

`cactus` has two memory regions, only one of which gets good reuse. WhirlTool correctly identifies these pools, letting Whirlpool cache the former near the core and bypass the latter (Figure 6-7). Meanwhile, Jigsaw cannot distinguish between pools and must use more cache banks to retain the working set. As a result, Whirlpool significantly reduces network traffic over Jigsaw, and reduces overall data movement energy by 42%. Reducing network traffic also reduces network latency, and Whirlpool improves performance over Jigsaw by 8.6%.

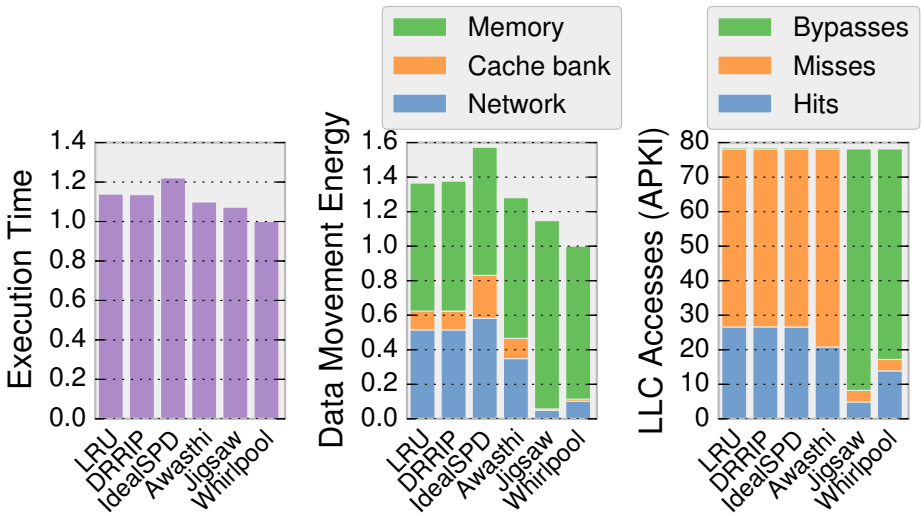


Figure 6-8: Breakdown of `SA`'s performance, energy, and accesses for different caching schemes.

`SA` offers an interesting contrast. Rather than using fewer banks to reduce network latency over Jigsaw, Whirlpool uses more banks to reduce cache misses (Figure 6-8). WhirlTool identifies the pools in `SA` that cache well, and Whirlpool can thus retain more of the working set and reduce main memory accesses. But in order to do so, it uses more banks—which can be seen in the higher network energy. Overall, this is a good tradeoff, and Whirlpool reduces data movement energy by 15% over Jigsaw while improving performance by 7.3%.

Single-threaded applications: We now extend these case studies across many benchmarks. Figure 6-9 compares Whirlpool's overall performance and data movement energy with S-NUCA, IdealSPD, Awasthi et al., and Jigsaw over the 31 memory-intensive applications from SPEC CPU2006 and PBBS.

Individual applications see large improvements, e.g., up to 53% lower data movement energy and 38% speedup in `mis`. However, average gains are more muted: Whirlpool reduces data movement energy over Jigsaw by 8.0%, and improves performance by 3.9%. As currently coded, many applications do not expose their memory access heterogeneity across different address regions. With more careful coding, WhirlTool may be able to extract more heterogeneity and improve performance further. As shown in Figure 6-9, Whirlpool achieves this by *(a)* placing data closer, which reduces network energy and latency, *(b)* caching data that is more likely to hit, and *(c)*

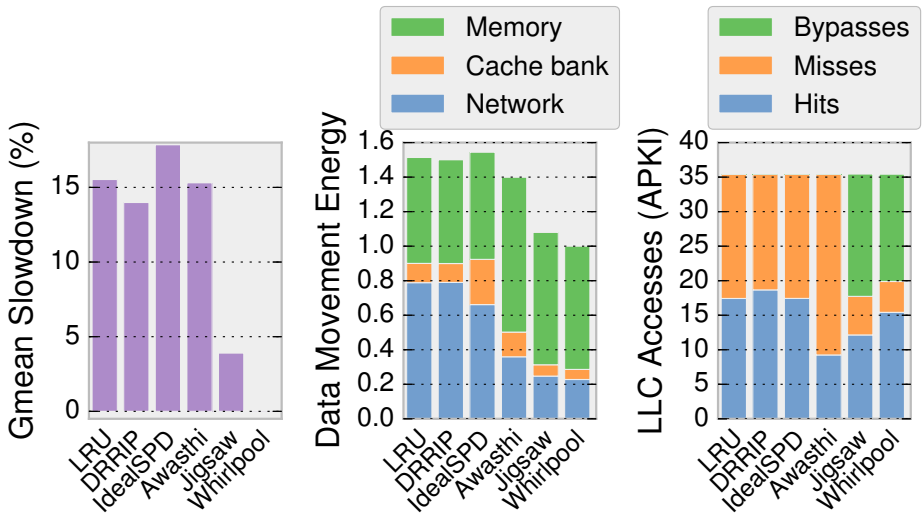


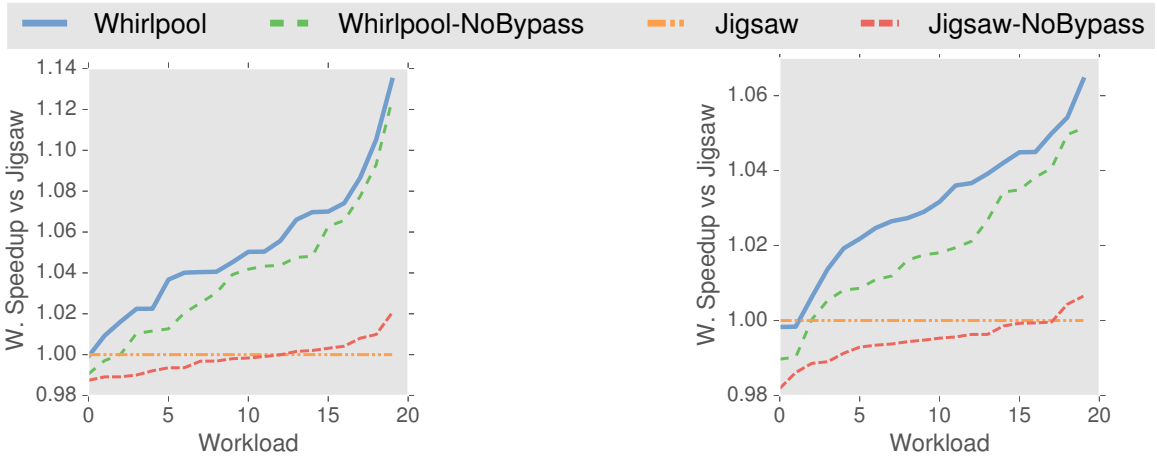
Figure 6-9: Breakdown of overall performance, energy, and accesses for different caching schemes across all single-threaded benchmarks.

bypassing more selectively than Jigsaw. By contrast, S-NUCA with LRU incurs 51% more data movement energy than Whirlpool and 15% worse performance; S-NUCA with DRRIP incurs 50% more data movement energy and 14% worse performance; IdealSPD incurs 54% more data movement energy and 18% worse performance; and Awasthi incurs 40% more data movement energy and 15% worse performance. However, while S-NUCA variants are generally slower, IdealSPD has a more bimodal behavior: it performs close to Jigsaw on benchmarks that fit within its private region (e.g., `bzip2`), but performs the worst of all schemes on benchmarks that do not fit due to unnecessary multi-level lookups that slow down misses and add data movement energy. Similarly, Awasthi performs much better than S-NUCA on benchmarks with small working sets, but performs poorly on benchmarks that need more than four cache banks (Awasthi's initial allocation). As a result, Awasthi significantly reduces network latency and energy but incurs more misses than S-NUCA.

Figure 6-9 also shows that, while Whirlpool and Jigsaw both benefit from bypassing, Whirlpool benefits more because it can distinguish and bypass pools with no reuse. Without bypassing, Jigsaw is 0.2% slower, while Whirlpool is 1.2% slower.

Multi-programmed mixes: We run 20 mixes of randomly-chosen, memory-intensive SPEC CPU2006 applications, at both 4 and 16 cores. Figure 6-10 shows the distribution of weighted speedups in both cases. Each line shows the weighted speedup

of a single scheme over the Jigsaw baseline, sorted along workload mixes (x-axis) by improvement (inverse CDF). Whirlpool outperforms Jigsaw by up to 13% at 4 cores (5.1% gmean), by up to 6.4% at 16 cores (3.0% gmean), and improves performance consistently. Improvements are larger with fewer cores because, with more applications, Jigsaw has many choices to improve cache performance even with a single VC per application.



(a) 4 cores.

(b) 16 cores.

Figure 6-10: Weighted speedup of Whirlpool over Jigsaw for 4- and 16-core systems.

Other schemes perform considerably worse. On 4- and 16-core mixes, Whirlpool outperforms S-NUCA by 32%/62% respectively, DRRIP by 25%/52%, IdealSPD by 30%/50%, and Awasthi by 18%/25%. This is because Jigsaw gathers detailed information about each VC over all possible allocations, allowing it to carefully optimize data placement at each reconfiguration.

Chapter 7

Conclusion

This thesis has presented Whirlpool, a classification-based approach to manage distributed caches. Whirlpool statically classifies data into different pools, which allows dynamic policies to tune the cache to each pool. In this way, it conveys semantic, application-level information about memory usage without fixing the caching policy. This can be done using a simple API that allows programmers to classify data or a profiling tool that works in unmodified binaries and achieves similar performance to manual classification. Whirlpool also improves the performance of parallel programs by executing tasks close to the data they use.

Conventional hardware techniques have ignored application-level semantics, sacrificing performance and efficiency to reduce programmer burden. As future systems become limited by data movement rather than compute performance, it becomes essential to exploit application-level information to reduce data movement. Whirlpool has demonstrated that it is possible to achieve significant gains by using a combination of simple hardware and software mechanisms, while minimizing programmer burden.

Whirlpool opens interesting avenues for future work that co-adapts applications and the memory system, especially for heterogeneous memory systems and applications with irregular, time-varying memory access patterns. Applications can dynamically adapt their behavior to improve locality of their memory access pattern. This will increase the benefits of Whirlpool's mechanisms and further reduce data movement. Whirlpool's techniques can also be applied to other aspects of the memory system, like prefetchers, replacement policies, or memory access scheduling policies.

Bibliography

[1] K. Aingaran, D. Smentek, T. Wicki, S. Jairath, G. Konstadinidis, S. Leung *et al.*, "M7: Oracle's Next-Generation Sparc Processor," *IEEE Micro*, no. 2, 2015.

[2] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *Proc. HPCA-15*, 2009.

[3] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv:1508.03619 [cs.DC]*, 2015.

[4] B. M. Beckmann, M. R. Marty, and D. A. Wood, "ASR: Adaptive selective replication for CMP caches," in *Proc. MICRO-39*, 2006.

[5] B. M. Beckmann and D. A. Wood, "Managing wire delay in large chip-multiprocessor caches," in *Proc. MICRO-37*, 2004.

[6] N. Beckmann, "Design and analysis of spatially-partitioned shared caches," Ph.D. dissertation, Massachusetts Institute of Technology, 2015.

[7] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proc. PACT-22*, 2013.

[8] N. Beckmann and D. Sanchez, "Talus: A Simple Way to Remove Cliffs in Cache Performance," in *Proc. HPCA-21*, 2015.

[9] N. Beckmann, P.-A. Tsai, and D. Sanchez, "Scaling Distributed Cache Hierarchies through Computation and Data Co-Scheduling," in *Proc. HPCA-21*, 2015.

[10] K. Beyls and E. D'Hollander, "Generating cache hints for improved program efficiency," *J. Syst. Architect.*, vol. 51, no. 4, 2005.

[11] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, 1999.

[12] J. Brock, X. Gu, B. Bao, and C. Ding, "Pacman: Program-assisted cache management," in *Proc. ISMM*, 2013.

[13] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proc. HPCA-11*, 2005.

[14] Q. Chen, M. Guo, and H. Guan, "LAWS: locality-aware work-stealing for multi-socket multi-core architectures," in *Proc. ICS*, 2014.

[15] X. E. Chen and T. M. Aamodt, "A first-order fine-grained multithreaded throughput model," in *Proc. HPCA-15*, 2009.

[16] Z. Chishti, M. D. Powell, and T. Vijaykumar, "Optimizing replication, communication, and capacity allocation in CMPs," in *Proc. ISCA-32*, 2005.

[17] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *Proc. MICRO-39*, 2006.

[18] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *Proc. PLDI*, 1995.

[19] W. J. Dally, "GPU Computing: To Exascale and Beyond," in *Supercomputing '10, Plenary Talk*, 2010.

[20] S. Das, T. M. Aamodt, and W. J. Dally, "SLIP: reducing wire energy in the memory hierarchy," in *Proc. ISCA-42*, 2015.

[21] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proc. PLDI*, 2003.

[22] D. Eklov, D. Black-Schaffer, and E. Hagersten, "StatCC: a statistical cache contention model," in *Proc. PACT-19*, 2010.

[23] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *Proc. PLDI*, 1998.

[24] X. Gu, T. Bai, Y. Gao, C. Zhang, R. Archambault, and C. Ding, "P-OPT: Program-directed optimal cache management," in *Proc. LCPC-21*, 2008.

[25] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *Proc. ISCA-36*, 2009.

[26] E. Herrero, J. González, and R. Canal, "Distributed cooperative caching," in *Proc. PACT-17*, 2008.

[27] E. Herrero, J. González, and R. Canal, "Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors," in *Proc. ISCA-37*, 2010.

[28] A. Hilton, N. Eswaran, and A. Roth, "FIESTA: A sample-balanced multi-program workload methodology," *Proc. MoBS*, 2009.

[29] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr, and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proc. PACT-17*, 2008.

[30] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. ISCA-37*, 2010.

[31] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. ISCA-17*, 1990.

[32] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, 1998.

[33] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, no. 5, 2011.

[34] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. ASPLOS-X*, 2002.

[35] R. Komuravelli, M. D. Sinclair, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have your scratchpad and cache it too," in *Proc. ISCA-42*, 2015.

[36] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. ASPLOS-IV*, 1991.

[37] D. Lea, "A memory allocator," http://gee.cs.oswego.edu/dl/html/malloc.html, 2000.

[38] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing memory systems for chip multiprocessors," in *Proc. ISCA-34*, 2007.

[39] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. MICRO-42*, 2009.

[40] J. Lira, C. Molina, and A. González, "LRU-PEA: A smart replacement policy for non-uniform cache architectures on chip multiprocessors," in *Proc. ICCD*, 2009.

[41] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*, 2005.

[42] A. A. Melkman, "On-line construction of the convex hull of a simple polyline," *Information Processing Letters*, vol. 25, no. 1, 1987.

[43] Micron, "1.35V DDR3L power calculator (4Gb x16 chips)," 2013.

[44] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. ASPLOS-V*, 1992.

[45] S. Palanca, V. Pentkovski, S. Tsai, and S. Maiyuran, "Method and apparatus for implementing non-temporal stores," 2001, US Patent 6,205,520.

[46] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. Kozuch *et al.*, "Exploiting compressed block size as an indicator of future reuse," in *Proc. HPCA-21*, 2015.

[47] P. Petoumenos, G. Keramidas, and S. Kaxiras, "Instruction-based reuse-distance prediction for effective cache management," in *Proc. SAMOS*, 2009.

[48] M. K. Qureshi, "Adaptive spill-receive for robust high-performance caching in CMPs," in *Proc. HPCA-15*, 2009.

[49] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO-39*, 2006.

[50] S. Rusu, H. Muljono, D. Ayers, S. Tam, W. Chen, A. Martin *et al.*, "Ivytown: A 22nm 15-core enterprise Xeon® processor family," in *Proc. ISSCC*, 2014.

[51] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *Proc. ISCA-38*, 2011.

[52] D. Sanchez and C. Kozyrakis, "ZSim: fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. ISCA-40*, 2013.

[53] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *Proc. ASPLOS-XV*, 2010.

[54] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan *et al.*, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proc. SIGMOD*, 2014.

[55] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proc. High Performance Computing for Computational Science (VECPAR)*, 2010.

[56] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri *et al.*, "Brief announcement: the problem based benchmark suite," in *Proc. SPAA*, 2012.

[57] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations," in *Proc. ASPLOS-XIV*, 2009.

[58] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *Proc. MICRO-28*, 1995.

[59] Z. Wang, K. McKinley, A. L. Rosenberg, and C. C. Weems, "Using the compiler to improve cache replacement decisions," in *Proc. PACT-11*, 2002.

[60] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proc. PLDI*, 1991.

[61] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "SHiP: signature-based hit predictor for high performance caching," in *Proc. MICRO-44*, 2011.

[62] X. Xiang, B. Bao, C. Ding, and Y. Gao, "Linear-time modeling of program working set in shared cache," in *Proc. PACT-20*, 2011.

[63] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, "Locality-aware task management for unstructured parallelism: A quantitative limit study," in *Proc. SPAA*, 2013.

[64] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *Proc. ISCA-32*, 2005.